Apunte de clase

Seguridad Informática

Maximiliano Cristiá

Licenciatura en Ciencias de la Computación

Facultad de Ciencias Exactas, Ingeniería y Agrimensura

Universidad Nacional de Rosario

Rosario – Argentina

© Maximiliano Cristiá – 2021-2025 – Todos los derechos reservados

Índice

1. Unidad I: Introducción **3**

2. Unidad II: Confidencialidad en sistemas de cómputo **9**
2.1. Seguridad multi-nivel . . . 10
2.2. Arquitectura de software para seguridad . . . 12
2.3. Modelos de confidencialidad . . . 13
2.4. Modelo de Bell-LaPadula . . . 14
2.4.1. Los problemas de BLP . . . 18
2.5. No inteferencia . . . 19
2.6. Análisis de la no interferencia . . . 21
2.6.1. Canales encubiertos . . . 23
2.6.2. Hipótesis detrás del modelo de no interferencia . . . 26
2.6.3. Sistemas manifiestamente seguros . . . 27
2.7. Multi-ejecución segura . . . 28

3. Unidad III: Seguridad en lenguajes de programación **34**
3.1. Desbordamiento de arreglos . . . 34
3.1.1. El mecanismo SUID . . . 35
3.1.2. Detalles técnicos del desbordamiento de arreglos . . . 36
3.1.3. Contramedidas . . . 41
3.1.4. Conclusiones sobre el ataque por desbordamiento de arreglos . . . 43
3.2. Inyección de sentencias SQL . . . 44
3.2.1. Contramedidas . . . 46
3.2.2. Conclusiones sobre el ataque por ISQL . . . 47
3.3. Cuando *security* es *safety*: proof-carrying code . . . 48
3.4. Cuando *security* es *safety*: un sistema de tipos para flujo de información seguro . . . 53

4. Unidad IV: Introducción a la criptografía aplicada **56**
4.1. Criptografía simétrica . . . 60
4.1.1. DES . . . 61
4.1.2. Modos de operación de DES . . . 64
4.2. Criptografía asimétrica . . . 64
4.2.1. RSA . . . 66
4.2.2. Análisis de RSA . . . 67
4.2.3. Firma digital o electrónica . . . 68
4.2.4. Certificados digitales y autoridades de certificación . . . 69
4.3. Funciones *hash* criptográficas . . . 70
4.4. Aplicaciones . . . 71
4.4.1. Autenticación en UNIX (y algo más) . . . 71
4.4.2. Integridad de datos . . . 73
4.4.3. Negociación de una clave simétrica . . . 74
4.4.4. El protocolo de Needham-Schroeder (versión simétrica) . . . 75
4.4.5. El protocolo Secure Sockets Layer . . . 78
4.4.6. Criptografía y no interferencia . . . 81

5. Unidad V: Introducción al análisis de protocolos criptográficos **81**

1. Unidad I: Introducción

La definición clásica de seguridad informática es la siguiente.

Definición 1 (Seguridad Informática)**.** *La seguridad informática es la rama de la informática que se encarga de estudiar cómo preservar la* confidencialidad, integridad *y* disponibilidad *de datos y programas en sistemas de cómputo.*

Como vemos, la seguridad informática se entiende como la combinación de tres atributos de calidad: confidencialidad, integridad y disponibilidad. Estos atributos los definimos de la siguiente forma.

Definición 2 (Confidencialidad, integridad y disponibilidad)**.** *Preservar la confidencialidad de los datos significa que solo los usuarios autorizados puedan* ver *dichos datos. Preservar la integridad de los datos y programas significa que solo los usuarios autorizados y solo por los mecanismos autorizados puedan* modificar *dichos datos y programas. Preservar la disponibilidad de datos y programas significa que los usuarios autorizados puedan usar los datos y programas cada vez que los necesiten.*

Las definiciones anteriores hacen mención a los *usuarios* y a los *datos y programas*. Desde un punto de vista más conceptual se utilizan los términos *sujeto* y *objeto*. Un sujeto es una entidad activa capaz de acceder información o programas en un sistema de cómputo; un objeto es o bien un sujeto o bien cualquier contenedor de información o código. Posibles sujetos son: personas, procesos, computadoras; posibles objetos son archivos, directorios, *sockets* y cualquiera de los sujetos. Algunos autores utilizan el término *agente* como sinónimo de sujeto.

Notar que el atributo de integridad menciona dos condiciones: que solo los sujetos autorizados modifiquen objetos y que lo hagan solo de las formas autorizadas. Observar que los otros dos atributos solo exigen una condición cada uno. Veamos el siguiente ejemplo.

Ejemplo 1 (Integridad)**.** *Consideremos un sistema de sueldos. Solo el contador de la empresa está autorizado a modificar el sueldo de un empleado. Sin embargo el contador no puede hacer la modificación de cualquier forma: debe utilizar un ABM de sueldos. Por ejemplo, no puede acceder a la base de datos vía SQL y hacer la modificación. Si el sistema le permite al contador modificar un sueldo vía SQL o le permite a otros usuarios ejecutar el ABM de sueldos y modificar sueldos, el sistema no preserva la integridad de los sueldos.*

La idea de que los sueldos solo puedan modificarse por medio del ABM de sueldos se basa en que la modificación de un sueldo puede requerir, precisamente, ciertos controles de integridad. Por ejemplo, no debería ser posible reducir el sueldo de un empleado por debajo del sueldo mínimo fijado por el Estado. Entonces dichos controles están implementados en el ABM de sueldos.

Por el contrario, la confidencialidad de los sueldos se viola cada vez que un usuario no autorizado puede conocer el sueldo de un empleado (por medios informáticos). No importa qué mecanismos se utilicen (por ejemplo, acceso vía SQL o vía el ABM de sueldos), si el sueldo es revelado la confidencialidad ya está comprometida.

En este curso desarrollaremos con cierta profundidad lo concerniente al atributo de confidencialidad y pondremos menos énfasis en lo relativo a integridad, mientras que el atributo de disponibilidad no será abordado. En parte esta decisión se basa en que no podemos abarcar toda la seguridad informática en un curso acotado pero también en una cuestión de fundamento. En efecto, las técnicas para proteger un sistema de cómputo contra ataques a la confidencialidad y la integridad son muy diferentes de aquellas que se utilizan para proteger la disponibilidad. Para los dos primeros atributos es suficiente con *restringir* el uso del sistema, pero se requiere lo

contrario para el tercero. Es decir, no es posible, en general, asegurar la disponibilidad *restringiendo* el uso del sistema puesto que la disponibilidad trata, precisamente, sobre usar el sistema lo más posible. En consecuencia las técnicas que se usan en confidencialidad e integridad son *fundamentalmente* diferentes a las que se usan en disponibilidad. Por ejemplo, la técnica más utilizada para confidencialidad e integridad es el control de acceso, que no sirve para la disponibilidad. Simétricamente, la técnica más utilizada para la disponibilidad es la redundancia (de hardware, datos, procesos, etc.) que no aporta nada (e incluso puede ser dañina) a los otros dos atributos.

Implícitamente, todo lo referido a la seguridad informática asume que los sistemas de cómputo están instalados en un *entorno* o *universo malicioso* u *hostil*. Esto es muy distinto al escenario clásico de la Ingeniería de Software donde se asume (también implícitamente) que el entorno es *benigno*. Si bien a primera vista esta diferencia puede no parecer importante, termina produciendo consecuencias sustanciales en la forma de estudiar el problema de la seguridad informática. Es tan así que técnicas probadas y aceptadas de la Ingeniería de Software no suelen ser suficientes para abordar el problema de la seguridad informática. Veremos más sobre estas cuestiones conceptuales más adelante.

La existencia de un universo hostil implica la existencia de *adversarios*, *amenazas*, *vulnerabilidades* y *ataques*. Si bien son conceptos que se estudian con cierta profundidad y son muy relevantes al problema de la ingeniería de seguridad informática [27, 16, 6], en este curso solo los utilizaremos con sus significados intuitivos. Lo que sí nos interesa resaltar es que un error en un programa instalado en un universo hostil se transforma en una vulnerabilidad que eventualmente puede ser usada por un adversario para montar un ataque. Ese mismo error, pero asumiendo un universo benigno, es simplemente un error más en el programa.

Ejemplo 2 (Desborde de arreglo)**.** *Digamos que un programa controla la longitud de los arreglos de caracteres buscando el caracter* `'\0'`*. Los programadores asumen que todos los arreglos de caracteres que se manejan en el programa eventualmente contienen ese caracter. Si uno de los programadores olvida o por omisión no usa tal convención entonces una rutina como la que sigue:*

```
while(a[i] != '\0') {
  ...
  i := i + 1;
  ...
}
```

generará un desborde de arreglo[1] *porque la condición jamás será falsa.*

En el universo benigno ese desborde de arreglo eventualmente hará que el programa termine con un error de violación de segmento[2]*. Cuando el usuario vea el mensaje de error o vea que la ventana donde corre el programa se cierra inesperadamente, informará el evento y nada más. Por el contrario, en un universo hostil, el mismo error en el mismo programa se convierte en una vulnerabilidad pues el adversario podría montar un ataque por desbordamiento de arreglo (el tema se desarrollará en la Unidad 3).*

La existencia de un universo hostil implica una diferencia fundamental entre las propiedades de *safety* y *security*. Todo lo que hemos visto durante los cursos de Ingeniería de Software I y II apunta a resolver el problema de *safety* de programas. Resolver el problema de *security* plantea nuevos desafíos o, en el mejor de los casos, los mismos desafíos pero bajos condiciones más

[1] 'buffer overflow', en inglés.
[2] 'segmentation fault', en inglés.

exigentes. Como muestra el ejemplo anterior, haber demostrado la corrección funcional de un programa no basta para resolver el problema de *security*. En efecto, aquel programa puede verificar su especificación funcional aun en presencia del desborde de arreglo y en consecuencia ser *safety*-correcto pero claramente no es *security*-correcto.

Más allá de las diferencias fundamentales entre *safety* y *security*, en la práctica la seguridad informática debe abordarse según los lineamientos de la ingeniería. Existe un conjunto de principios, buenas prácticas, métodos, técnicas y herramientas que permiten desplegar sistemas razonablemente seguros. Más allá de la verdad de Perogrullo que establece que la seguridad perfecta o absoluta no existe[3], se pueden alcanzar grados convenientes de seguridad entendiendo que la seguridad informática es un proceso, no un producto [28]. Este principio de Schneider es, en un sentido, la aplicación a la seguridad informática de la inexistencia de balas de plata postulada por Brooks para el desarrollo de software en general. En efecto, Schneider sugiere que, por ejemplo, pensar que la instalación de un anti-virus protegerá nuestras computadoras de virus es muy riesgoso. Además de la instalación de un anti-virus se necesitan otras medidas de seguridad que son parte de un proceso más grande. Por ejemplo, se debe mantener actualizado el anti-virus, debe estar instalado en todas las computadoras, debe estar ejecutando siempre, etc.

Como mencionamos más arriba la definición clásica de seguridad informática incluye los atributos de confidencialidad, integridad y disponibilidad. Abordajes más recientes incluyen también los conceptos de *autenticación*, *autorización* y *auditoría*.

La autenticación de sujetos tiene como paso previo su *identificación*. Un sujeto se identifica ante un sistema cuando provee una *identidad*. El proceso de corroborar que esa identidad es la pretendida se denomina *autenticación*. Para corroborar la identidad el sujeto debe proveer *credenciales de acceso* (es decir, datos u objetos que presenta al sistema con el fin de que este pueda corroborar la identidad).

Ejemplo 3 (Autenticación)**.** *El ejemplo canónico de identificación y autenticación es el mecanismo de login que utilizan los usuarios para ingresar a un sistema. Cuando el usuario provee su 'nombre de usuario' o 'login' se está realizando la identificación; cuando el sistema comprueba que la contraseña provista es la que corresponde a la cuenta de usuario, el usuario ha sido correctamente autenticado.*

Las credenciales de acceso pueden corresponder a uno o más *factores de autenticación*:

- *Algo que se sabe*: algo que sabe únicamente el portador de la identidad pretendida. El ejemplo canónico es una contraseña, pero también puede ser, por ejemplo, un mecanismo de *desafío-respuesta*. Por ejemplo, como método de autenticación el sistema podría presentarle un número al usuario (desafío) y este debería responder con otro que es el resultado de aplicar una operación matemática al primero (respuesta). Notar que en este caso no sería suficiente ver un desafío y su respuesta para descubrir la credencial de acceso e incluso tanto el desafío como la respuesta podrían viajar sin ser encriptados. Aunque el método tiene la desventaja de que el usuario debe tener la capacidad de realizar cálculos (posiblemente) complejos.
- *Algo que se tiene*: es un objeto (físico o virtual) que solo el portador de la identidad pretendida puede tener. El ejemplo canónico es la tarjeta que usamos en los cajeros automáticos. Otro ejemplo son los certificados digitales (el tema se desarrollará en la Unidad 4).

[3]Dado que tales atributos no son aplicables a ninguna creación humana.

- *Algo que se es*: cualquier característica física del usuario. El ejemplo canónico son las huellas dactilares. Otros ejemplos son la retina, ADN, voz, rostro, etc.

Obviamente cuántos más factores de autenticación se utilizan más seguro es el proceso pero también más costoso (no solo computacionalmente sino también económicamente, por ejemplo el costo de imprimir y distribuir tarjetas).

Una vez que la identidad ha sido corroborada se la utiliza en el proceso de *autorización*. Este proceso está muy relacionado con lo que se denomina *control de acceso*. En este contexto, los sujetos acceden objetos por medio de un conjunto de relaciones básicas que se pueden resumir en *leer* (o *read*) y *escribir* (o *write*). A estas relaciones se las suele llamar *permisos*, *modos* o *derechos de acceso*. En el proceso de autorización un sujeto solicita acceso a un objeto en uno de los dos modos básicos, y el sistema se encarga de autorizar o negar dicha solicitud. En este sentido se dice que el sistema *controla el acceso* de los sujetos a los objetos. Algunos autores ven a la autorización como uno de los pasos del control de acceso, al cual le sigue, en caso afirmativo, la entrega del objeto al sujeto en el modo solicitado.

Ejemplo 4 (Autorización en UNIX)**.** *En sistemas tipo UNIX los procesos corren con la identidad del usuario que los lanza (el cual a su vez fue oportunamente autenticado). Por otro lado, cada archivo puede ser leído o escrito por ciertos usuarios o grupos de usuarios. Cuando un proceso solicita acceso en un cierto modo a un archivo, el sistema operativo controla que el usuario a nombre del cual está ejecutando el proceso tenga el acceso solicitado. En caso afirmativo el proceso recibe un descriptor de archivo a través del cual podrá leer o escribir (o ambas) en el archivo, según lo que haya solicitado.*

En la práctica los modos de acceso básicos (lectura y escritura) se extienden a otros modos tales como *añadir*[4], *crear*, *control*, etc.

Ejemplo 5 (Dueños de archivos)**.** *El permiso de* control *permite modificar el modo de acceso en que cada usuario puede acceder a un determinado objeto. En el sistema de archivos típico de UNIX esto está dado por el* propietario *del archivo. Es decir cada archivo tiene un dueño el cual, implícitamente, tiene el permiso de control sobre ese archivo. Es decir el dueño puede determinar qué otros usuarios o grupos de usuarios pueden acceder al archivo y en qué modos.*

El permiso de control es una forma del modo básico de escritura puesto que se lo interpreta como el permiso de escritura sobre los meta-datos del archivo (el i-nodo en sistemas tipo UNIX).

Aunque la forma más común de control de acceso se basa en la identidad del sujeto existen otras que se aplican en cierto contextos más específicos.

Ejemplo 6 (Control de acceso basado en roles)**.** *Una forma de control de acceso bastante difundida entre los gestores de bases de datos relacionales se basa en el concepto de* rol *en lugar de la identidad del sujeto. Por ejemplo, en un sistema de gestión comercial cada usuario puede o no tener el rol de* gerente*. Entonces luego de que el usuario se autentica ante la base de datos, esta le asigna un rol con el cual podrá acceder a ciertas tablas en ciertos modos de acceso. Eventualmente el usuario puede solicitarle al sistema asumir otro rol en cuyo caso el sistema lo autorizará o no.*

Finalmente, el proceso de *auditoría* refiere tanto al registro de los eventos del sistema concernientes a la seguridad, como al análisis y gestión de dicha información. Algunos ejemplos de eventos concernientes a la seguridad son: intento de login fallido, login exitoso, intento de apertura de un archivo, creación de un archivo, establecer una conexión con otra computadora, etc. Conceptualmente todos estos eventos se registran en un repositorio denominado *bitácora*

[4] 'append', en inglés

de auditoría[5]. El contenido de la bitácora puede ser analizado para detectar ataques en progreso o para recolectar evidencia de un ataque anterior que permita, por ejemplo, determinar la identidad del atacante.

Por cuestiones de tiempo, en este curso solo estudiaremos en profundidad el problema del control de acceso o autorización (en la Unidad 2) y someramente el problema de la autenticación (en la Unidad 4).

La seguridad de un sistema de cómputo también puede clasificarse en *seguridad interna* y *seguridad externa*. La seguridad interna es la seguridad del software y de los mecanismos de hardware que sirven a la protección de la información (por ejemplo la separación en anillos de protección provista por los microprocesadores de la familia x86). En tanto que la seguridad externa se ocupa de los controles y actividades de seguridad que el sistema mismo no puede brindar. Entran dentro de la seguridad externa, por ejemplo, la seguridad física de las instalaciones, la seguridad del personal y la seguridad de procedimientos. Si bien en este curso estudiaremos exclusivamente la seguridad interna, la seguridad de un sistema de cómputo empieza por la externa.

Ejemplo 7 (Seguridad interna vs. externa)**.** *Supongamos que instalamos en una PC el sistema operativo más seguro que se conoce (seguridad interna) pero al mismo tiempo dejamos la computadora en un pasillo público de la empresa (seguridad externa). En este caso el resultado neto será que la información allí almacenada y procesada no estará segura puesto que queda expuesta a todo tipo de ataques tales como: robo de la PC misma; robo de su disco rígido; arranque de la PC con un disco externo que permite acceder a todo el contenido del disco interno; instalación por ese mismo medio de programas de ataque; etc.*

El ejemplo anterior evidencia la importancia de la seguridad externa. Por consiguiente al encarar la protección de un sistema de cómputo[6] es conveniente comenzar por determinar la *frontera del sistema* y el *perímetro de seguridad*, como se muestra en la Figura 1. Dentro de la frontera del sistema está todo el hardware y software que debe ser protegido. Estos componentes deben estar regidos por las reglas de la seguridad externa. Dentro del perímetro de seguridad está todo el hardware y software que implementa la *política de seguridad (interna)*. La política de seguridad es, en términos de Ingeniería de Software, el requerimiento de seguridad (recordar el gráfico con los cuatro elementos básicos R, S, D y P visto en clase de Ingeniería de Software I y II).

Como puede apreciarse en la figura los usuarios están fuera de la frontera del sistema. Sin embargo los usuarios poseen información valiosa y eventualmente tienen acceso a procesos críticos de la organización. Como no hay forma de evitar que los usuarios traicionen a la organización (por ejemplo pueden vender la información confidencial que han memorizado), toda la teoría de la seguridad informática se basa sobre la hipótesis de que los *usuarios son confiables o de confianza*. Es decir se asume que los usuarios *autorizados* no dañarán los activos informáticos. Esto significa que un usuario que *no está autorizado* a acceder cierta información es potencialmente un adversario, aunque pertenezca a la organización. O sea que los ingenieros de seguridad informática deben actuar suponiendo que esta persona, a pesar de ser un miembro de la organización, intentará acceder a esa información a la que no está autorizado y eventualmente la divulgará, la corromperá o la tornará indisponible.

Hipótesis de seguridad *La organización confía en que los usuarios no divulgarán, dañarán o harán indisponible la información a la que están autorizados.*

[5] 'audit log' o 'audit trail', en inglés.

[6] Entendido como un conjunto de dispositivos de cómputo interconectados entre sí.

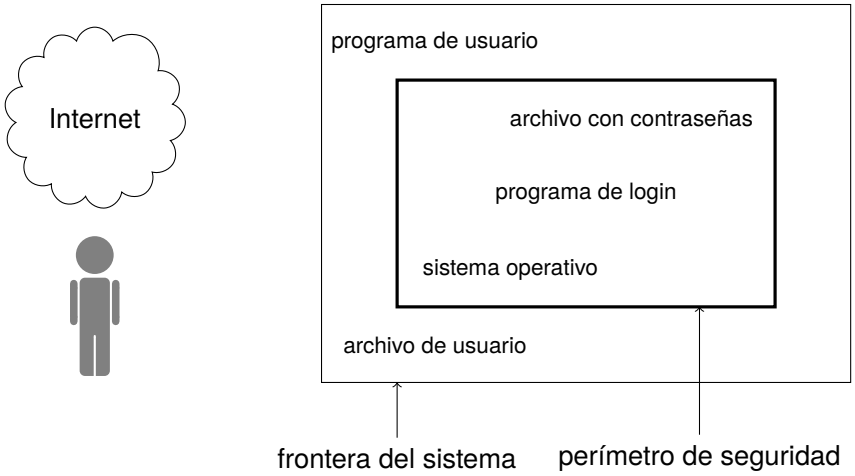


Figura 1: Frontera del sistema y perímetro de seguridad. Se muestran algunos elementos dentro y fuera de estos límites.

Respecto a la seguridad informática el software se clasifica en tres categorías:

- *confiable*: es el software responsable de implementar la política de seguridad. Está dentro del perímetro de seguridad y una falla compromete la seguridad de todo el sistema. Por lo tanto, este software no puede tener errores y debe ser desarrollado por personal de máxima confianza.
- *benigno*: es software que no es responsable de implementar la seguridad pero usa privilegios especiales; sus errores se consideran involuntarios o accidentales.
- *hostil*: se desconoce el origen de este software; es posible que sea usado para atacar el sistema.

Sin embargo, en general, no es posible determinar cuándo un software benigno que contenga errores podrá ser usado para violar la seguridad del sistema. Por este motivo las categorías benigno y hostil se agrupan en la categoría *no confiable*.

La forma más letal de software no confiable son los denominados *caballos de Troya*. Un caballo de Troya es un software que provee cierta funcionalidad útil pero a la vez contiene funciones de ataque.

Ejemplo 8 (Caballo de Troya)**.** *Prácticamente cualquier tipo de software sirve para ejemplificar la idea del caballo de Troya. Un usuario podría encontrar en Internet una planilla de cálculo distribuida gratuitamente con funciones sofisticadas para reconocimiento de patrones en grandes cantidades de datos. Esta persona instalaría el software en su computadora, abriría sus archivos con los datos en cuestión y posiblemente obtendría resultados razonables. Sin embargo, al mismo tiempo, el software podría estar enviando por la red los archivos con los datos. Si esos datos debían permanecer confidenciales, por ejemplo datos provenientes de estudios clínicos sobre uso de diferentes drogas, el administrador del sistema no le debería haber permitido al usuario utilizar software no confiable.*

Por otro lado, si en lugar de descargar el software de Internet lo compramos a una empresa privada, ¿deja de ser un caballo de Troya? Si los datos de esos estudios clínicos valen millones de dólares, ¿no cabe la posibilidad de que estafadores, la competencia o un gobierno monten un ataque de este tipo?

No es decidible determinar si un programa es o no un caballo de Troya. Por lo tanto se debería impedir la instalación de software no confiable. Sin embargo, ¿es Ubuntu confiable? ¿Lo es Windows? ¿Firefox? Ciertamente en ciertos contextos puede ser razonable asumir que estos programas no son caballos de Troya. Por ejemplo, usuarios individuales que no tienen información crítica pueden utilizar estos programas sin riesgo. No obstante, para ciertos usuarios institucionales estas premisas podrían ser demasiado riesgosas. Por ejemplo, ¿es razonable que el gobierno de la República Popular China utilice para procesar información crítica versiones de Windows? ¿Y de Linux? [12, 19, 31] Pero variando un poco el contexto, incluso usuarios individuales podrían verse seriamente afectados por software aparentemente confiable. Por ejemplo, si Firefox llegase a ser un caballo de Troya sería muy riesgoso para una persona utilizar el servicio de banca electrónica ofrecido por su banco.

Como veremos en detalle en la Unidad 2 en la actualidad ningún sistema operativo de uso masivo (Linux, Windows, Mac OS, Android, etc.) puede detener, en general, ataques a la confidencialidad que utilicen caballos de Troya. Notar que si la confidencialidad del sistema resulta comprometida es relativamente fácil comprometer también la integridad y la disponibilidad. En efecto, al comprometer la confidencialidad se puede tener acceso a las credenciales de acceso al sistema de los usuarios privilegiados (e.g. contraseñas) lo que permitirá a los atacantes acceder al sistema y ejecutar los procesos que aquellos usuarios están autorizados. De esta forma los atacantes podrían modificar (y en particular borrar) información crítica, apagar servidores, cambiar configuraciones de red, etc. En consecuencia la protección contra ataques a la confidencialidad por medio de caballos de Troya está en la base del problema de la seguridad informática.

Esta unidad debe completarse leyendo [14, capítulos 1–4]

2. Unidad II: Confidencialidad en sistemas de cómputo

¿Cómo preservar un secreto almacenado en un sistema de cómputo? ¿Cómo evitar el espionaje informático? ¿Es tan difícil mantener un secreto dentro de una computadora? ¿Por qué no serían suficientes los mecanismos de protección de los sistemas operativos comerciales o de uso masivo?

Efectivamente esos mecanismos de protección no son suficientes debido a los ataques por medio de caballos de Troya. Este tipo de ataque consiste, esencialmente, de los siguientes pasos:

1. El atacante desarrolla un caballo de Troya (T)
2. Logra instalarlo en el sistema que quiere atacar
3. Un usuario (U) con acceso al secreto que busca el atacante ejecuta T
4. T lee el secreto y lo escribe en un objeto al cual el atacante tiene acceso (por ejemplo otro archivo o un *socket*)

En parte este ataque es posible porque T hereda los permisos del usuario U. Es decir, para el sistema operativo, T es un proceso como cualquier otro y por lo tanto tiene acceso a los mismos objetos que el usuario U. A su vez esta situación es posible porque los sistemas operativos de uso masivo implementan una forma de control de acceso denominada *control de acceso discrecional* (DAC[7]). Este nombre proviene del hecho de que parte de la política de seguridad puede ser

[7] 'discretionary access control', en inglés.

	s_1	s_2	s_3	s_4	s_5	s_6
o_2		r,w,c	r		r,w,c	r,w
o_1	r,w,c		r,c			

Figura 2: Parte de una matriz de control de acceso. r permiso de lectura, w de escritura y c de control. Las celdas vacías indican que el sujeto no tiene ningún permiso sobre el objeto.

fijada *discrecionalmente* por los usuarios ordinarios. En sistemas DAC los usuarios y procesos ordinarios pueden determinar y modificar los permisos de algunos objetos y pueden establecer los permisos de algunos de los objetos que ellos crean (en general de los archivos, directorios y otros objetos del sistema de archivos).

Conceptualmente, en DAC existe una *matriz de control de acceso* en cuyas filas están los objetos del sistema y en cuyas columnas están los sujetos. En cada componente de la matriz se listan los permisos que el sujeto tiene sobre el objeto (ver Figura 2). Los usuarios ordinarios con permiso de control pueden cambiar los derechos de acceso sobre el objeto por medio de operaciones provistas por el sistema operativo (en sistemas tipo UNIX son los comandos `chmod`, `chown`, etc. que utilizan las llamadas al sistema correspondientes).

Sin embargo, en la práctica la mayoría de los sistemas operativos implementan alguna forma de *lista de control de acceso* (ACL[8]). En este caso cada objeto tiene asociada una ACL donde se listan los permisos de los sujetos que tienen alguno. Por ejemplo, respecto de la Figura 2, la lista de control de acceso del objeto o_1 sería $[(s_1, \{r,w,c\}), (s_3, \{r,c\})]$. Es decir una ACL es equivalente a una fila de la matriz de acceso sin las celdas vacías.

En los párrafos anteriores hemos hecho referencia siempre a la seguridad de sistemas operativos. De alguna forma descartamos de plano la posibilidad de implementar la seguridad de un sistema de cómputo en algún otro estrato. En particular las teorías de sistemas operativos y seguridad informática no consideran apropiado ni seguro intentar implementar controles de seguridad a nivel de aplicación. De todas formas se acepta como razonable que ciertas aplicaciones, en general muy sofisticadas y que manejan sus propios datos e incluso sus propios sistemas de archivos, tales como los gestores de bases de datos relacionales (DBMS), implementen sus propios controles de seguridad. Por ejemplo, los DBMS suelen tener ACL que se pueden asociar a bases de datos, tablas, filas, columnas e incluso celdas (en parte porque estos objetos no son visibles a nivel del sistema operativo). En general, todo lo que mencionemos en esta unidad asume que estamos hablando de sistemas operativos pero también se puede aplicar a DBMS.

2.1. Seguridad multi-nivel

Para comenzar a aproximarnos a las soluciones al problema de la confidencialidad debemos entender primero cuál es la política de seguridad para este atributo. La política de seguridad

[8] 'access control list', en inglés.

informática que regula la confidencialidad se denomina *seguridad multi-nivel* (MLS[9]). Esta es la política de seguridad aplicada por el Departamento de Defensa (DoD) de los EE.UU., incluso antes de la aparición de las computadoras, a toda su información crítica o sensible, en particular aquella vinculada con los datos de inteligencia y militares.

Según la política de MLS:

- Cada documento y cada persona posee (o tiene asignada) una *clase de acceso*.
- Cada clase de acceso es un par ordenado de la forma (n, C) donde n se denomina *nivel de seguridad* y C es un conjunto de *categorías* (denominado *departamento*[10]).
- El nivel de seguridad es un elemento de un conjunto finito totalmente ordenado. El DoD utiliza el siguiente conjunto de etiquetas: *unclassified* < *confidential* < *secret* < *top secret*. Cuando esta política de seguridad se formaliza con algún lenguaje matemático se suele generalizar de forma tal que el nivel de seguridad es un número natural.
- Cada categoría presente en la segunda componente de una clase de acceso pertenece a un cierto conjunto (de categorías) que podemos notar con $\mathbb{C}$ (es decir si C es la segunda componente de una clase de acceso entonces $C \subseteq \mathbb{C}$). Cada categoría refiere a un tema o asunto del cual trata el documento o sobre el cual la persona tiene un interés legítimo. Por ejemplo, en el ambiente militar de EE.UU. algunas categorías podrían ser: Iraq, Al-Qaeda, armas nucleares, misiles de largo alcance, etc. En la terminología del DoD a las categorías se las suele llamar *need-to-know*, es decir que representan temas o asuntos que la persona tiene la necesidad de saber o conocer. Usualmente $\mathbb{C}$ es un conjunto de mayor tamaño que el conjunto de niveles de seguridad.
- Una persona puede *leer* un documento sí y solo sí la clase de acceso de la persona *domina* a la clase de acceso del documento. La clase de acceso (n_1, C_1) domina a la clase de acceso (n_2, C_2) sí y solo sí $n_2 \leq n_1$ y $C_2 \subseteq C_1$. Es decir la relación *domina* define un orden parcial en el conjunto de clases de acceso.

Observar que la regla principal de la política MLS (la última) hace referencia únicamente al permiso de lectura. Esto es así debido a que la confidencialidad está relacionada con la posibilidad de ver o leer información. Es decir, un dato se mantiene confidencial en tanto y en cuanto nadie no autorizado logre leerlo. En ese caso ese dato permanece secreto en el sentido de que solo lo saben o conocen aquellos que lo comparten. Esta regla de MLS se conoce, en inglés, como *no read-up*. Aun así, cuando esta política debe ser implementada en un sistema operativo se deben determinar reglas que regulen la escritura de objetos y la creación de nuevos objetos, como veremos un poco más adelante.

En MLS una vez que se asigna una clase de acceso a un documento o a una persona, solo se puede modificar a través de un procedimiento administrativo complejo y altamente controlado. Por ejemplo, para aumentar la clase de acceso de un oficial de inteligencia este será sometido a interrogatorios, averiguación de antecedentes, etc. con el fin de disminuir al mínimo las posibilidades de que traicione a la organización (recordar la hipótesis de seguridad vista en la Unidad 1). En el mismo sentido los documentos deben pasar por una serie de controles antes de ser *desclasificados*. En particular debe pasar un cierto número de años (en ocasiones más de 50) para que ciertos documentos puedan ser vistos por el público en general.

[9] 'multi-level security', en inglés.
[10] 'compartment', en inglés.

La asignación de una clase de acceso a un documento no queda a *discreción* de quien lo genera o ingresa al sistema. En este sentido la política MLS entra dentro de la categoría de *control de acceso obligatorio* (MAC[11]). Es decir, los usuarios ordinarios no pueden determinar ni modificar la política de seguridad.

2.2. Arquitectura de software para seguridad

Uno de los problemas que surgen al implementar sistemas de alta seguridad es definir una arquitectura que minimice el código necesario para implementar la seguridad del sistema. Un concepto que ayuda a abordar este problema es la *TCB*.

Definición 3 (TCB)**.** *La* base de cómputo confiable *(TCB[12]) es el conjunto de componentes de hardware y software responsables de implementar la política de seguridad.*

La TCB debe ser desarrollada por personal de máxima confianza puesto que cualquier error o código hostil añadido a ella pondrá en riesgo la seguridad de todo el sistema. Por consiguiente el código fuera de la TCB puede ser desarrollado incluso por un enemigo de máxima peligrosidad; la seguridad del sistema depende únicamente de la TCB. Se espera que la TCB sea lo más pequeña y simple posible de forma tal que sea posible verificarla formalmente. Pero a la vez se espera que ninguna operación que pueda poner en riesgo la seguridad sea ejecutada sin ser controlada por la TCB. Es decir la TCB debe imponer el principio de *mediación total*[13] el cual estipula que todos los accesos de sujetos a objetos deben ser *mediados* por la TCB. Una forma probada de implementar los componentes de software de la TCB es agrupándolos en *aplicaciones confiables*[14] y *núcleo de seguridad*[15]. El núcleo de seguridad es aquella parte del sistema operativo responsable de implementar la seguridad. La teoría de sistemas operativos indica que los componentes responsables de la seguridad representan una parte relativamente pequeña del sistema operativo. Las aplicaciones confiables son componentes de la TCB (típicamente login, cambio de contraseña, encriptación, etc.) que por cuestiones de arquitectura no se pueden implementar a nivel del sistema operativo. De esta forma, como muestra la Figura 3, el núcleo de seguridad ejecuta en un anillo de protección más interior que el resto del sistema operativo.

Un elemento interesante de la TCB es el *camino confiable*[16]. Un camino confiable es un mecanismo que garantiza al usuario que la próxima interacción con el sistema será con un componente de la TCB, típicamente una de las aplicaciones confiables. De esta forma el usuario estará seguro que no está interactuando con un caballo de Troya.

Ejemplo 9 (Camino confiable)**.** *El ejemplo canónico de uso de un camino confiable es el proceso de login. Supongamos que la función de ataque de un caballo de Troya es presentar una pantalla de login exactamente igual a la legítima luego de que pase un cierto tiempo de inactividad de la estación de trabajo. De esta forma el caballo de Troya podrá capturar la contraseña del usuario y transmitirla al atacante.*

Si el sistema implementa alguna forma de camino confiable este ataque se puede evitar de la siguiente manera. El camino confiable se activa cuando el usuario pulsa una cierta combinación de teclas (digamos Ctrl+Alt+Del). Esta combinación de teclas es capturada por el núcleo de seguridad (hardware) el cual

[11] 'mandatory access control', en inglés.
[12] 'trusted computer base', en inglés.
[13] 'complete mediation', en inglés.
[14] 'trusted functions', en inglés.
[15] 'security kernel', en inglés.
[16] 'trusted path', en inglés.

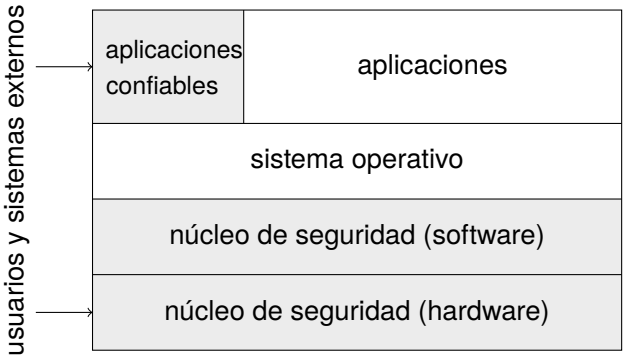


Figura 3: Diagrama de bloques que representa los componentes de un sistema de cómputo confiable. Los componentes en gris forman la TCB.

la pasa al componente de software del mismo núcleo. Este componente ejecuta entonces la aplicación de login (que es una de las aplicaciones confiables), la cual solicita la contraseña.

Aun así, si el usuario no es precavido podría dar su contraseña a cualquier programa que se la pida aunque no tenga la apariencia de un programa confiable. Incluso podría usar la misma contraseña para acceder a servidores críticos y servidores no críticos. En definitiva esto nos lleva a una conclusión importante: la seguridad es una cadena que se corta por el eslabón más débil. Y muchas veces ese eslabón es el usuario. Ningún mecanismo de seguridad, por más sofisticado que sea, funcionará sin la colaboración activa de sus usuarios.

En este caso particular los usuarios deberían estar capacitados de forma tal de no ingresar su contraseña sin antes activar el camino confiable. Observar que sin el mecanismo de camino confiable los usuarios no tienen una forma segura de ingresar su contraseña.

Esta sección debe completarse leyendo [14, capítulos 5–7]

2.3. Modelos de confidencialidad

Ya hemos mencionado que la política de seguridad corresponde a los requerimientos (con respecto a la Ingeniería de Software). En este sentido el *modelo de seguridad* corresponde a la especificación. En seguridad informática existe una larga tradición de aplicar métodos formales para la especificación y verificación de los componentes responsables de implementar la política de seguridad. Por lo tanto, casi invariablemente, cuando en seguridad informática se habla de modelo de seguridad, se hace referencia a un modelo formal.

Entonces, un modelo de seguridad es la especificación formal del comportamiento (o función) de la interfaz de los componentes de software responsables de implementar la seguridad de un sistema de cómputo (es decir de los componentes de software de la TCB).

Especificar un modelo de seguridad para la política de MLS ha resultado ser más complejo de lo esperado. Lo que es simple fuera de las computadoras se torna muy complejo dentro de ellas si se consideran los ataques por medio de caballos de Troya. En las secciones que siguen veremos tres modelos de MLS en el orden cronológico en que fueron publicados:

1. modelo de Bell-LaPadula (BLP)

2. modelo de no interferencia de Goguen-Meseguer
3. modelo de multi-ejecución segura (SME[17])

2.4. Modelo de Bell-LaPadula

El primer modelo de MLS fue propuesto en 1972 por dos investigadores norteamericanos de Mitre Corp., Elliot Bell y Leonard LaPadula [2, 3]. El modelo BLP es la especificación de las llamadas al sistema de un sistema operativo antecesor a UNIX. Bell y LaPadula utilizaron una notación matemática simple pero ad-hoc para describir una máquina de estados cuyo espacio de estados está constituido por las estructuras de datos necesarias para imponer MLS y cuyas transiciones son las llamadas al sistema operativo. Además de MLS el modelo BLP incluye la especificación de un control DAC. Las operaciones del sistema solo se pueden ejecutar si se pasan los controles DAC y MLS. Luego especificaron dos invariantes de estado que describen de manera concisa las propiedades claves para un sistema operativo que implemente MLS. Finalmente demostraron que la especificación de las transiciones preserva esos invariantes.

En lugar de utilizar la notación que usaron Bell y LaPadula nosotros utilizaremos Z. Además no vamos a especificar todas las llamadas al sistema que ellos especificaron sino solo dos que son muy representativas de las dificultades que se deben enfrentar cuando se trabaja con MLS. El control DAC tampoco lo especificaremos.

Comenzamos introduciendo los tipos básicos de nuestra especificación Z que son las categorías (CAT), niveles de seguridad (SL) y objetos (OBJ).

$$[CAT, OBJ]$$

$$SL == \mathbb{N}$$

Los sujetos son un subconjunto no vacío de los objetos.

$$\begin{array}{|l} SUBJ : \mathbb{P}\, OBJ \\ \hline SUBJ \neq \emptyset \end{array}$$

Las clases de acceso son pares ordenados de niveles de seguridad y conjuntos de categorías.

$$SC == SL \times \mathbb{P}\, CAT$$

También podemos definir la relación de orden parcial entre clases de acceso.

$$\begin{array}{|l} _dominates_ : SC \leftrightarrow SC \\ \hline \forall a, b : SC \bullet a \ dominates \ b \Leftrightarrow b.1 \leq a.1 \wedge b.2 \subseteq a.2 \end{array}$$

Aunque BLP usa varios modos de acceso nosotros solo utilizaremos los dos modos de acceso básicos.

$$AM ::= rm \mid wm$$

El estado del sistema se compone de:

[17] 'secure multi-execution', in inglés.

- una función que asocia objetos con clases de acceso, *sc*
- una función que asocia sujetos con los objetos que están accediendo y en qué modo, *oobj* (por 'Open OBJects')

SecState

$sc : OBJ \nrightarrow SC$
$oobj : SUBJ \nrightarrow (OBJ \leftrightarrow AM)$

Si pensamos en un sistema operativo tipo UNIX, los sujetos serían procesos, los objetos archivos, la variable *sc* los permisos guardados en los i-nodos, y la variable *oobj* los archivos que cada proceso tiene abiertos. El estado inicial es trivial.

ISecState

SecState

$sc = \emptyset$
$oobj = \emptyset$

Antes de formalizar los invariantes de estado que dieron originalmente Bell y LaPadula vamos a dar invariantes simples que complementan a los tipos que hemos definido.

InvType

SecState

$\mathrm{dom}\, oobj \subseteq SUBJ$
$\mathrm{dom}\, oobj \subseteq \mathrm{dom}\, sc$
$\forall x : \mathrm{ran}\, oobj \bullet \mathrm{dom}\, x \subseteq \mathrm{dom}\, sc$

Es decir, estos invariantes exigen que los sujetos que tienen abiertos objetos y esos mismos objetos sean sujetos y objetos del sistema.

La primera propiedad enunciada por Bell y LaPadula (en forma de invariante de estados) la llamaron *condición de seguridad*[18]. La condición de seguridad es la formalización directa de la regla de acceso a la información estipulada en MLS: una persona puede leer un documento sí y solo sí la clase de acceso de la persona domina a la clase de acceso del documento. Traducido al estado del sistema que hemos definido significa que si un sujeto tiene acceso de lectura a un objeto (i.e. $(s, o, rm) \in oobj$) es porque la clase de acceso de ese sujeto domina a la del objeto (i.e. $sc(s)$ *dominates* $sc(o)$).

InvSecCond

SecState

$\forall s : \mathrm{dom}\, oobj;\ o : OBJ \mid (s, o, rm) \in oobj \bullet sc(s)\ dominates\ sc(o)$

La segunda propiedad exigida en BLP se llama *propiedad estrella* y se suele escribir *propiedad-**[19]. Esta propiedad no está enunciada en MLS y aparece en BLP pura y exclusivamente porque

[18] 'security condition', en inglés.
[19] '*-property', en el original en inglés.

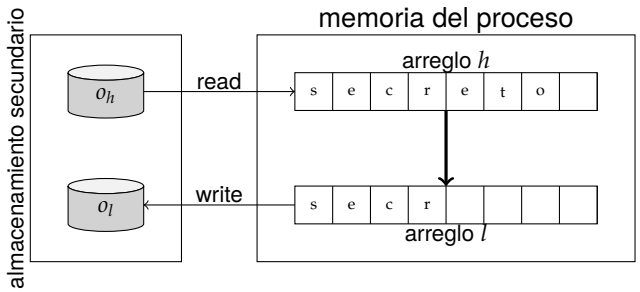


Figura 4: La desclasificación no autorizada de información es posible sin la propiedad-*.

se trata de un sistema de cómputo donde existe la posibilidad de que caballos de Troya *escriban* datos. En efecto, si un sujeto tiene abiertos los objetos o_h y o_l, respectivamente, en modo de lectura y escritura[20], y son tales que $sc(o_h)$ *dominates* $sc(o_l)$, entonces le sería posible copiar los datos de o_h en o_l produciendo la desclasificación[21] no autorizada de la información contenida en o_h. Esto es posible porque un sistema operativo puede controlar a los procesos solo cuando efectúan llamadas al sistema cosa que no es necesaria cuando copian información entre áreas de memoria propias. Esta situación se representa gráficamente en la Figura 4. Allí se puede observar que la lectura del archivo o_h se efectúa por medio de la llamada al sistema read en tanto que la escritura en el archivo o_l se hace usando la llamada write; en ambos casos el sistema operativo (o más precisamente la TCB) puede controlar que todo esté de acuerdo a la política de seguridad. Sin embargo, la copia de los datos del arreglo h en el arreglo l (simbolizada por la línea gruesa) escapa al control de la TBC. Por lo tanto esta no tiene forma de saber que el contenido de l es de clase de acceso $sc(o_h)$.

Entonces, la propiedad-* exige que, para cada sujeto, todos los objetos abiertos en modo de lectura tengan una clase de acceso dominada por la de cualquiera de los objetos abiertos en modo de escritura. En inglés esto se denomina *no write-down*; o sea que tenemos *no read-up* y *no write-down*.

InvStarProp

SecState

$$\forall s : \mathrm{dom}\, oobj;\ o_1, o_2 : OBJ \mid \{(s, o_1, wm), (s, o_2, rm)\} \subseteq oobj \bullet sc(o_1)\ dominates\ sc(o_2)$$

A continuación especificaremos las dos operaciones claves del sistema: solicitar acceso a un objeto en modo de lectura y en modo de escritura (es decir, las operaciones que representan la

[20] Usualmente la letra 'h' se utiliza para denotar objetos con clase de acceso alta (por *high*, en inglés), mientras que 'l' se usa para objetos de clase de acceso baja (por *low*, en inglés). Luego del trabajo de Bell y LaPadula la mayoría de los investigadores del área comenzaron a proponer soluciones al problema de la confidencialidad donde las clases de acceso se simplifican considerando solo el nivel de seguridad, la relación de dominación se vuelve un orden total (<) y donde solo se consideran dos niveles, $L < H$, donde L es el nivel de la información *pública* y H es el de la información *secreta*. En este caso se habla más de niveles de seguridad que de clases de acceso. La generalización de estas soluciones a un retículo de clases de acceso en general es trivial. De aquí en más consideraremos esta simplificación del problema cuando sea conveniente.

[21] 'downgrading', en inglés.

llamada al sistema UNIX `open`). Comenzamos por la apertura en modo *rm*. Observar que en la llamada al sistema el proceso que solicita la apertura es un parámetro de entrada implícito puesto que el sistema operativo está en condiciones de saber cuál es el proceso que hace la solicitud.

OpenReadOk

$\Delta SecState$
$s?, o? : OBJ$

$s? \in SUBJ$
$\{s?, o?\} \subseteq \mathrm{dom}\, sc$
$o? \notin \mathrm{dom}(oobj(s?) \rhd \{rm\})$
$sc(s?)\ dominates\ sc(o?)$
$\forall o : \mathrm{dom}(oobj(s?) \rhd \{wm\}) \bullet sc(o)\ dominates\ sc(o?)$
$oobj' = oobj \cup \{(s?, o?, rm)\}$
$sc' = sc$

Es decir:

- $s?$ debe ser un sujeto
- $s?$ y $o?$ deben ser objetos existentes en el sistema
- $s?$ no tiene abierto a $o?$ en modo de lectura
- La clase de acceso de $s?$ domina a la de $o?$. Este predicado implementa la regla *no read-up* de MLS.
- La clase de acceso de $o?$ está dominada por la clase de acceso de todos los objetos que $s?$ tiene abiertos en modo de escritura. Este predicado apunta a controlar posibles *write-downs*.
- En ese caso $o?$ pasa a ser uno de los objetos que $s?$ tiene abiertos en modo de lectura

No especificaremos los casos de error aunque Bell y LaPadula lo hacen en el trabajo original. Asumimos que la especificación final es el esquema *OpenRead* que reúne la disyunción de *OpenReadOk* y los esquemas de error.

La siguiente operación es solicitar acceso en modo de escritura a un objeto. Si bien desde el punto de vista de MLS un sujeto podría escribir en objetos más altos, en BLP solo se permite la escritura en objetos al mismo nivel que el sujeto.

OpenWriteOk

$\Delta SecState$
$s?, o? : OBJ$

$s? \in SUBJ$
$\{s?, o?\} \subseteq \mathrm{dom}\, sc$
$o? \notin \mathrm{dom}(oobj(s?) \rhd \{wm\})$
$sc(s?) = sc(o?)$
$\forall o : \mathrm{dom}(oobj(s?) \rhd \{rm\}) \bullet sc(o?)\ dominates\ sc(o)$
$oobj' = oobj \cup \{(s?, o?, wm)\}$
$sc' = sc$

Luego de especificar las demás operaciones del sistema, Bell y LaPadula demuestran que todas ellas preservan la condición de seguridad y la propiedad-*. Básicamente escriben teoremas como los que siguen:

theorem OpenReadSecCond
$InvSecCond \wedge OpenRead \Rightarrow InvSecCond'$

theorem OpenReadStarProp
$InvStarProp \wedge OpenRead \Rightarrow InvStarProp'$

theorem OpenWriteSecCond
$InvSecCond \wedge OpenWrite \Rightarrow InvSecCond'$

theorem OpenWriteStarProp
$InvStarProp \wedge OpenWrite \Rightarrow InvStarProp'$

a los cuales nosotros debemos agregar:

theorem OpenReadInvType
$InvType \wedge OpenRead \Rightarrow InvType'$

theorem OpenWriteInvType
$InvType \wedge OpenWrite \Rightarrow InvType'$

todos los cuales pueden demostrarse formalmente con una herramienta como Z/EVES.

Dado que todas las operaciones preservan *InvSecCond* e *InvStarProp*, si el sistema parte de un estado en donde esos invariantes se verifican entonces el sistema será siempre seguro.

Es posible verificar automáticamente que las operaciones de BLP verifican las dos propiedades claves del modelo usando {*log*} [9].

2.4.1. Los problemas de BLP

El mayor problema con BLP es que es innecesariamente restrictivo. BLP no solo prohíbe el escenario descripto en la Figura 4 sino muchas ejecuciones seguras. Es decir, BLP prohíbe que se abran archivos que violan la propiedad-* aun cuando esos programas no necesariamente desclasificarán información de forma ilegítima. Esto lleva a que los sistemas basados en BLP sean menos usables de lo que MLS permite.

Ejemplo 10 (BLP no es usable)**.** *Supongamos que disponemos de una prueba formal de que un editor de textos no desclasifica información de forma ilegítima. Esto significa que este editor podría abrir los archivos f_h y f_l en modo de edición (es decir lectura y escritura simultáneamente), donde $sc(f_h)$ dominates $sc(f_l)$. De esta forma el usuario podría leer y escribir en ambos archivos estando seguro de que la información secreta no sería divulgada por el editor. Sin embargo, si este editor corriera sobre un sistema operativo que implementa BLP el sistema prohibiría el escenario anterior haciendo que el editor sea menos usable para el usuario de lo que en realidad podría ser.*

Otro problema con BLP, que se deriva del anterior, se denomina *escalda de nivel*[22]. Debido a la propiedad-*, aun cuando un sujeto haya cerrado un objeto abierto en modo de lectura, no podrá abrir objetos en modo de escritura cuya clase de acceso no domine a la del objeto que acaba de ser cerrado. Esto implica que el sujeto (aunque en este caso sería más apropiado hablar de proceso) queda clasificado con el supremo de las clases de acceso de todos los archivos que haya abierto en modo de lectura durante toda su vida. Obviamente este supremo tiende a ser cada vez mayor. Para frenar esta escalada de nivel habría que destruir al sujeto para iniciarlo nuevamente (en cuyo caso comienza con el ínfimo del orden parcial). Nuevamente esto va en detrimento de la usabilidad de los sistemas basados en BLP[23].

Finalmente, otro de los problemas señalados por varios autores es que es muy difícil determinar todos los objetos del sistema. Si un cierto contenedor de información no es clasificado como objeto del sistema entonces no será considerado a la hora de controlar la validez de las dos propiedades del modelo y en consecuencia un atacante podría aprovecharse de este tipo de contenedores.

Ejemplo 11 (Meta-datos como objetos)**.** *Un caso típico, aunque bastante evidente como para que no sea detectado por un ingeniero de seguridad, son los diversos meta-datos de los archivos (nombre, fechas de acceso, etc.). Es decir, por ejemplo, si la la fecha de acceso no es considerada un objeto del sistema (y por lo tanto no está clasificada con una clase de acceso) un atacante podría utilizarla para comunicar información. Volveremos sobre esta idea en la siguiente sección cuando abordemos el concepto de* canal encubierto.

2.5. No inteferencia

El concepto de *no iterferencia* fue propuesto originalmente por Joseph Goguen y José Meseguer en 1982 [15]. Aun hoy es considerado 'la' definición del problema de la confidencialidad en sistemas de cómputo.

Dados dos grupos de usuarios, G_h y G_l (recordar el significado de h y l), informalmente, se dice que G_h no interfiere con G_l si las acciones de los usuarios de G_h no pueden ser percibidas por los miembros del grupo G_l. En este contexto 'percibir' se entiende como la posibilidad de *ver* una salida producida por el sistema. Entonces, el grupo G_h no interfiere con el grupo G_l si la salida del sistema que pueden ver estos últimos no cambia con lo que hagan los usuarios de G_h.

Goguen y Meseguer dan la definición de no interferencia a través de lo que ellos llaman *sistemas de capacidades*. Un sistema de capacidades es un sistema donde lo que está permitido hacer varía con el tiempo. Formalmente un sistema de capacidades M consta de los siguientes elementos:

- un conjunto de usuarios, U
- un conjunto de estados, S
- un conjunto de comandos o transiciones de estado, T
- un conjunto de salidas del sistema, Out

  Las salidas incluyen pantalla, impresora, disco, red, etc.; es decir todo aquello que el usuario pueda sensar, percibir o ver del sistema.

[22] 'level creep', en inglés.

[23] Si bien es siempre cierto que a mayores exigencias de seguridad menores niveles de usabilidad, en el caso de BLP la reducción del nivel de seguridad es exagerada puesto que, como vimos, prohíbe comportamientos seguros.

- un conjunto de tablas de capacidades, C.

  Una tabla de capacidades es conceptualmente equivalente a una matriz de control de acceso. En sistemas donde los permisos no cambian a medida que el sistema ejecuta, este elemento puede no considerarse.
- un conjunto de comandos de capacidades, P

  Estos comandos toman una tabla de capacidades y retornan una nueva tabla de capacidades; es decir modifican los permisos del sistema.
- la función $\mathcal{O} : S \times C \times U \to Out$, que indica lo que un usuario dado puede ver cuando el sistema está en cada estado y está vigente una cierta tabla de capacidades. $\mathcal{O}$ se llama función de salida.
- la función $\mathcal{T} : S \times C \times U \times T \to S$, que indica cómo se modifican los estados por medio de transiciones de estado.
- la función $\mathcal{P} : C \times U \times P \to C$, que indica cómo se actualizan las tablas de capacidades.
- las constantes s_0 y c_0 que representan el estado y la tabla de capacidades iniciales.

En lo que sigue A denota la unión de los conjuntos de comandos de estado y comandos de capacidades, es decir $A = T \cup P$. También definimos $\mathcal{E}$ como la función que ejecuta tanto una transición de estados como un cambio en la tabla de capacidades:

$$\begin{aligned}
&\mathcal{E} : S \times C \times U \times A \to S \times C \\
&\quad (s,c,u,a) \to \mathcal{E}(s,c,u,a) = \begin{cases} (\mathcal{T}(s,c,u,a),c) & \text{si } a \in T \\ (s,\mathcal{P}(c,u,a)) & \text{si } a \in P \end{cases}
\end{aligned}$$

De esta forma un sistema de capacidades es una máquina de estados con $S \times C$ como espacio de estados, $U \times A$ como espacio de entradas y espacio de salidas Out. Ahora extendemos $\mathcal{E}$ a secuencias de entradas siguiendo la forma habitual:

$$\begin{aligned}
&\mathcal{E} : S \times C \times (U \times A)^* \to S \times C \\
&\quad (s,c,w) \to \mathcal{E}(s,c,w) = \begin{cases} (s,c) & \text{si } w = \langle\rangle \\ \mathcal{E}(\mathcal{E}(s,c,v),u,a) & \text{si } w = v \frown \langle (u,a) \rangle \end{cases}
\end{aligned}$$

donde $\mathcal{E}$ se usa como nombre para ambas funciones dado que los tipos son distintos.

Antes de poder formalizar el concepto de no interferencia necesitamos un poco de notación. Notamos con $[\![w]\!]$ a $\mathcal{E}(s_0,c_0,w)$ para cualquier secuencia $w \in (U \times A)^*$; y con $[\![w]\!]_u$ a $\mathcal{O}([\![w]\!],u)$ para cualquier usuario $u \in U$. Es decir $[\![w]\!]_u$ es todo lo que el usuario u puede ver como salida del sistema luego de que este ejecute la secuencia de comandos w desde la configuración inicial. Sea G un subconjunto de U, B un subconjunto de A y $w \in (U \times A)^*$. Entonces $\tilde{w}_G$ es la secuencia que surge de eliminar de w todos los pares cuya primera componente es un elemento de G; $\tilde{w}^B$ es la secuencia que surge de eliminar de w todos los pares cuya segunda componente es un elemento de B; y $\tilde{w}^B_G = (\tilde{w}_G)^B$.

Definición 4 (No interferencia)**.** *Dado un sistema de capacidades M y grupos de usuarios G_h y G_l, decimos que G_h no interfiere con G_l, simbolizado $G_h :| G_l$, sí y solo sí:*

$$\forall w \in (U \times A)^*, u \in G_l : [\![w]\!]_u = [\![\tilde{w}_{G_h}]\!]_u$$

O sea que la definición establece que lo que el usuario u puede ver como salida del sistema M es lo mismo tanto si los usuarios de G_h ejecutan comandos en M como si no lo hacen. En otras palabras, nada de lo que hacen con M los usuarios de G_h *interfiere* con lo que *ven* los usuarios de G_l. Notar que la definición es una cuantificación universal sobre *todas* las posibles ejecuciones de M.

De forma similar, dados $B \subseteq A$ y $G, G_1 \subseteq U$, definimos:

- $B :| G \Leftrightarrow \forall w \in (U \times A)^*, u \in G : [\![w]\!]_u = [\![\tilde{w}^B]\!]_u$
- $B, G :| G_1 \Leftrightarrow \forall w \in (U \times A)^*, u \in G_1 : [\![w]\!]_u = [\![\tilde{w}^B_G]\!]_u$

Ejemplo 12 (Crear un archivo)**.** *Si $B \subseteq A$ entonces $B :| \{u\}$ significa que los comandos de B no tienen efecto sobre lo que u puede ver como salida del sistema. Entonces si* create *(crear un archivo) pertenece a B, la salida que ve u a través de* ls *(listar archivos) antes y después de ejecutar* create *debe ser la misma. Por ejemplo, si en determinado momento u ejecuta* ls *y ve:*

apunte.tex
apunte.pdf

pero luego alguien ejecuta create(examen.tex) *y u vuelve a ejecutar* ls *debería volver a ver:*

apunte.tex
apunte.pdf

Es decir que el sistema debería recordar el listado que le mostró a u la primera vez y debería mostrarle siempre el mismo aunque cualquier usuario cree nuevos archivos. Si esto hay que hacerlo para todos los directorios y usuarios, el sistema se puede volver muy ineficiente.

Ejemplo 13 (Seguridad multi-nivel)**.** *Un sistema de capacidades M es MLS sí y solo sí para toda clase de acceso vale:*

$$U_{+x} :| \overline{U_{+x}}$$

donde $U_{+x} = \{u \in U \mid sc(u)$ dominates $x\}$ y $\bar{X}$ es el complemento del conjunto X.

La definición de MLS en términos de no interferencia dada en el ejemplo anterior se puede simplificar si primero definimos el concepto de *grupo de usuarios invisible*.

Definición 5 (Usuarios invisibles)**.** *Si $G \subseteq U$, decimos que G es un* grupo de usuarios invisible *sí y solo sí vale que $G :| \bar{G}$.*

Ejemplo 14 (Seguridad multi-nivel cont.)**.** *Un sistema de capacidades M es MLS sí y solo sí para toda clase de acceso x el grupo U_{+x} es invisible.*

Esta sección debe completarse leyendo [15]

2.6. Análisis de la no interferencia

El contenido de esta sección es un resumen del trabajo realizado por un grupo de investigadores liderado por Ian Sutherland, Tanya Korelsky y Daryl McCullough, cuando trabajaron para el DoD [29].

Aun cuando estamos seguros de tener una buena definición de seguridad[24], no es fácil saber si un sistema la verifica o no. Esto está relacionado con el hecho de que *safety* y *security* son dos clases de propiedades diferentes[25]. Como vimos en Ingeniería de Software I[26], *safety* es una de las clases de propiedades fundamentales de la teoría de Alpern-Schneider sobre sistemas concurrentes; la otra propiedad es vitalidad. En cierto sentido las propiedades de *safety* son más simples de implementar que las de vitalidad porque es suficiente con construir un sistema con menos comportamientos que los indicados en la propiedad. En otras palabras, si S es una propiedad de *safety* y P es una implementación tal que $P \Rightarrow S$, entonces P tiene menos comportamientos que S. Y si P' es una implementación más determinista que P (es decir con aun menos comportamientos que P) entonces P' también verificará S. En lo que concierne a las propiedades de *safety*, aumentar el determinismo garantiza la corrección.

Sin embargo, en lo que concierne a las propiedades de vitalidad lo anterior no es cierto en general. Aumentar el determinismo no es garantía de que la implementación verifique la propiedad.

Se ha argumentado, e incluso a primera vista parece razonable, que las propiedades de seguridad son propiedades de *safety*. El razonamiento es que si un sistema que no hace nada es seguro, entonces la seguridad no requiere que algo ocurra. Si no tiene que ocurrir algo, en consecuencia no son propiedades de vitalidad y entonces deberían ser de *safety*. Sin embargo, las propiedades de *security* no son propiedades de *safety*, en particular porque en general no vale la propiedad descripta más arriba. Es decir, una implementación más determinista que la especificación de seguridad no necesariamente es segura.

Ejemplo 15 (Seguridad no es *safety*)**.** *Digamos que a nivel de especificación se establece que cuando un proceso le solicita un área de memoria al sistema operativo, este entrega parte de la memoria que tiene disponible pero no se especifica un valor particular para escribir en cada celda entregada. Es decir, el sistema operativo puede entregar las celdas con cualquier valor posible. Desde el punto de vista de la seguridad esta especificación es razonable pues el proceso no tiene forma de saber si los valores que recibe en esas celdas contienen información importante o no.*

Una implementación posible para esa especificación es que el sistema operativo sobrescriba cada celda con cero antes de entregarla. Esta implementación es tan segura como la especificación y es más determinista (pues hay menos comportamientos posibles ya que de todos los valores posibles siempre se entrega el mismo).

Ahora pensemos en otra implementación más determinista: el sistema operativo entrega las celdas con el último valor guardado. Es más determinista pues de todas las sobrescrituras posibles el sistema operativo elige no sobrescribir nada. Sin embargo esta implementación es obviamente insegura pues un caballo de Troya solo debería solicitar un área de memoria suficientemente grande, escribir en ella (con repeticiones si fuese necesario) la información secreta y liberar la memoria. Por otro lado un programa ejecutando con una clase de acceso baja solicita toda la memoria posible sabiendo que es muy probable que contenga el secreto buscado.

Dos implementaciones más deterministas, una segura, la otra no. La seguridad no se preserva aumentando el determinismo. Las propiedades de seguridad no son propiedades de safety

Ejemplo 16 (Refinamiento no preserva no interferencia)**.** *Este ejemplo está tomado de [7]. Digamos que un sistema produce de forma no determinista las salidas 0, 1 o un valor secreto h. Claramente este*

[24] En esta sección 'seguridad' es sinónimo 'confidencialidad'.

[25] Dado que 'safety' y 'security' se traducen como 'seguridad', en esta sección, cuando sea necesario, utilizaremos las palabras en inglés para evitar confusiones.

[26] Sugerimos releer el apunte de clase "Seguridad, Vitalidad y Equidad", de esa materia.

sistema verifica la propiedad de no interferencia: "las salidas posibles del sistema son independientes de los valores secretos" ya que son independientes de la actividad de los usuarios H. Sin embargo, un refinamiento posible del sistema es que solo emita la salida h en cuyo caso no cumpliría con la propiedad de seguridad.

El hecho de que *safety* y *security* sean clases de propiedades diferentes, no significa que sean disjuntas. Es decir, en particular, no significa que *ninguna* propiedad de *security* es de *safety*. De hecho hay propiedades de *security* que son también propiedades de *safety* (por ejemplo, el desbordamiento de arreglos, ver Sección 3.1).

No obstante, la no interferencia no es de *safety*. Peor aun, el mayor problema es que la no interferencia *no* es una propiedad en el sentido de Alpern-Schneider. La validez de una propiedad en el contexto de Alpern-Schneider depende de *una* ejecución[27] mientras que la validez de la no interferencia depende de un *conjunto* de ejecuciones (notar que la definición de no interferencia requiere de al menos dos ejecuciones, una donde actúan los usuarios H y la otra donde no lo hacen). De hecho Clarkson y Schneider [7] definen el concepto de *hiperpropiedad*, como un predicado que depende de conjuntos de ejecuciones, dentro del cual pueden expresar la no interferencia. Más aun, en ese mismo trabajo demuestran que, como ya hemos visto, el refinamiento *no* puede invalidar las propiedades de *safety* pero *sí* puede invalidar una hiperpropiedad (o sea que puede invalidar la no interferencia). También definen las hiperpropiedades de *hypersafety* y *hyperliveness* como generalizaciones naturales de las propiedades de *safety* y vitalidad. Clarkson y Schneider no pueden demostrar que la no interferencia es de hypersafety pero solo por un tecnicismo; según ellos sería razonable que lo fuera. Finalmente, como demuestran que el refinamiento preserva hypersafety, sigue siendo razonable que *no necesariamente* preserve no interferencia (puesto que no es de hypersafety).

Esto tampoco significa que la no interferencia *nunca* sea de *safety*. Es decir, hay versiones de la no interferencia (menos generales, obviamente) que se pueden expresar como propiedades de *safety*. Estas versiones usualmente se expresan como sistemas de tipos y por lo tanto el problema de verificar la no interferencia se convierte en un problema de verificación de tipos, que es claramente de *safety* (ver por ejemplo el trabajo de Volpano y otros [32]). Uno de los problemas con estas variantes de la no interferencia es que generan sistemas innecesariamente restrictivos.

2.6.1. Canales encubiertos

Detrás del ataque por medio de caballos de Troya está la idea de que este tipo de programas usa canales de comunicación legítimos (como archivos, IPC, memoria compartida, etc.) para comunicar información de forma ilegítima. Sin embargo los sistemas operativos complejos presentan muchos mecanismos que se pueden usar como canales de comunicación pero que no fueron pensados con ese fin. A este tipo de canal de comunicación se lo llama *canal encubierto*[28].

Definición 6 (Canal encubierto)**.** *Un canal encubierto es cualquier mecanismo que le permite a dos o más procesos comunicar información entre sí, a pesar de que ese mecanismo no fue pensado como un medio para comunicar información.*

En un sistema que implementa BLP, normalmente, un caballo de Troya que ha leído información H utiliza un canal encubierto que tiene en el otro extremo un proceso ejecutando a nivel

[27] Sucesión infinita de estados.
[28] 'covert channel', en inglés.

L para comunicar la información H. Este otro proceso suele llamarse *proceso espía* o simplemente *espía*.

Ejemplo 17 (Canal encubierto)**.** *El Ejemplo 11 muestra un mecanismo que puede ser usado como canal encubierto. Entonces tenemos un caballo de Troya que leerá información H y un proceso espía que ejecutará siempre a nivel L. Ambos acuerdan en usar el archivo* canal.txt *como canal encubierto. Entonces el caballo de Troya lee la información de nivel H, digamos la palabra 'secreto'. En ese momento solo podrá escrbir en archivos de ese nivel. Sin embargo usará los segundos de la fecha de acceso del archivo* canal.txt *para comunicarle esa palabra al espía que estará consultando periódicamente tal fecha. Cuando el caballo de Troya quiere comunicar la 's' espera hasta el segundo 19 porque la 's' es la décima novena letra del alfabeto y escribe cualquier cosa en el archivo* canal.txt*; cuando quiere comunicar la 'e' espera hasta el segundo 5 (del minuto siguiente) y vuelve a escribir cualquier cosa en el archivo* canal.txt*; y así sucesivamente. Aunque el archivo* canal.txt *será de nivel H, como la fecha de acceso no fue clasificada como objeto a proteger el espía al otro extremo del canal encubierto no tendrá problema en consultarla.*

Como todo canal de comunicación los canales encubiertos tienen un cierto ancho de banda que puede medirse en bits o bytes (o unidades superiores) por alguna unidad de tiempo. El canal encubierto del ejemplo anterior tiene un ancho de banda de un caracter (byte) por minuto pero esquemas más sofisticados podrían incrementarlo notablemente. Obviamente nadie espera transmitir una base de datos de cientos de megabytes de a un caracter por minuto. De todas formas no siempre es necesario transmitir la información en sí sino solo una clave de encriptación de unos pocos bits.

Ejemplo 18 (Transmitir una clave por un canal encubierto)**.** *Supongamos que el objetivo de un atacante es hacerse con los datos de una base de datos de cientos de megabytes. Esos datos son consultados por clientes que están fuera de la red corporativa; o sea los datos viajan a través de canales públicos como Internet. Para proteger los datos en tránsito la organización utiliza algún sistema criptográfico con claves de 1024 bits (estos temas serán profundizados en la Unidad 4).*

Como el atacante solo dispone del canal encubierto descripto en el Ejemplo 17, en lugar de transmitir toda la base de datos por este canal solo trasmitirá los 1024 bits de la clave. De esta forma cuando obtenga la clave 'solo' tendrá que intervenir[29] *el canal público, recuperar los datos que viajan encriptados y desencriptarlos con la clave que obtuvo. Si bien estos pasos no son triviales el más complejo por lejos es lograr sacar la clave fuera del servidor. Por ejemplo si la red pública está basada en tecnología ethernet no es tan difícil intervenirla.*

Observar que en este caso particular el canal encubierto podrá transmitir en cada comunicación una secuencia de bits que represente un número no mayor a 59. Esto significa transmitir 6 bits por minuto o sea que la clave podría transmitirse en 3 horas.

Si este ataque parece demasiado complicado, otro ataque posible usando el mismo canal encubierto sería transmitir la contraseña de alguno de los administradores de la base de datos. Para ello el caballo de Troya debería ser un keystroke logger*. . . aunque habría que asegurarse que el sistema no cuenta con un camino confiable para ingresar las contraseñas. Aun así el atacante podría instalar un* keystroke logger *físico, es decir un dispositivo electrónico que se enchufa entre el teclado y la computadora. . . aunque si el teclado es inalámbrico el ataque puede simplificarse notablemente. . . ¿tiene sentido un camino confiable en este caso?*

El ejemplo anterior muestra que en realidad se deben eliminar los canales encubiertos que puedan transmitir al menos 1 bit cada alguna unidad de tiempo razonable (milisegundo,

[29] 'eavesdropping', en inglés.

segundo, minuto, hora. . . aunque probablemente 1 bit/año sea un canal inútil). A este tipo de canales encubiertos los llamaremos canales de un 1 bit de ancho de banda o simplemente de 1 bit.

De todas formas, como cualquier otro canal de comunicación, los canales encubiertos suelen funcionar sobre medios o en entornos que pueden producir errores o ruido, lo que tiende a reducir el ancho de banda efectivo del canal. Además se debe considerar el grado de coordinación que se requiere entre el transmisor (caballo de Troya) y el receptor (el espía), el cual está muy influenciado, entre otras cosas, por el algoritmo de *scheduling* y la cantidad de procesos activos en el sistema operativo. Cuando el canal tiene mucho ruido los datos que recibe el espía no son muy confiables. En este caso la confiabilidad requiere un tratamiento probabilístico y algún protocolo de retransmisión o control de errores. Podemos simplificar la cuestión asumiendo que existen canales encubiertos de 1 bit de ancho de banda pero donde hay un cierto nivel de ruido que implica que el espía puede recibir tres respuestas: 'sí', 'no' o 'tal vez'. El problema con estos canales es que el caballo de Troya solo necesita repetir su mensaje lo suficiente hasta que la respuesta no sea 'tal vez'.

Notar que un canal como los anteriores sigue siendo útil para cierto tipo de información muy crítica. Por ejemplo, la respuesta a la pregunta "¿Aceptará Amazon pagos con bitcoins?" es binaria por lo que un canal de este tipo es más que suficiente. Conocer la respuesta a esa pregunta con minutos de antelación a su anuncio público puede hacer al atacante muy rico.

Si los canales ruidosos de 1 bit son intolerables en ciertos sistemas, aun quedan los llamados canales encubiertos de *medio bit* de ancho de banda. En este caso el caballo de Troya solo puede transmitir una señal, no dos como en los canales de 1 bit.

Ejemplo 19 (Canal encubierto de medio bit)**.** *Las colas de impresión en general son estructuras de datos finitas. Ahora imaginemos que el caballo de Troya puede imprimir documentos de una cola de impresión que es usada por procesos que operan a nivel L. Entonces el espía completa la cola de impresión y trata de enviar un trabajo más. Si el caballo de Troya quiere comunicar algo, imprime uno de los trabajos lo que le permitirá al espía enviar el trabajo extra y de esta forma sabrá que el caballo de Troya se está comunicando. Si el trabajo extra no es aceptado el espía no puede concluir nada pues puede ocurrir que el caballo de Troya no ha encontrado la información que estaba buscando o que aun la está transmitiendo (está intentando imprimir un documento).*

Por ejemplo, con este canal el caballo de Troya solo *podría comunicar un 'sí' o un 'no', pero no ambas, como respuesta a la pregunta " ¿Aceptará Amazon pagos con bitcoins?". No imprimir un documento de una cola de impresión completa no significa necesariamente que se está enviando la otra respuesta.*

Sin embargo, el sistema del ejemplo no verifica no interferencia pues el proceso L recibe salidas diferentes dependiendo de lo que hagan los procesos H. Por otro lado el problema aparece porque la cola de impresión es *finita* porque de lo contrario el espía no podría completarla y así recibir dos respuestas diferentes. Entonces, la solución pasaría por tener colas de impresión infinitas o tener una cola de impresión para cada nivel que es usada solo por procesos de ese nivel. La primera solución es, en general, irrealizable. La segunda restringe innecesariamente la usabilidad del sistema pues los procesos de nivel L *no* pueden enviar datos a procesos de nivel H cuando no hay nada en la política MLS ni en la no interferencia que lo impida. Por ejemplo, un proceso L no podría enviar un mensaje de correo electrónico a un proceso H sin que esto genere un canal encubierto.

Por otra parte, se podría argumentar que podemos convivir con canales de medio bit puesto que transmiten muy poca información. El problema es que dos canales de medio bit se pueden

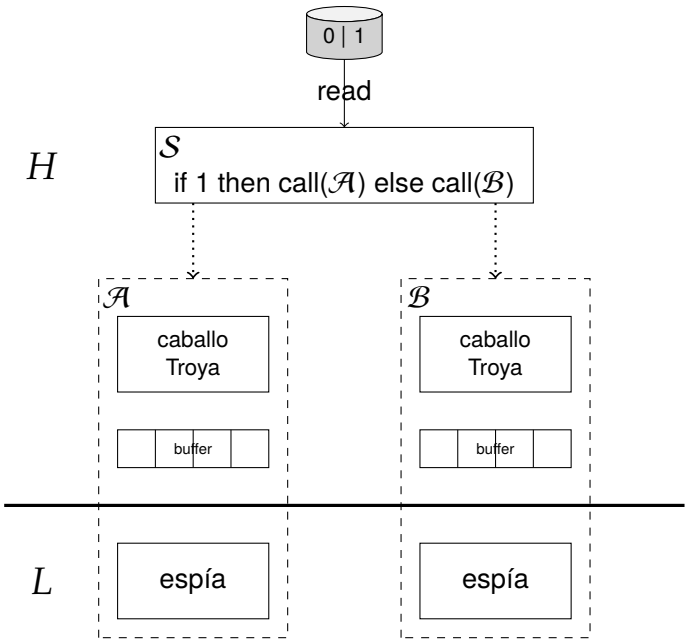


Figura 5: Dos canales de medio bit ($\mathcal{A}$ y $\mathcal{B}$) generan un canal de 1 bit ($\mathcal{S}$)

componer para generar un canal de 1 bit. Esta situación se grafica en la Figura 5. Digamos que empaquetamos un caballo de Troya, su canal de medio bit y su espía en un sistema que llamamos $\mathcal{A}$; y hacemos lo propio con los elementos del otro canal de medio bit al que llamamos $\mathcal{B}$. Asumimos que la respuesta positiva del sistema $\mathcal{A}$ es 'sí' y la de $\mathcal{B}$ es 'no'. Ahora consideramos un tercer caballo de Troya, que llamamos $\mathcal{S}$, que leerá la información necesaria para determinar la respuesta binaria que queremos descubrir. Obviamente $\mathcal{S}$ leerá información H y operará a nivel H. Si la respuesta binaria es 'sí' (por ejemplo Amazon aceptará pagos con bitcoins) $\mathcal{S}$ le enviará una señal al sistema $\mathcal{A}$ el cual le permitirá a su espía ver el 'sí'; si la respuesta binaria es 'no' (Amazon no aceptará pagos con bitcoins) $\mathcal{S}$ le enviará una señal al sistema $\mathcal{B}$ el cual le permitirá a su espía ver el 'no'. Notar que $\mathcal{S}$ se puede comunicar, a nivel H, con los caballos de Troya de $\mathcal{A}$ y $\mathcal{B}$ pues ambos también operan a nivel H.

2.6.2. Hipótesis detrás del modelo de no interferencia

El modelo de Goguen-Meseguer se basa sobre al menos dos hipótesis: el sistema es determinista y no admite interrupciones. En los sistemas de capacidades de Goguen-Meseguer la salida es función de la entrada. Esto significa que el modelo aplica solo a sistemas deterministas. Tampoco aplica a sistemas que admiten *interrupciones*. En estos sistemas los usuarios pueden abortar un proceso que demora demasiado en terminar. Los sistemas que admiten interrupciones se vuelven no deterministas. En efecto, si un proceso está corriendo y el usuario decide interrumpirlo pueden darse dos resultados: si el evento de interrupción llega *antes* de que el proceso haya terminado, el sistema responderá 'proceso interrumpido'; pero si el evento de interrupción llega *luego* de que el proceso haya terminado, el sistema responderá 'proceso inexistente'. O sea que la salida (cualquiera de los dos mensajes) no es función de la entrada (i.e. la secuencia de comandos 'ejecutar comando ; interrumpir proceso'). Más aun, este hecho implica

que el modelo de Goguen-Meseguer *no* tiene en cuenta el tiempo de arribo de cada entrada.

Estas hipótesis tienen la ventaja de que si la función de salida realmente describe *todo* lo que los usuarios pueden ver, cualquier implementación del modelo también será no inteferente pues *todo* lo que los usuarios pueden ver está descripto a nivel de la especificación. Sin embargo, cuando se admite la posibilidad de no determinismo o interrupciones podría darse el caso de que la implementación afecte la forma precisa en que funcionan el no determinismo y las interrupciones, lo que podría invalidar la demostración de seguridad (ver Ejemplo 15).

Pero, ¿cuál es el problema de considerar *solo* sistemas deterministas? El problema es que dos computadoras conectadas en red en general se comportan como un sistema no determinista y en consecuencia el modelo de Goguen-Meseguer no podría ser aplicado a redes de computadoras. Supongamos que una de las dos computadoras es, digamos, un Pentium de 1993 y la otra es un I7 de última generación ambas conectadas por una cable ethernet de 100 Gbit/s. Si el I7 envía a toda velocidad paquetes de datos hacia el Pentium, este no tendrá tiempo de procesarlos a todos. En consecuencia la salida que vería un usuario sentado en el Pentium luciría no determinista pues dependerá de cuáles paquetes el Pentium sea capaz de procesar y cuáles no.

Evitar esta forma de no determinismo requiere *buffers* infinitos o *buffers* finitos con procesos que esperan a que haya espacio. La primera solución es, en general, irrealizable y la segunda genera canales encubiertos de medio bit.

El problema del no determinismo aparece incluso en una única computadora con un sistema operativo que al menos simula la concurrencia. Esto puede verse con esta simple especificación CSP:

$$e \rightarrow a_1 \rightarrow b_1 \rightarrow STOP \parallel e \rightarrow a_2 \rightarrow b_2 \rightarrow STOP$$

que representa a dos procesos que esperan una entrada en común (e) que luego es procesada por cada uno de ellos de forma independiente (a_1 y a_2) lo que produce dos salidas diferentes (b_1 y b_2). En este caso tenemos dos secuencias de salidas posibles, $\langle b_1, b_2 \rangle$ y $\langle b_2, b_1 \rangle$, dependiendo de cuál de los dos procesos trabaja más rápido[30]. O sea que dos corridas diferentes de esta composición concurrente pueden dar resultados distintos aunque se les haya provisto la misma entrada. En otras palabras el sistema es no determinista. O sea que el modelo de Goguen-Meseguer no aplicaría a sistemas operativos que al menos simulan la concurrencia.

McCullough [17, 18] extiende el concepto de no interferencia a sistemas no deterministas y lo llama *no interferencia generalizada*[31].

2.6.3. Sistemas manifiestamente seguros

Llegados a este punto cabe preguntarse si es posible construir sistemas MLS que no sean innecesariamente restrictivos y tales que podamos dar su especificación y garantizar que cualquier implementación preserve esas propiedades.

En este caso Sutherland, Korelsky y McCullough [29] hablan de *sistemas manifiestamente seguros*. Uno de ellos se describe gráficamente en la Figura 6. Allí los bloques son máquinas ejecutando a nivel H y L, respectivamente. Es decir cada una de esas máquinas procesa información a lo sumo del nivel indicado pero todos sus procesos ejecutan exactamente en ese nivel.

[30] Como ejercicio podrían obtener el proceso secuencial equivalente para comprobar que efectivamente esas son las salidas posibles.

[31] 'generalized noninterference', en inglés.

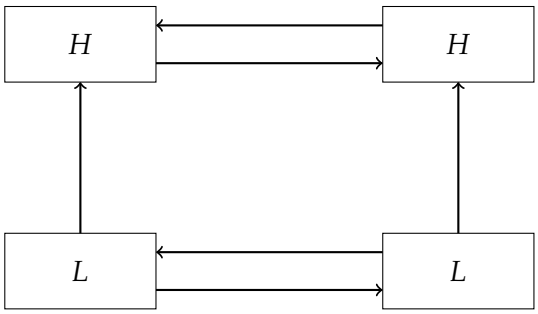


Figura 6: Descripción gráfica de un sistema manifiestamente seguro. Las flechas representan comunicaciones unidireccionales.

Las flechas representan mecanismos de comunicación unidireccionales o semidúplex[32], es decir mecanismos donde solo son posibles comunicaciones en un único sentido. Como muestra la figura, los procesos L, por un lado, y los procesos H, por el otro, pueden comunicarse entre sí libremente, pero solo los procesos L pueden enviar información a los procesos H y lo pueden hacer por medio de mecanismos de comunicación que no tengan ningún efecto lateral 'hacia abajo'.

El sistema de la figura es manifiestamente seguro aun cuando solo se ha dado una especificación muy abstracta (y breve). Según los autores la clave en la definición de sistemas seguros es la *conectividad* entre los componentes que procesan la información. Si la conectividad es manifiestamente segura, entonces cualquier implementación que preserve la conectividad será segura.

En la siguiente sección veremos un modelo que se basa en esta idea.

Esta sección debe completarse leyendo [29, capítulos 1, 2.3, 2.4, 3 y 4]

2.7. Multi-ejecución segura

Lo que hoy se conoce como *multi-ejecución segura* (SME[33]) fue presentado por primera vez por Maximiliano Cristiá en las 5^{tas} Jornadas de Ciencias de la Computación en octubre de 2007[34], fue publicado por primera vez por Cristiá y Mata en 2009 [8], pero ganó reconocimiento internacional con la publicación de los belgas Dominique Devriese y Frank Piessens [11].

La idea detrás del modelo de SME es la implementación del sistema manifiestamente seguro de la Figura 6 a nivel de un sistema operativo. Es decir, en este caso, los bloques son procesos de un sistema operativo y las flechas mecanismos de comunicación entre procesos controlados por el sistema operativo. Básicamente un sistema operativo que implementa SME hace un *fork* de un proceso cada vez que accede datos con una clase de acceso más alta. Informalmente, SME funciona de la siguiente forma:

- Cuando un proceso es lanzado se lo clasifica en L
- Esta clasificación se mantiene en tanto acceda información L

[32] 'half-duplex', en inglés.
[33] 'secure-multi execution', en inglés
[34] Ver en `http://jcc.dcc.fceia.unr.edu.ar/2007/cronograma.html`.

- Cuando el proceso lee información H, el sistema operativo hace un *fork* creando dos procesos, digamos P_L y P_H. P_L es el mismo proceso que ejecutaba a nivel L y no se le permite acceder a la información H; P_H es un nuevo proceso clasificado con H al que se le permite acceder a la información H.
- Un proceso clasificado en L puede escribir información en objetos clasificados con L y H; un proceso clasificado en H solo puede escribir información en objetos clasificados con H.

De esa forma, un sistema SME es seguro y a la vez tan usable como lo permite MLS.

Ejemplo 20 (SME es más usable que BLP)**.** *Si un usuario de un sistema BLP quiere editar un archivo L y un archivo H, debe lanzar dos veces el editor de textos pasándole como parámetro cada uno de los archivos, o sea:*

```
$ gedit archivo_L &
$ gedit archivo_H &
```

En cambio en un sistema SME esto lo hace automáticamente el sistema operativo; no se requiere intervención del usuario:

```
$ gedit archivo_L  archivo_H &
```

crea dos procesos, uno que accede solo a archivo_L *y el otro que accede a ambos archivos pero que solo puede escribir en* archivo_H*, de manera tal que el usuario puede editar ambos.*

Esta automatización es más evidente y útil en sistemas que funcionan con menos interacción con el usuario, como por ejemplo un sistema de distribución de correo electrónico que debe operar a varios niveles de seguridad diferentes.

Formalización de SME. La formalización de SME se hace de la siguiente forma:

1. Se da la sintaxis y semántica de un lenguaje de programación abstracto que constituye el conjunto de programas que pueden ejecutarse. Usualmente es un lenguaje muy simple pero Turing completo. En particular nosotros daremos un lenguaje que incluye aritmética de punteros.
2. Se especifica el funcionamiento de seguridad del sistema operativo o del núcleo de seguridad. Concretamente se especifican las llamadas al sistema relevantes a la seguridad (normalmente las que permiten leer y escribir datos).
3. Se adapta y formaliza la noción de no interferencia
4. Se demuestra que el sistema operativo preserva esa noción de no interferencia.

El lenguaje de programación. La gramática del lenguaje de programación que usaremos en este modelo se resume en la Figura 7. El único tipo del lenguaje son los números naturales simbolizado con $\mathbb{N}$. Las expresiones $*v$ y $\&v$ tienen el significado habitual con respecto a la aritmética de punteros (i.e., $*v$ es el contenido de la celda de memoria apuntada por v y $\&v$ es la dirección de la variable v). El operador $\boxplus$ representa cualquier operador binario de los naturales. syscall() representa cualquier llamada al sistema operativo; el primer parámetro determina cuál llamada se quiere ejecutar y los restantes parámetros son los que se le pasan a dicha llamada. Cuando una expresión es la condición de una sentencia if - fi o while - done se adopta alguna convención para determinar el significado de verdadero y falso (por ejemplo 1 y 0).

$$
\begin{aligned}
&\mathit{Expr} ::= \mathbb{N} \mid var \mid *var \mid \&var \mid \mathit{Expr} \boxplus \mathit{Expr} \\
&\mathit{BasicStatement} ::= \\
&\qquad \mathsf{skip} \mid \mathit{var} := \mathit{Expr} \mid *\mathit{var} := \mathit{Expr} \mid \mathsf{syscall}(\mathbb{N}, \mathit{arg}_1, \ldots, \mathit{arg}_n) \\
&\mathit{ConditionalStatement} ::= \\
&\qquad \mathsf{if}\ \mathit{Expr}\ \mathsf{then}\ \mathit{Program}\ \mathsf{fi} \mid \mathsf{while}\ \mathit{Expr}\ \mathsf{do}\ \mathit{Program}\ \mathsf{done} \\
&\mathit{BasicAndConditional} ::= \mathit{BasicStatement} \mid \mathit{ConditionalStatement} \\
&\mathit{Program} ::= \mathit{BasicAndConditional} \mid \mathit{Program}\ ;\ \mathit{Program}
\end{aligned}
$$

Figura 7: Gramática del lenguaje de programación

La semántica del lenguaje de programación se resume en la Figura 8, donde $\oplus$ es el operador relacional definido en Z. La semántica está dada en términos de cómo cada sentencia del lenguaje modifica la memoria y el entorno del proceso[35]. Para ello definimos el conjunto de todas las memorias posibles, $\mathbb{M}$; el conjunto de todos los entornos posibles, $\mathbb{E}$; el conjunto de todos los programas posibles, $\mathbb{P}$; y la función $\mathcal{P}$ que toma una memoria, un entorno y un programa y devuelve una memoria y un entorno: $\mathcal{P} : \mathbb{M} \times \mathbb{E} \times \mathbb{P} \to \mathbb{M} \times \mathbb{E}$. Entonces, $\mathcal{P}(m,e,p) = (m',e')$ se interpreta como que el programa p arrancó su ejecución en (m,e) y terminó en (m',e'). A su vez una memoria es una función del conjunto de todas las variables, $\mathbb{V}$, en $\mathbb{N}$ (es decir $\mathbb{M} \mathrel{\widehat{=}} \mathbb{V} \to \mathbb{N}$). Si $m \in \mathbb{M}$ y $x \in \mathbb{V}$, $m(x)$ es el valor almacenado en la variable x. Por otro lado, definimos $\mathbb{E} \mathrel{\widehat{=}} \mathbb{D} \to \mathrm{seq}\,\mathbb{N}$, donde $\mathbb{D}$ es un conjunto de dispositivos de entrada y salida. Una función $d \in \mathbb{E}$ representa el estado de los dispositivos de E/S: $d(i)$ representa los datos que están disponibles en el dispositivo de entrada i de manera tal que el sistema los puede leer en ese orden; y $d(o)$ representa los datos escritos por el sistema en el dispositivo de salida o.

Entonces, en la Figura 8 se da la definición de $\mathcal{P}$ para cada sentencia del lenguaje. Notar que en la parte inferior de la Figura 8 se define $\mathcal{P}$ para expresiones, la cual se distingue por el número de parámetros que recibe y retorna. Como el lenguaje admite referencias, asumimos que existe una función biyectiva $\mathcal{A}$ que retorna el nombre de variable almacenado en una dirección de memoria dada. Si $x \in \mathbb{V}$ y $m \in \mathbb{M}$ entonces definimos $\overrightarrow{x,m} = (\mathcal{A} \circ m)(x)$. O sea que $\overrightarrow{x,m}$ es la variable referida por x en m. Además definimos $\mathsf{read}(i,x)$ como $\mathsf{syscall}(1,i,x)$, es decir $\mathsf{read}()$ es la primera llamada al sistema donde i es el dispositivo de entrada y x es la variable donde se almacenará el valor que se lea; de forma similar definimos $\mathsf{write}(o,a)$ como $\mathsf{syscall}(2,o,a)$ donde o es el dispositivo de salida y a la expresión que se quiere emitir.

Es decir que solo consideramos dos llamadas al sistema: una que toma entrada del entorno y otra que emite salida al entorno. Esto es suficiente como para mostrar el funcionamiento fundamental de un sistema que pretende verificar no interferencia puesto que lo importante son las entradas y salidas.

Imposición de no interferencia. En esta sección daremos la especificación de seguridad del sistema operativo. Esta especificación está dada en términos de una máquina, $\mathcal{S}$, que ejecuta programas, escritos en el lenguaje de programación que describimos más arriba, de forma tal

[35] Es necesario incluir el entorno porque el lenguaje incluye E/S.

$$\mathcal{P} : \mathbb{M} \times \mathbb{E} \times \mathbb{P} \to \mathbb{M} \times \mathbb{E}$$

$$\mathcal{P}(m,e,\mathsf{skip}) = (m,e) \tag{$\mathcal{P}_1$}$$
$$\mathcal{P}(m,e,x := a) = (m \oplus \{x \mapsto \mathcal{P}(m,a)\},e) \tag{$\mathcal{P}_2$}$$
$$\mathcal{P}(m,e,*x := a) = (m \oplus \{\overrightarrow{x,m} \mapsto \mathcal{P}(m,a)\},e) \tag{$\mathcal{P}_3$}$$
$$\mathcal{P}(m,e,\mathsf{read}(i,x)) = (m \oplus \{x \mapsto head \circ e(i)\},e \oplus \{i \mapsto tail \circ e(i)\}) \tag{$\mathcal{P}_4$}$$
$$\mathcal{P}(m,e,\mathsf{write}(o,a)) = (m,e \oplus \{o \mapsto \langle \mathcal{P}(m,a)\rangle \frown e(o)\}) \tag{$\mathcal{P}_5$}$$
$$\mathcal{P}(m,e,\mathsf{if}\ c\ \mathsf{then}\ p\ \mathsf{fi}) = \textbf{if}\ \mathcal{P}(m,c)\ \textbf{then}\ \mathcal{P}(m,e,p)\ \textbf{else}\ (m,e) \tag{$\mathcal{P}_6$}$$
$$\mathcal{P}(m,e,\mathsf{while}\ c\ \mathsf{do}\ p\ \mathsf{done}) = \textbf{if}\ \mathcal{P}(m,c)\ \textbf{then}\ \mathcal{P}(m,e,p\ ;\ \mathsf{while}\ c\ \mathsf{do}\ p\ \mathsf{done})\ \textbf{else}\ (m,e) \tag{$\mathcal{P}_7$}$$
$$\mathcal{P}(m,e,p_1\ ;\ p_2) = \mathcal{P}(\mathcal{P}(m,e,p_1),p_2) \tag{$\mathcal{P}_8$}$$

$$\mathcal{P}(m,n) = n \tag{$\mathcal{P}_9$}$$
$$\mathcal{P}(m,x) = m(x) \tag{$\mathcal{P}_{10}$}$$
$$\mathcal{P}(m,*x) = m(\overrightarrow{x,m}) \tag{$\mathcal{P}_{11}$}$$
$$\mathcal{P}(m,\&x) = \mathcal{A}^{-1}(x) \tag{$\mathcal{P}_{12}$}$$
$$\mathcal{P}(m,a_1 \boxplus a_2) = \mathcal{P}(m,a_1) \boxplus \mathcal{P}(m,a_2) \tag{$\mathcal{P}_{13}$}$$

Figura 8: Semántica del lenguaje de programacion ($m \in \mathbb{M}, p, p_i \in \mathbb{P}$)

que se garantiza la no interferencia. $\mathcal{S}$ trabaja controlando el flujo de información en términos de la entrada y salida de los procesos. Como muestra la Figura 9, se asume que $\mathcal{S}$ se interpone entre los procesos y el hardware. De esta forma cada vez que un proceso quiere acceder a información o interactuar con el usuario o un sistema externo debe pasar por el control de $\mathcal{S}$. Para todo propósito práctico se puede pensar que $\mathcal{S}$ es parte del sistema operativo. Es decir, $\mathcal{S}$ puede controlar todas las llamadas al sistema efectuadas por los procesos.

El tipo de $\mathcal{S}$ es:

$$\mathcal{S} : \mathbb{M} \times \mathbb{M} \times \mathbb{E} \times \mathbb{P} \to \mathbb{M} \times \mathbb{M} \times \mathbb{E}$$

donde, informalmente, la primera memoria es la que usan los procesos que ejecutan a nivel L y la segunda es la que usan aquellos que lo hacen a nivel H. Es decir que $\mathcal{S}(m_L, m_H, e, p) = (m'_L, m'_H, e')$ significa que el proceso p comienza a ejecutar con las memorias m_L y m_H y con el entorno e y finaliza dejando las memorias y el entorno en el estado (m'_L, m'_H, e'). Notar que hay dos memorias pero un único entorno; es decir ambas copias del proceso usan el mismo entorno.

La definición de $\mathcal{S}$ la pueden ver en la Figura 10, donde las definiciones de m'_L, m'_H, etc. se encuentran al final de la figura. En este caso $\mathcal{S}$ funciona con cuatro dispositivos de E/S: $\mathbb{D} \mathrel{\widehat{=}} \{i_L, i_H.o_L, o_H\}$ tales que $sc(i_L) = sc(o_L) = L$ y $sc(i_H) = sc(o_H) = H$. Si bien se espera que cada proceso use solo la memoria L hasta que sea necesario duplicarlo para así comenzar a usar la H, la especificación de $\mathcal{S}$ dada en la Figura 10 supone que el proceso ya fue duplicado. Esto se puede apreciar, por ejemplo, en la regla ($\mathcal{S}_2$) donde ante una asignación se modifican ambas memorias.

Posiblemente los casos más interesantes sean los de read() y write() pues se puede ver cómo

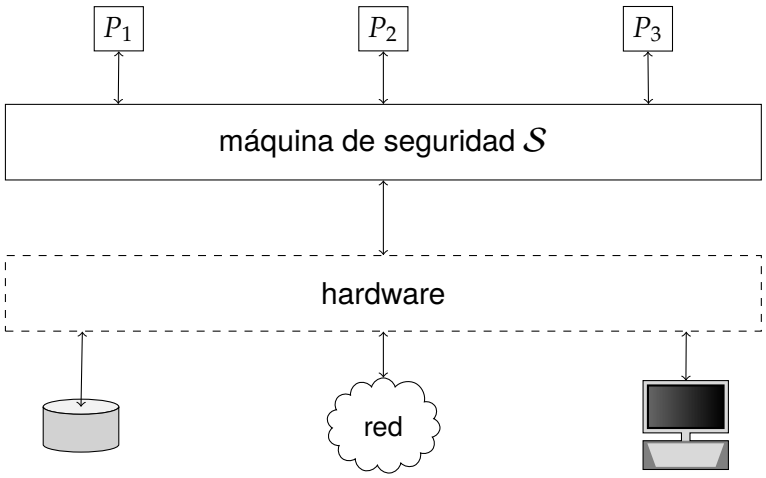


Figura 9: Representación gráfica de $\mathcal{S}$

$\mathcal{S}$ altera la E/S normal de los procesos para imponer la no interferencia. Por ejemplo en ($\mathcal{S}_7$) los dispositivos de salida de nivel L permanecen sin cambios; y en ($\mathcal{S}_4$) se ve cómo la entrada de nivel L la toman ambas copias del proceso. El caso ($\mathcal{S}_8$) también es revelador pues se ve que ambas copias del proceso ejecutan por separado pero $\mathcal{S}$ junta las salidas de ambos (i.e. $e' = e'_L \cup e'_H$).

El teorema de no interferencia. El paso final es adaptar el concepto de no interferencia a este modelo y demostrar que $\mathcal{S}$ lo verifica. En no interferencia necesitamos dos ejecuciones del sistema tales que las entradas de nivel L sean las mismas para comprobar que, al final, las salidas del nivel L también son las mismas; es decir, las únicas diferencias entre las dos ejecuciones son las entradas y las salidas de nivel H. En el modelo de SME la E/S está dada por el tercer parámetro de $\mathcal{S}$, que normalmente llamamos e y es un elemento de $\mathbb{E}$. Por lo tanto, tomamos $e_1, e_2 \in \mathbb{E}$ tales que $e_1(i_L) = e_2(i_L)$. Cada ejecución se hará sobre la misma memoria de nivel L pero (posiblemente) diferentes memorias de nivel H; en consecuencia necesitamos tres memorias, m_1, m_2 y m_3. Luego tomamos un programa cualquiera, $p \in \mathbb{P}$, y vemos cómo es la salida de nivel L que produce $\mathcal{S}$ al ser ejecutado p partiendo, por un lado, de m_1, m_2 y e_1 y, por el otro, de m_1, m_3 y e_2. El siguiente teorema formaliza lo antedicho.

Teorema (No interferencia)**.**

$$
\begin{aligned}
&\forall e_1, e_2 \in \mathbb{E};\ m_1, m_2, m_3 \in \mathbb{M};\ p \in \mathbb{P} \ \bullet \\
&\quad e_1(i_L) = e_2(i_L) \\
&\quad \wedge\ \mathcal{S}(m_1, m_2, e_1, p) = (m'_1, m'_2, e'_1) \wedge \mathcal{S}(m_1, m_3, e_2, p) = (m'_3, m'_4, e'_2) \\
&\qquad \Rightarrow e'_1(o_L) = e'_2(o_L)
\end{aligned}
$$

La demostración del teorema es por medio de inducción estructural sobre p.

Análisis de SME. SME no está exenta de problemas. En primer lugar es evidente que recarga el sistema pues se necesita un proceso por cada objeto de clase de acceso diferente a las de los que estén en uso. En el mismo sentido se requiere más memoria.

$\mathcal{S} : \mathbb{M} \times \mathbb{M} \times \mathbb{E} \times \mathbb{P} \rightarrow \mathbb{M} \times \mathbb{M} \times \mathbb{E}$

$$\mathcal{S}(m_L, m_H, e, \mathsf{skip}) = (m_L, m_H, e) \tag{$\mathcal{S}_1$}$$

$$\mathcal{S}(m_L, m_H, e, x := expr) = (m'_L, m'_H, e) \tag{$\mathcal{S}_2$}$$

$$\mathcal{S}(m_L, m_H, e, *x := expr) = (m'_L, m'_H, e) \tag{$\mathcal{S}_3$}$$

$$\mathcal{S}(m_L, m_H, e, \mathsf{read}(i_L, x)) = (m'_L, \mathcal{P}(m_H, e_L, \mathsf{read}(i_L, x)).1, e'_L \cup e_H) \tag{$\mathcal{S}_4$}$$

$$\mathcal{S}(m_L, m_H, e, \mathsf{read}(i_H, x)) = (m_L, m'_H, e_L \cup e'_H) \tag{$\mathcal{S}_5$}$$

$$\mathcal{S}(m_L, m_H, e, \mathsf{write}(o_L, a)) = (m_L, m_H, e'_L \cup e_H) \tag{$\mathcal{S}_6$}$$

$$\mathcal{S}(m_L, m_H, e, \mathsf{write}(o_H, a)) = (m_L, m_H, e_L \cup e'_H) \tag{$\mathcal{S}_7$}$$

$$\mathcal{S}(m_L, m_H, e, \mathsf{if}\ c\ \mathsf{then}\ p\ \mathsf{fi}) = (m'_L, m'_H, e') \tag{$\mathcal{S}_8$}$$

$$\mathcal{S}(m_L, m_H, e, \mathsf{while}\ c\ \mathsf{do}\ p\ \mathsf{done}) = (m'_L, m'_H, e') \tag{$\mathcal{S}_9$}$$

$$\mathcal{S}(m_L, m_H, e, p_1\ ;\ p_2) = \mathcal{S}(\mathcal{S}(m_L, m_H, e, p_1), p_2) \tag{$\mathcal{S}_{10}$}$$

donde

$m'_L = \mathcal{P}(m_L, e, \bar{p}).1, \quad m'_H = \mathcal{P}(m_H, e, \bar{p}).1$

$e'_L = \mathcal{P}(m_L, e_L, \bar{p}).2, \quad e'_H = \mathcal{P}(m_H, e_H, \bar{p}).2$

$e' = e'_L \cup e'_H$

$e_L = \{i_L, o_L\} \lhd e, \quad e_H = \{i_H, o_H\} \lhd e$

$\bar{p}$ es el cuarto parámetro de la $\mathcal{S}$ sobre la cual se aplica la regla correspondiente

Figura 10: Especificación de la máquina de seguridad

También continúa habiendo problemas de usabilidad. Por ejemplo, supongamos que tenemos tres niveles de seguridad $L < H < V$. Un usuario lanza un editor de textos que inicia en nivel L. Luego abre un archivo de nivel V por lo que el editor se duplica en dos, uno de nivel L y otro de nivel V. ¿Para qué sirve el editor L si no tiene archivos abiertos? ¿Se lo puede cerrar? ¿Se lo puede volver a abrir? ¿Cuál de los dos editores debería usar el usuario para abrir un archivo de nivel H? ¿Puede hacerlo desde el editor de nivel V?

Una de las razones que llevaron a proponer SME fue la falta de practicidad de otras soluciones que proponen desarrollar todas las aplicaciones nuevamente de forma tal que esté demostrado que son no interferentes. ¿De qué magnitud serían los cambios en las aplicaciones existentes si tuvieran que ejecutar sobre un sistema SME? Por ejemplo, hay aplicaciones que dejan archivos temporales en un directorio definido por la configuración. ¿Podrían dejar archivos temporales de diferentes niveles de seguridad en el mismo directorio?

3. Unidad III: Seguridad en lenguajes de programación

En esta unidad veremos ataques que son posibles debido a malas prácticas de programación y estudiaremos algunas soluciones posibles. Algunos de estos ataques evidencian la necesidad de asumir la hipótesis del universo hostil. En efecto, algunas de las vulnerabilidades que permiten estos ataques son simples errores de programación que en un universo benigno no tendrían mayor impacto sobre el sistema. En algunos casos, incluso, el programa puede ser correcto respecto de las propiedades de *safety* que se hayan especificado.

3.1. Desbordamiento de arreglos

El ataque por *desbordamiento de arreglo*[36] consiste en encontrar cierto tipo de error en un programa y enviarle una entrada no especificada (o extraña o no esperada) la cual contiene código de ataque. Cuando se produce el error, el flujo de control del programa pasa al código de ataque que forma parte de la entrada enviada por el atacante. A partir de ese momento, en general, el atacante toma el control del programa lo que le permite efectuar diversos ataques sobre el sistema donde ejecuta ese programa.

Antes de ver los detalles del ataque por desbordamiento de arreglo cabe preguntarse cuáles son los programas que conviene atacar de esta forma. La respuesta más amplia es que, en general, este ataque es útil para atacar cualquier programa. Sin embargo hay algunas situaciones en que el ataque es más conveniente o efectivo:

- *Ataques remotos*. Es decir el atacante no es un usuario del sistema que quiere atacar y por lo tanto lo debe hacer desde otra computadora. En este caso hay tres clases de programas que son objetivos ideales para el ataque por desborde de arreglo:
  - servidores TCP/IP (servidores FTP, HTTP, etc.)
  - programas que reciben datos desde el exterior tales como aplicaciones de correo electrónico, navegadores web, etc. y cualquiera de las aplicaciones que se ejecutan más o menos automáticamente cuando arriba algún tipo de archivo específico (visores PDF, reproductores de vídeo, etc.)
  - biblotecas (o API) que se utilizan en cualquiera de los casos anteriores

[36] 'buffer overflow', en inglés.

- *Ataques internos*. Es decir el atacante ya tiene una cuenta de usuario en la computadora que quiere atacar pero no tiene acceso a la información que quiere copiar, destruir, etc. En este caso los programas a atacar son los que le permiten al atacante *escalar privilegios*. Esta clase de programas depende del caso particular pero en general son programas que usan alguna forma de *delegación de permisos*. Es decir, mecanismos del sistema operativo que permiten que un programa o parte de él ejecute temporalmente con más permisos de los que tiene el usuario que lo ejecutó. En sistemas tipo UNIX la forma estándar de delegación de permisos se llama *set user ID* o *set UID* o *SUID*.

3.1.1. El mecanismo SUID

Esta sección es opcional y solo la deben leer si no recuerdan o no saben cómo funciona el mecanismo SUID en sistemas UNIX.

En UNIX un *programa SUID* es un programa que ejecuta un bloque de sentencias con la identidad de un usuario distinto al que lo lanzó. Los programas SUID se pueden identificar con el comando ls:

```
$ ls -l /usr/bin/passwd
```

da como resultado

```
-rwsr-xr-x 1 root root 54256 may 16 2017 passwd
```

donde la 's' indica que el ejecutable passwd es un programa SUID.

¿Por qué el programa passwd (que sirve para que los usuarios cambien sus contraseñas) es un programa SUID? Las contraseñas de los usuarios se guardan en el archivo /etc/shadow cuyos permisos son:

```
ls -l /etc/shadow
-rw-r- - - - - 1 root shadow 1505 nov 22 21:42 shadow
```

donde se puede ver que solo el dueño del archivo (i.e. root) tiene permiso para escribirlo. Entonces, ¿cómo hace un usuario ordinario para cambiar su contraseña siendo que solo root puede modificar shadow? Precisamente, el usuario ejecuta el programa passwd el cual en algún momento le solicita al sistema operativo cambiar su identidad a la de su dueño (es decir, root), como el programa es SUID el sistema operativo acepta la petición y en consecuencia el programa puede abrir shadow en modo de escritura. Cuando la nueva contraseña se graba en el archivo, passwd lo cierra y le solicita al sistema operativo volver a su identidad original[37].

La estructura básica del código de passwd sería la siguiente:

```
main(. . . ) {
. . . . . .
. . . . . .// ejecuta a nombre del usuario ordinario
. . . . . .
setuid(...) // comienza a ejecutar como root
. . . . . .grabar la nueva contraseña . . . . . .
setuid(...) // termina de ejecutar como root
. . . . . .
```

[37] Las solicitudes de cambio de identidad se hacen por medio de la llamada al sistema setuid.

```
......// ejecuta a nombre del usuario ordinario
......
}
```

es decir, solo la porción crítica del código (respecto de la seguridad) se ejecuta a nombre de root. De esta forma se minimiza la porción de código que se vuelve susceptible de ser atacada pues antes y después de ella el programa ejecuta con la identidad del usuario que lo lanzó (y por lo tanto el atacante no obtendría ninguna ventaja).

Además del mecanismo SUID existe el *set group ID* o *SGID*.

3.1.2. Detalles técnicos del desbordamiento de arreglos

Para comprender el error de programación que permite este tipo de ataque y para comprender el ataque en sí, necesitamos entender el comportamiento y la estructura generales de un proceso.

De todas formas, lo que vamos a ver aplica más que nada a versiones más o menos antiguas de sistemas operativos tipo UNIX. En la actualidad existen varias protecciones que desactivan estos ataques o los hacen muy complejos. Aun así, técnicas más complejas de la que vamos a mostrar aquí, por ejemplo *return-oriented programming* (ROP), siguen siendo útiles. Por otro lado, el error de programación y la técnica básica de ataque siguen siendo conceptualmente válidos aun cuando no sean posibles en los sistemas operativos más modernos.

El ataque por desbordamiento de arreglo consiste esencialmente en modificar la dirección de retorno de una subrutina de forma tal de alterar el flujo de control esperado de un programa. Se logra sobrescribir la dirección de retorno si una subrutina no controla la longitud de un arreglo y referencia posiciones del arreglo que en realidad no existen.

Ejemplo 21 (Arreglos de caracteres en C)**.** *En el lenguaje C se utiliza una convención para marcar la última componente usada en los arreglos de caracteres que consiste en poner en la componente siguiente el caracter* `'\0'` *(barra cero). Por ejemplo, en el siguiente arreglo:*

```
{'h','e','l','l','o',' ','w','o','r','l','d','\0','z','2','%'}
```

las componentes `'z','2','%'` *no son consideradas por la mayoría de las funciones de la libc y por la mayoría de los programadores puesto que están luego del caracter* `'\0'`*. Es decir muchos programadores escriben código que procesa arreglos de caracteres con ciclos de la forma:*

```
while(*s != '\0') { . . . }
```

En consecuencia, si ninguna de las componentes de un arreglo de caracteres es `'\0'`*, el ciclo comenzará a referir posiciones de memoria que no son del arreglo. Es decir* desbordará el arreglo*. Cuando se alcance una posición de memoria fuera del área de memoria asignada al proceso se producirá una falla de segmento y el programa terminará abruptamente con el mensaje '*segmentation fault*'.*

Suponiendo que haya un ciclo que no controla correctamente la longitud de un arreglo, ¿cómo hacemos para llegar hasta él? A través de las entradas del programa. Es decir, el atacante debe proveer una entrada tal que haga que el flujo de control del programa pase por ese ciclo mal implementado.

Ejemplo 22 (Entradas para desbordar arreglos)**.** *Si el objetivo de ataque es un visor PDF que sabemos tiene un ciclo mal implementado cuando debe presentar una tabla, se debe proveer un documento PDF que contenga una tabla. Además esta tabla deberá tener alguna particularidad que haga que el ciclo desborde*

Figura 11: Estructura de la memoria de un proceso

el arreglo. Por ejemplo, podría ser una cantidad de filas o columnas inusualmente grande o el texto de alguna celda contiene caracteres en una codificación no prevista, etc.

La Figura 11 muestra la estructura de la memoria de un proceso. Cada bloque se denomina *segmento de memoria*. El segmento de texto tiene un tamaño fijo e incluye el código ejecutable. Este segmento es generado por el compilador y el ensamblador y no se puede modificar. Puede ser compartido por varios procesos que ejecutan el mismo programa. El segmento de datos contiene el espacio para almacenar las variables globales y estáticas. El tamaño de este segmento puede modificarse a medida que el proceso ejecuta.

El segmento que nos interesa es la pila. Aquí es donde se pueden producir los desbordamientos de arreglos. Este segmento tiene un tamaño variable y su contenido puede ser modificado. La pila se utiliza para administrar la ejecución de subrutinas junto a sus variables y datos.

La Figura 12 muestra cómo crece una pila a medida que se invocan subrutinas (es decir estamos viendo más en detalle el segmento inferior de la Figura 11). A la izquierda tenemos los *marcos* de pila correspondientes a cada subrutina de la derecha. Cada marco almacena los parámetros y variables de la subrutina correspondiente. Observar que main() invoca a f() la cual a su vez invoca a g(). Cuando el programa inicia el único marco en la pila es el correspondiente a main(); cuando main() invoca a f() se *apila* su marco; y cuando esta última invoca a g() se apila su marco. Notar que la pila crece hacia abajo. Simétricamente, cuando g() termina su marco se quita de la pila; y así se continúa desapilando a medida que las subrutinas van terminando.

En la Figura 13 mostramos la estructura de cada marco de pila. Las variables locales a la subrutina se almacenan en la parte inferior del marco, luego viene la dirección de retorno y finalmente los parámetros pasados por valor. Si una variable local es un arreglo las componentes con índice superior se ubican en las posiciones de memoria más altas. La dirección de retorno corresponde a la ubicación de la instrucción siguiente a aquella donde se efectúo la invocación. Las áreas en gris son las importantes puesto que una de ellas almacena un arreglo y la otra es la dirección de retorno.

El primer paso en el ataque es sobrescribir la dirección de retorno. Para esto es necesario que una subrutina refiera componentes de un arreglo declarado localmente que no existen. A su vez esto se logra si tal subrutina no controla la longitud de algún arreglo (o lo hace de forma errónea). La función strcpy() de la biblioteca estándar de C es famosa por no implementar ese control. En la parte derecha de la Figura 14 podemos ver el código de strcpy()[38]. Esta función copia el contenido del arreglo t en el arreglo s[39]. Notar que en el ciclo while no se tienen en cuenta las longitudes de los arreglos. Por lo tanto si t es más largo que s habrá un desborde de

[38]En realidad en la figura el código está escrito de una forma que creemos es mucho más clara que la verdadera implementación: while(*s++=*t++).

[39]Más que arreglos s y t son punteros a caracter.

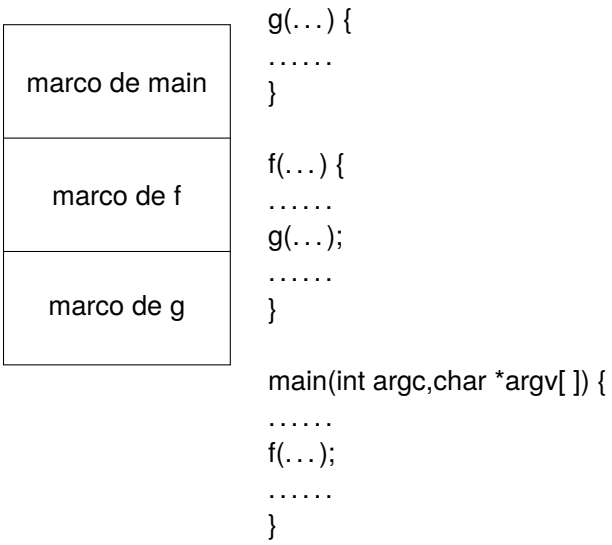


Figura 12: Relación entre la pila y las subrutinas

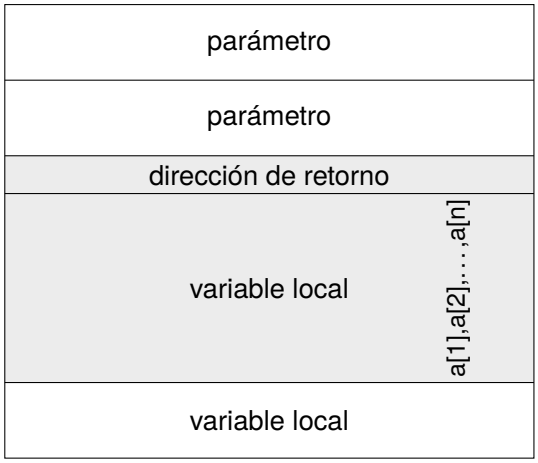


Figura 13: Estructura de un marco de pila

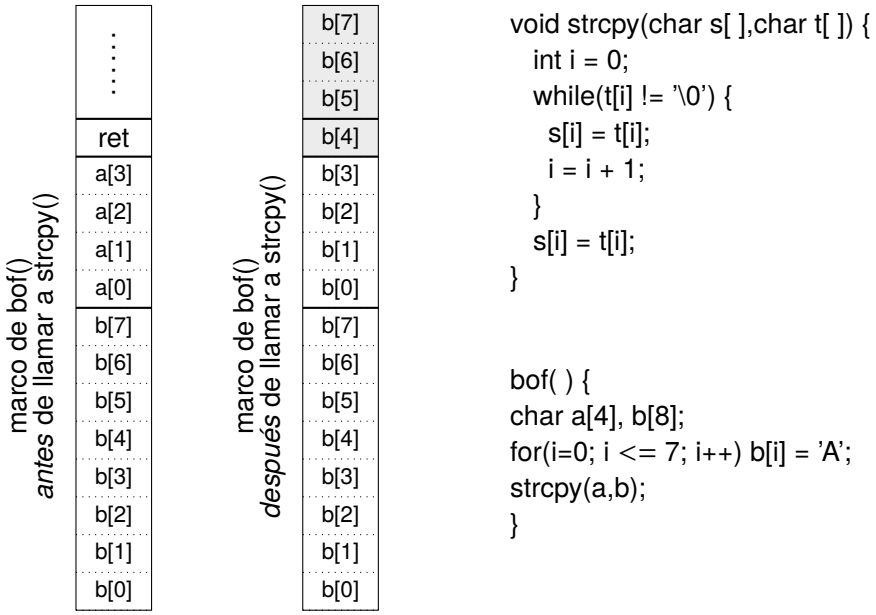


Figura 14: A la izquierda: marco de bof() *antes* y *después* de invocar a strcpy(). A la derecha: código de strcpy() y de bof() que al invocar a strcpy() produce un desbordamiento de arreglo.

arreglo. Es decir, dentro del ciclo, i tomará valores tales que s[i] no es un componente de s. Como i crece, las direcciones de memoria s[i] son cada vez más grandes y en algún momento una de ellas corresponderá a la celda donde está almacenada la dirección de retorno de la función. En ese momento el valor de t[i] sobrescribirá la dirección de retorno. Esto se puede ver en la parte izquierda de la figura donde se muestra el marco de la función bof() *antes* y *después* de invocar a strcpy() (el código de bof() también se muestra en la figura). Esta función declara dos arreglos y le pide a strcpy() que copie uno en el otro. El problema es que el arreglo de origen (b) es más largo que el de destino (a) y por lo tanto la quinta componente de b se escribirá sobre la dirección de retorno. Como b[4] guarda el código ASCII de la letra 'A', el sistema creerá que debe retornar a la dirección 0x41 que es el hexadecimal de 'A'[40]. Pero la dirección 0x41 es muy baja y seguramente estará ocupada por el sistema operativo por lo que habrá un fallo grave y el programa terminará abruptamente. Sin embargo, si el número con el que se sobrescribe la dirección de retorno resulta ser una dirección dentro del espacio de memoria del proceso, no habrá ningún error y el flujo de control continuará en ese punto.

El segundo paso del ataque es, entonces, sobrescribir la dirección de retorno con una dirección dentro del espacio de memoria del proceso. La técnica usada cuando este ataque fue descubierto consiste en darle al programa una entrada parte de la cual es un pequeño programa assembler, llamado *payload* o *shell code* (recordar que el ataque comienza proveyendo una entrada inesperada, ver Ejemplo 22). El nombre *shell code* proviene del hecho de que lo que casi siempre intentan los atacantes es ejecutar un intérprete de comandos (*shell*) que les permita ejecutar cualquier programa. La Figura 15 muestra esquemáticamente cómo sería una entrada

[40]En realidad la dirección de retorno ocupa 4 bytes por lo que se necesitarían 4 componentes del arreglo b para sobrescribirla completamente. Hemos considerado solo un byte para simplicar la explicación.

cadena de entrada: {'A','A',exec,/bin,/bash,exit,23,'\0'}

parte de un marco de pila

	variable local						ret
direcciones	21	22	23	24	25	26	27
contenido	'A'	'A'	exec	/bin	/bash	exit	23

Figura 15: La dirección de retorno apunta al inicio del *shell code* (en gris)

que contiene un *shell code* y su ubicación en un marco de pila. Para simplificar la exposición consideramos que ciertos elementos de la cadena de entrada ocupan un byte cuando en realidad es más complejo y nos hemos tomado algunas licencias en cuanto a los tipos de datos usados. La cadena de entrada contiene dos caracteres 'A' que se los usa para ubicar el *shell code* en el lugar preciso. Luego viene el *shell code* el cual consta de la llamada al sistema exec a la cual se le pasa como parámetro el programa /bin/bash. Luego sigue la instrucción exit para cerrar el programa adecuadamente. El *shell code* se cierra con la dirección de memoria que corresponde al inicio del *shell code*; este valor debe caer justo donde está almacenada la dirección de retorno.

En realidad programar y ubicar el *shell code* es algo más complicado. A continuación mostramos el assembler (x86) correspondiente a un *shell code* que solo ejecuta el intérprete de comandos bash; más abajo explicamos el programa.

```
 1: jmp    0x2a
 2: popl   %esi
 3: movl   %esi, 0x8 (%esi)
 4: movb   $0x0, 0x7 (%esi)
 5: movl   $0x0, 0xc (%esi)
 6: movl   $0xb,%eax
 7: movl   %esi,%ebx
 8: leal    0x8 (%esi),%ecx
 9: leal    0xc (%esi),%edx
10: int    $0x80
11: movl   $0x1,%eax
12: movl   $0x0,%ebx
13: int    $0x80
14: call    -0x2b
15: .string "/bin/bash"
```

donde

1. Salta a la instrucción 14. Este salto se calcula luego de armar el *shell code*.
2. Recupera la dirección de la instrucción posterior al call y la almacena en el registro esi. Esto es así pues en la instrucción 14 hay una instrucción call la cual introduce esa dirección en la pila. O sea que el flujo de control de este *shell code* es: $1 \rightarrow 14 \rightarrow 2 \rightarrow 3 \rightarrow \ldots$. La dirección que esta instrucción almacena en esi es importante porque es una dirección del área de memoria asignada al proceso. De esta forma el *shell code* conoce al menos una dirección de memoria del proceso lo que le permite, en las siguientes instrucciones, usarla como

pivote para operaciones con la memoria (es decir, esas operaciones toman direcciones de memoria relativas a la almacenada en esi).

3. Copia la dirección de la cadena /bin/bash en algún lugar de la memoria.
4. Ídem anterior con el caracter 0x0.
5. Ídem anterior con el caracter 0x0.
6. Se prepara para ejecutar execve (11 o b es el índice en la tabla de llamadas al sistema).
7. Copia la dirección de la dirección de /bin/bash en el registro ebx pues así lo necesita execve.
8. Copia la dirección de la cadena /bin/bash en el registro ecx.
9. Copia la dirección de la cadena 0x0 en el registro edx
10. Ejecuta la interrupción x080 que se utiliza para llamadas al sistema. En este caso está ejecutando execve con parámetro /bin/bash.
11. Las sentencias 11, 12 y 13 equivalen a ejecutar exit(0).
14. Se invoca una supuesta subrutina que comienza en 2.

El paso final es convertir este programa en una cadena de caracteres posiblemente agregándole caracteres normales al inicio para que tenga la longitud justa. Normalmente varios de estos pasos se pueden automatizar con herramientas de *hacking* modernas.

Esta sección debe completarse leyendo [1]

3.1.3. Contramedidas

No existe una bala de plata para evitar los ataques por desbordamiento de arreglo (como en general no existen balas de plata para muchos otros problemas de la Ingeniería de Software). Las defensas o contramedidas para hacer frente a este tipo de ataques requieren un diseño de seguridad en capas. Es decir una serie de defensas en profundidad que abarquen medidas de *prevención*, *detección* y *respuesta*. De esta forma un atacante deberá atravesar todas las barreras para tener éxito en su ataque. En otras palabras se requiere un abordaje que abarque todo el proceso de la ingeniería de seguridad informática.

Las medidas de prevención buscan evitar que existan desbordes de arreglos. Como la posibilidad de desbordar un arreglo es básicamente un error de programación, un grupo importante de medidas de prevención se focalizan en mejorar las prácticas de programación para evitar este tipo de errores. Usualmente esto se llama *programación segura* y en general abarca todas las vulnerabilidades que tienen origen en la programación. Algunas de las contramedidas de esta clase son las siguientes:

- Utilizar lenguajes de programación fuertemente tipados.
- Definir una longitud máxima para todas las entradas del usuario y controlar este máximo en todos los ciclos.
- Controlar que los formatos de datos definidos por los desarrolladores (por ejemplo todas las cadenas de caracteres terminan con `'\0'`) se cumplen siempre.
- Utilizar analizadores estáticos de código (dado que algunos de los desbordamientos de arreglo se pueden detectar estáticamente).

- No utilizar funciones de biblioteca que se sabe no controlan debidamente los desbordamientos de arreglos (por ejemplo en C las funciones gets(), strcpy(), strcat(), etc.).
- Diseñar aplicaciones de forma segura. En general solo unas pocas líneas de código realizan tareas críticas usualmente a nombre de usuarios privilegiados. Lo correcto es que tales líneas formen parte de un núcleo que esté claramente separado del resto de los componentes, que sea pequeño, simple y sea verificado extensivamente.
- Contratar especialistas en seguridad informática que realicen tareas de verificación sobre el código desarrollado a medida para la organización.

Otro grupo de contramedidas de prevención apuntan a configurar la plataforma sobre la cual ejecutarán las aplicaciones de la forma más segura posible. A esto se lo suele llamar *reforzar*[41] *el sistema operativo* o la plataforma en general. El proceso de refuerzo consiste en ajustar uno por uno los parámetros de configuración de la plataforma de manera tal de imponer el principio de seguridad de *mínimo privilegio*. Este principio establece que los usuarios y los procesos deben tener acceso únicamente a lo que necesitan para hacer su trabajo o cumplir con su especificación. La hardenización de una plataforma depende de las aplicaciones y servicios que debe proveer. En la actualidad los sistemas operativos vienen considerablemente hardenizados desde el momento de la instalación (al menos en comparación con el criterio utilizado en décadas pasadas donde se le daba total preeminencia a la funcionalidad y por lo tanto la configuración rara vez restringía algo). Si esto no es suficiente existen *listas de control*[42] elaboradas por equipos de expertos para cada plataforma que indican multitud de parámetros de configuración que se pueden modificar para proveer una plataforma lo más segura posible. Para el caso específico de los desbordamientos de arreglos algunos de los parámetros a considerar son los siguientes:

- Configurar la pila como no ejecutable. En la actualidad esta es una configuración por defecto de los sistemas operativos. Aun así no evita ataques más sofisticados como los basados en *return-oriented programming*. En general requiere hardware que soporte este característica y también puede ser implementada por el compilador.
- Minimizar el número de servicios de red. Muchos de los ataques por desbordamiento de arreglo son realizados desde el exterior sobre la base de servicios de red vulnerables. En muchas instalaciones están activos servicios de red que en realidad no son necesarios o no son necesarios en todas las máquinas. Estos servicios se deben desactivar.
- Asociar los servicios de red a cuentas de usuario no privilegiadas. Si el ataque por desbordamiento de arreglo tiene éxito, se debe intentar minimizar los daños. Una forma de hacerlo es que los servicios de red que sí tienen que ser provistos corran a nombre de usuarios no privilegiados. En lo posible, una vez que se haya autenticado al usuario remoto, el servicio debe crear un proceso a su nombre.
- Minimizar el número de programas que utilicen alguna forma de delegación de permisos.
- Asociar los programas que usan delegación de permisos a cuentas de usuario no privilegiadas.
- Mantener actualizada la plataforma y las aplicaciones teniendo en cuenta los boletines sobre nuevas vulnerabilidades.

[41] 'hardening', en inglés.
[42] 'checklist', en inglés.

- Capacitar a los usuarios para que no abran contenido de origen sospechoso. Por ejemplo si un usuario recibe un PDF de un remitente desconocido o de uno conocido pero que no esperaba que se lo enviara, no debería abrirlo (o en todo caso debería seguir un procedimiento seguro).

Una tercer área donde se pueden implementar defensas de prevención son las de tipo organizacional o de procesos. Estas defensas no son exclusivas de los ataques por desbordamiento de arreglos sino que aplican a toda la seguridad informática. Algunas de estas medidas defensivas son:

- Contar con un área de seguridad informática separada del departamento de sistemas.
- Contar con políticas, estándares y procedimientos para todas las tareas relacionadas con el mantenimiento de la seguridad informática.
- Contar con un plan de capacitación permanente en seguridad informática de todo el personal que usa software y computadoras.
- Contratar expertos externos que evalúen periódicamente el estado de la seguridad informática de la organización.

Las contramedidas para detección y respuesta, en general, no son exclusivas de los ataques por desbordamiento de arreglo sino que aplican también a muchos otros ataques. Las defensas de *detección* apuntan a detectar la presencia de un ataque o un intruso. Entre las de detección podemos mencionar:

- Introducción de *canarios* en la pila. Un canario es un valor en la pila que solo conoce el binario y que cuando es sobrescrito por el desbordamiento es una indicación de ataque.
- Cuando se utilizan técnicas de *return-oriented programming* se pueden implementar algoritmos que detectan binarios que tienen 'muchos' *returns*.
- Instalación de *sistemas de detección de intrusos*[43] los cuales buscan patrones de ataque que permiten levantar alarmas ante presuntos ataques.

Las defensas relativas a la *respuesta* en general pasan, primero, por mantener bitácoras de la actividad del sistema relativa a la seguridad (cf. a auditoría). En efecto, si un ataque no se pudo prevenir ni detectar, lo que queda es poder determinar cómo ocurrió, qué daños produjo, de dónde provino, quién lo efectuó, etc. Para esto son muy valiosas las bitácoras de seguridad pues pueden dar indicios de las IP desde donde se efectuaron conexiones, las computadoras y cuentas de usuario que fueron comprometidas, etc. En segundo lugar, la respuesta incluye desde acciones técnicas de reparación del daño (por ejemplo, restaurar respaldos o cortar una conexión de red si el ataque aun no cesó) hasta acciones legales (por ejemplo, denunciar el ataque a las autoridades), pasando por acciones de estudio que nos permitan mejorar la seguridad en el futuro. Una buena ingeniería de seguridad informática debe contar con planes de respuesta para cada tipo de ataque razonable, confeccionados con anticipación.

3.1.4. Conclusiones sobre el ataque por desbordamiento de arreglos

Este tipo de ataques necesita de un error en un programa. Resaltamos esto porque los ataques por medio de caballos de Troya no requieren ningún error en ningún programa. Lo

[43] 'intrusion detection system', en inglés.

que sí necesita un ataque con caballo de Troya es que el usuario ejecute un programa, mientras que con el desbordamiento de arreglos, en el peor de los casos, se necesita que el usuario abra un archivo de datos (por ejemplo un PDF). Dada la facilidad con que muchos usuarios (en particular usuarios avanzados, desarrolladores y administradores de sistemas) instalan y usan software descargado de Internet, parecería que no es tan difícil lograr que un usuario ejecute un programa más o menos desconocido. Por el contrario, si bien es cierto que en general el software está repleto de errores, encontrar un desbordamiento de arreglos explotable no es cosa de todos los días, y aun cuando lo encontramos requiere un trabajo nada trivial montar el ataque (más aun si es preciso usar técnicas como ROP o ROP sin *returns*).

Aun así, los atacantes tipo *hackers* parecerían estar mucho más interesados en encontrar desbordes de arreglos o ataques de una complejidad técnica igual o superior que en infiltrar caballos de Troya en computadoras. Por ejemplo, ¿por qué la Agencia Nacional de Seguridad (NSA) de los EE.UU. hace grandes esfuerzos por romper complejos sistemas criptográficos cuando podría convertir cualquier versión de Linux y Windows (o sus programas más usados) en caballos de Troya? ¿Serán ya caballos de Troya? El gobierno de los EE.UU., ¿no habrá solicitado ya la colaboración de empresas esencialmente norteamericanas tales como Intel, Microsoft, Oracle, IBM, Google, etc. para que sus productos se comporten como caballos de Troya si el gobierno así lo necesita? ¿O tal vez habrá infiltrado esas empresas con programadores que incluyeron unas pocas líneas de código entre los millones de sus principales productos de tal manera que ahora son caballos de Troya al servicio del gobierno de ese país? ¿No le daría eso acceso a virtualmente todas las computadoras del mundo? Probablemente hayan hecho esto [12, 19, 31] y también intenten romper criptografía, encontrar desbordes de arreglo, etc.

Más aun, si el sistema operativo implementa alguna forma de no interferencia, un ataque por medio de desbordamiento de arreglos sería totalmente inofensivo pues las protecciones implementadas por el sistema asumen que todo el software fuera de la TCB puede ser desarrollado por un enemigo letal. En este caso el único desborde de arreglo peligroso sería aquel presente en uno de los componentes de la TCB pues destruiría las protecciones del sistema. Sin embargo, el software de la TCB se espera haya sido sometido a verificación formal por lo cual no debería existir la posibilidad de desbordar arreglos.

3.2. Inyección de sentencias SQL

Un ataque por *inyección de sentencias SQL*[44] (ISQL) consiste en la inserción de sentencias SQL a través de los datos de entrada de una aplicación (cliente). Un ataque exitoso puede leer datos confidenciales, modificar datos, ejecutar tareas administrativas sobre el DBMS, recuperar archivos del sistema de archivos donde está instalado el DBMS y en algunos casos ejecutar comandos del sistema operativo.

Mediante estos ataques se puede falsear identidades, causar problemas de repudiación[45] (tales como eliminación de transacciones o cambios en los saldos de cuentas corporativas), divulgar todos los datos del sistema de bases de datos y tomar el control del DBMS.

En resumen un ataque por ISQL puede ocasionar problemas de:

- Confidencialidad: ya que permite divulgar datos privados almacenados en el DBMS.

[44] 'SQL injection', en inglés.

[45] En este contexto 'repudio' o 'repudiación' se refiere a la posibilidad de negar la autenticidad de una transacción o de datos en general.

- Autenticación: puesto que en ocasiones las credenciales de autenticación se guardan en el mismo DBMS y se acceden vía SQL.
- Autorización: nuevamente en muchos sistemas los datos de control de acceso (permisos sobre las tablas del sistema) se guardan en el mismo DBMS.
- Integridad: puesto que las sentencias SQL permiten modificar datos.

La ISQL es muy común en aplicaciones implementadas en PHP o ASP (Microsoft) debido a la presencia de interfaces antiguas. Los ataques por ISQL son menos factibles en las plataformas J2EE y ASP.NET por cuestiones de implementación. Este tipo de vulnerabilidad es muy común en aplicaciones web que procesan grandes volúmenes de datos.

Los errores de ISQL se dan cuando:

1. Hay ingreso de datos provenientes de una fuente no confiable.
2. Los datos son usados para construir dinámicamente sentencias SQL.

El ataque consiste en insertar un meta-caracter (i.e. un caracter de control o símbolo reservado del lenguaje) en una cadena de entrada que luego se usa para crear una sentencia SQL que no estaba prevista por los desarrolladores. Estas vulnerabilidades se producen debido a que SQL no distingue entre caracteres de control y datos.

Ejemplo 23 (Inyección de SQL)**.** *Consideremos la sentencia SQL*

select password from authors where firstname=fn and lastname=ln

donde suponemos que fn *y* ln *toman valores directamente de la entrada del usuario. Ahora digamos que el usuario ingresa:*

fn = evil'ex
ln = newman

donde se debe prestar atención al apóstrofo entre evil *y* ex*.*

En este caso la sentencia SQL que se obtiene cuando fn *y* ln *se sustituyen por sus valores es:*

select password from authors where firstname='evil'ex' and lastname='newman'

donde nuevamente queremos resaltar el apóstrofo entre evil *y* ex*. Cuando esta sentencia es enviada al DBMS este responde con un error:*

Incorrect syntax near il' as the database tried to execute evil

el cual se debe a que hay un apóstrofo no balanceado (es decir hay 5 y no 4).

Como en el caso de los ataques por desbordamiento de arreglo, si la cadena de entrada se manipula correctamente es posible ejecutar sentencias SQL no previstas por los desarrolladores. Para ver un ejemplo, digamos que el atacante ingresa:

fn = evil' or 1=1 #
ln = newman

que genera la sentencia SQL:

select password from authors where firstname='evil' or 1=1 # ' and lastname='newman'

en cuyo caso el DBMS solo considera y ejecuta:

select password from authors where firstname='evil' or 1=1

dado que el símbolo # es interpretado como el inicio de un comentario SQL. Entonces, esta sentencia devolverá todos los valores del campo password *ya que la condición* firstname='evil' or 1=1 *se satisface trivialmente para todos los registros de la tabla.*

De igual forma se pueden ejecutar otras sentencias SQL; por ejemplo, ingresando:

```
fn = evil'; delete from algunatabla #
ln = newman
```

el DBMS ejecuta:

```
select password from authors where firstname='evil'; delete from algunatabla
```

3.2.1. Contramedidas

Muchas de las contramedidas de carácter general estudiadas para el ataque por desbordamiento de arreglos, también aplican a ISQL. Por otro lado, hay al menos dos mecanismos de defensa preventivos dentro del área de programación (que es la que nos ocupa en esta unidad) que son exclusivos para ISQL.

La primera defensa que veremos se denomina *sentencias preprogramadas*[46] o *consultas parametrizadas*[47]. Este estilo de programación SQL debería ser el primero que deberían aplicar los desarrolladores de este tipo de aplicaciones. Las sentencias preprogramadas obligan a los programadores a definir inicialmente todo el código SQL y luego pasarle los parámetros. Este estilo de programación le permite al DBMS distinguir código de datos. Las sentencias preprogramadas impiden que un atacante cambie el significado esperado de las sentencias SQL.

Cada lenguaje de programación define su propio entorno para preprogramar sentencias:

- En Java se debe usar PreparedStatement()
- En .NET se usan consultas parametrizadas como SqlCommand() o OleDbCommand()
- En PHP se debe usar PHP Data Objects (PDO) con consultas parametrizadas fuertemente tipadas (vía bindParam()).

En situaciones poco frecuentes las sentencias preprogramadas pueden afectar el desempeño de la aplicación. Si esto ocurre las alternativas son:

- Validar toda la entrada del usuario
- Escapar toda la entrada del usuario utilizando una subrutina de escape de caracteres provista por el fabricante del DBMS.
- Usar *procedimientos almacenados*[48] (ver más abajo)

Ejemplo 24 (Sentencias preprogramadas en Java)**.** *El código Java que sigue muestra el uso de las sentencias preprogramadas.*

```
String fn = request.getParameter('firstname');
String ln = request.getParameter('lastname');
String query = "select password from authors where firstname = ? and lastname = ?";
PreparedStatement ps = connection.prepareStatement(query);
```

[46] 'prepared statements', en inglés.
[47] 'parameterized queries', en inglés.
[48] 'store procedure', en inglés.

```
ps.setString(1,fn);
ps.setString(2,ln);
try {ResultSet res = ps.executeQuery();}
```

Los desarrolladores tienden a preferir las sentencias preprogramadas porque todo el código SQL permanece en la aplicación y la mantiene relativamente independiente del DBMS. Sin embargo, los *procedimientos almacenados* permiten guardar todo el código SQL en el mismo DBMS lo que tiene ventajas desde la seguridad pero también desde otros aspectos (como el desempeño).

Entonces el segundo mecanismo de defensa que veremos son los procedimientos almacenados. Estos tiene el mismo efecto que las sentencias preprogramadas si se los implementa de manera segura (lo que significa no usar SQL dinámico). Como con las sentencias preprogramadas, primero se deben definir las sentencias SQL y luego se les pasan los parámetros. Entonces, dentro del DBMS se programa una subrutina (el procedimiento almacenado) que tiene un identificador (nombre) y parámetros. Luego, desde la aplicación, se invoca esta subrutina proveyéndole los parámetros. Obviamente esta solución es muy dependiente del DBMS concreto pues cada uno define su propia interfaz y lenguaje para los procedimientos almacenados.

Ejemplo 25 (Procedimiento almacenado en Java)**.** *El código Java que sigue muestra el uso de los procedimientos almacenados.*

```
String fn = request.getParameter('firstname');
String ln = request.getParameter('lastname');
try {
    CallableStatement cs = connection.prepareCall("{call sp_getPassword(?,?)}");
    cs.setString(1,fn);
    cs.setString(2,ln);
    ResultSet res = cs.executeQuery();
}
```

donde todo lo que está entre comillas en la primera sentencia del try *es dependiente del DBMS (es decir, por ejemplo, otro DBMS podría usar* invoke *en lugar de* call*); y* sp_getPassword() *es el nombre del procedimiento almacenado.*

Esta sección debe completarse leyendo [25]

3.2.2. Conclusiones sobre el ataque por ISQL

Como en el caso del desbordamiento de arreglos, los ataques por ISQL necesitan de un error de programación (en general no funcional). Por lo tanto aplican consideraciones similares al otro ataque. Por otro lado, es conveniente resaltar que dado:

- La cantidad de aplicaciones web existentes (a los fines prácticos infinita);
- Que a través de ellas muchas veces se puede llegar hasta el corazón de los sistemas de una organización;
- La complejidad de muchas de estas aplicaciones y la frecuencia de cambios a que se ven sometidas; y
- La facilidad con que se detecta y explota esta vulnerabilidad

es muy difícil evitar estos ataques y cuando ocurren suelen tener consecuencias muy graves para las organizaciones.

3.3. Cuando *security* es *safety*: proof-carrying code

La técnica de verificación de software conocida como *proof-carrying code* (PCC) fue inventada por George C. Necula y Peter Lee[49] en 1997 [23]. PCC apunta sobre todo a verificar propiedades de seguridad de tipos de programas. Por consiguiente, PCC es una técnica orginalmente pensada para un problema de *safety* pero como ciertos problemas de seguridad, por ejemplo el desbordamiento de arreglos, son también problemas de *safety* (ver Sección 2.6), fue muy estudiada por la comunidad de seguridad informática. Lo que sigue corresponde a un resumen del artículo publicado por Necula en POPL 97 [22].

El problema que quiere resolver Necula es el siguiente:

- Tenemos un programa F que va a invocar rutinas de las cuales no tenemos el código fuente.
- En este caso se pierde la garantía de la seguridad de tipos de F pues no podemos saber si los binarios invocados respetan el sistema de tipos del lenguaje de programación de F.

Entonces, la pregunta que se hace Necula es: ¿hay alguna forma de garantizar que el binario respeta el sistema de tipos? Es importante notar que a Necula no le preocupa la especificación funcional de F si no la garantía de que se preserva su sistema de tipos.

La respuesta que encuentra es que evidentemente hay que trabajar a nivel del código assembler. O sea que de alguna forma debemos ser capaces de verificar los tipos a nivel del código assembler. El siguiente paso es generalizar esta situación:

- Tenemos un consumidor de código C; es decir un programador que usará binarios provenientes de orígenes desconocidos o no confiables. C es quien programa F y por lo tanto tiene su código fuente.
- Tenemos un productor de código P que envía a C sus binarios.
- C establece las propiedades de tipos que el código programado por P debe verificar. A estas propiedades se las llama *política de seguridad de tipos*. La política de seguridad de tipos es un conjunto de invariantes de tipos sobre los tipos de datos que usa F. Estos invariantes se dan usando alguna lógica relativamente simple.
- P debe suministrar el código assembler junto a una demostración de que el assembler verifica la política de tipos definida por C. Es decir es como si proveyera el *script* de prueba de algún demostrador de teoremas.
- Cuando C recibe el código, ejecuta la demostración en relación al código binario recibido. Es decir, comprueba que esa demostración corresponde al binario recibido.

Aunque la comprobación de la demostración que hace C es automática y muy eficiente, la prueba que debe hacer P no siempre es automática y debe efectuarse a nivel del assembler, de aquí que PCC tenga un uso práctico limitado.

[49] Actualmente Director de Microsoft Research.

Supongamos entonces que tenemos un lenguaje de programación (de alto nivel), $\mathcal{P}$, que tiene enteros, pares ordenados, sumas disjuntas o uniones y listas como tipos de datos. La gramática del sistema de tipos de $\mathcal{P}$ es:

$$\tau ::= \mathsf{Int} \mid (\tau, \tau) \mid (\textit{Constructor}\ \mathsf{of}\ \tau;\ \textit{Constructor}\ \mathsf{of}\ \tau) \mid \mathsf{List}(\tau)$$

donde $(\cdot,\cdot)$ simboliza un par ordenado cuyas componentes pueden ser de distintos tipos; y $(\cdot;\cdot)$ simboliza una suma disjunta o unión donde una expresión de ese tipo o bien tiene el tipo de la izquierda o bien el de la derecha. Cuando se define un tipo unión se deben dar los constructores correspondientes.

Ejemplo 26 (Declaración de uniones)**.** *La siguiente es una declaración de un tipo unión:*

```
datatype T = (int of Int; par of (Int,Int))
```

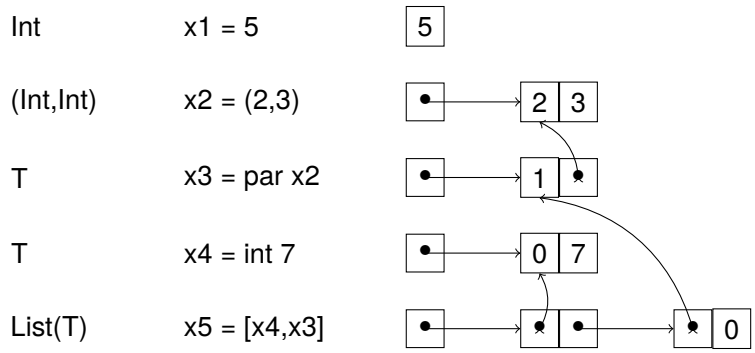
donde int *y* par *son los constructores de la unión* T*. Los elementos de* T *pueden ser o bien enteros o bien pares ordenados de enteros. De esta forma los elementos de tipo* T *siempre tienen la forma* int *x o* par *x (por ejemplo* int *7 o* par $(6,8)$*). Entonces, por caso, una lista de tipo* T *tiene la forma [int 2,par (3,-1),int 9,int -4].*

Cuando un programa escrito en $\mathcal{P}$ se compila, cada tipo se implementa a nivel de la máquina mediante *palabras* (de memoria[50]) y punteros. Como C debe dar la política de tipos a nivel del código máquina vamos a mostrar cómo se representa a ese nivel cada tipo τ.

- Enteros. Cada entero se representa con una palabra.
- Pares ordenados. Cada par ordenado se representa como un puntero a dos palabras contiguas.
- Uniones. Cada valor de un tipo unión se representa como un puntero a dos palabras contiguas donde la primera almacena el constructor usado (0 para el izquierdo y 1 para el derecho) y la segunda almacena el valor del tipo correspondiente (el cual obviamente depende del constructor utilizado).
- Listas. La lista vacía se representa con 0. Una lista no vacía se representa con un puntero a un par de palabras contiguas (i.e. cada nodo de la lista) donde la primera almacena el primer valor de la lista y la segunda un puntero al siguiente nodo.

Ejemplo 27 (Representación binaria de los tipos de $\mathcal{P}$)**.** *A continuación mostramos varios valores de diferentes tipos de $\mathcal{P}$ y su representación binaria en forma gráfica. Asumimos que contamos con la declaración de la unión* T *del Ejemplo 26. Cada cuadrado representa una palabra; el cuadrado de la columna de la izquierda representa la variable* x *correspondiente.*

[50] Usamos el término 'palabra' (*word*) de memoria como una secuencia contigua de posiciones de memoria que almacenan un valor indivisible. Por ejemplo, en una arquitectura dada, un entero se almacena en una palabra de cuatro bytes. Suponemos que la primera dirección de memoria identifica la palabra. Entonces si e es una dirección de memoria al inicio de una palabra, $e+4$ es la dirección de memoria de la palabra siguiente si la longitud de palabra es de 4 bytes.



En la tercer línea, la primera de las dos palabras contiguas vale 1 porque x3 es un par ordenado de tipo T, mientras que en la siguiente línea vale 0 porque x4 es un entero del mismo tipo.

El consumidor de código formaliza estas representaciones con *proposiciones de tipos*[51] de la forma:

$$m \vdash e : \tau$$

donde m es el estado de la memoria, e es una expresión y τ es un tipo. Estas proposiciones se leen "en la memoria m la expresión e tiene tipo τ". En general e refiere a una palabra de la memoria (más precisamente a la dirección de la primera celda de la palabra). Consideramos que la memoria es una función que toma una dirección de memoria y devuelve el contenido correspondiente. A este fin definimos el tipo Addr, de las direcciones de memoria, como un subconjunto de Int.

Las proposiciones de tipos se dan mediante reglas de inferencia. A continuación damos solo algunas de ellas (comparar con la representación del Ejemplo 27). Si un par ordenado tiene tipo (τ_1, τ_2), la primera componente tiene tipo τ_1, la segunda τ_2 y, en una arquitectura que usa palabras de 4 bytes, la segunda componente está a 4 bytes de distancia de la primera:

$$\frac{m \vdash e : (\tau_1, \tau_2)}{m \vdash e : \mathsf{Addr} \qquad m \vdash e+4 : \mathsf{Addr} \qquad m \vdash m(e) : \tau_1 \qquad m \vdash m(e+4) : \tau_2} \tag{1}$$

Algo similar ocurre con las uniones de tipo $(\tau_1;\ \tau_2)$ solo que en este caso la expresión o tiene un tipo o tiene el otro, dependiendo del valor del constructor:

$$\frac{m \vdash e : (\tau_1;\ \tau_2)}{m \vdash e : \mathsf{Addr} \quad m \vdash e+4 : \mathsf{Addr} \quad m(e) = 0 \Rightarrow m \vdash m(e+4) : \tau_1 \quad m(e) \neq 0 \Rightarrow m \vdash m(e+4) : \tau_2} \tag{2}$$

La regla para las listas no vacías de tipo τ se da en forma similar.

$$\frac{m \vdash e : \mathsf{List}(\tau) \qquad e \neq 0}{m \vdash e : \mathsf{Addr} \quad m \vdash e+4 : \mathsf{Addr} \quad m \vdash m(e) : \tau \quad m \vdash m(e+4) : \mathsf{List}(\tau)} \tag{3}$$

Por último, una regla que establece que la la suma de enteros tiene tipo entero:

$$\frac{m \vdash e_1 : \mathsf{Int} \qquad m \vdash e_2 : \mathsf{Int}}{m \vdash e_1 + e_2 : \mathsf{Int}} \tag{4}$$

[51] 'type judgement', en inglés.

Estas reglas le permiten al productor de código anotar su código binario de forma tal que, usando una adaptación de la lógica de Hoare, se puede verificar que si vale la pre-condición (del programa) entonces vale la post-condición. Tanto la pre-condición como la post-condición son, precisamente, proposiciones de tipos ($m \vdash e : \tau$).

Entonces, veamos un ejemplo de cómo el productor de código debe anotar su código de forma tal que luego pueda hacer la demostración de que el código verifica la política de tipos del consumidor. Supongamos que el productor ha escrito una función, sum, que suma los elementos de una lista de tipo List(T) donde T es el tipo definido en el Ejemplo 26. Por ejemplo sum([int 2,par (3,-1),int 9,int -4]) debería sumar 2 + 3 - 1 + 9 - 4, o sea el resultado debería ser 9. El productor compila el programa y produce el binario siguiente[52].

```
 1        pre   m ⊢ r0 : List(T)                          ; r0 es la lista a sumar
 2        mov   r1 ,0                                     ; r1 es el acumulador, se inicializa a 0
 3  :ini  inv   m ⊢ r0 : List(T) ∧ m ⊢ r1 : Int           ; invariante de ciclo
 4        beq   r0 ,:fin                                  ; lista vacía? Si true a :fin
 5        ld    r2 ,0(r0)                                 ; cargar primer elemento
 6        ld    r0 ,4(r0)                                 ; cargar el resto
 7        ld    r3 ,0(r2)                                 ; cargar constructor
 8        ld    r2 ,4(r2)                                 ; cargar dato
 9        beq   r3 ,:sum                                  ; es un int? Si true a :sum
10        ld    r3 ,0(r2)                                 ; cargar x
11        ld    r2 ,4(r2)                                 ; cargar y
12        add   r2,r3,r2                                  ; x + y
13  :sum  add   r1,r2,r1                                  ; sumar
14        br    :ini                                      ; reiniciar ciclo
15  :fin  mov   r0,r1                                     ; copiar resultado en r0
16        post  m ⊢ r0 : Int                              ; post—condición
17        ret                                             ; resultado en r0
```

Las instrucciones **pre**, **inv** y **post** son en realidad comentarios insertados al código binario con el fin de proveer las proposiciones de tipos necesarias para realizar la verificación; es decir, estas líneas no se ejecutan. Se asume que hay solo una instrucción **pre** al inicio del código; una instrucción **post** antes de cada instrucción **ret**; y tantas instrucciones **inv** como sean necesarias (ver más adelante). No es necesario que comprendan el código en detalle. Lo importante es que: $\mathbf{r}_i$ son registros del micro; en $\mathbf{r}_0$ se copia la lista a sumar; $\mathbf{r}_1$ es el acumulador para llevar la suma; en la primera línea está la pre-condición[53] de la función; de la línea 4 a la 14 se define un ciclo (*loop*) cuyo invariante se da en la línea 3 y donde se itera sobre la lista para ir sumando sus elementos; :ini, :sum y :fin son etiquetas para realizar saltos; en la línea 16 está la post-condición de la función; y el resultado de la suma se devuelve en $\mathbf{r}_0$.

Entonces, sum asume que al inicio en $\mathbf{r}_0$ hay algo de tipo List(T) y en ese caso garantiza que al finalizar, en $\mathbf{r}_0$ habrá algo de tipo Int. El objetivo es que el consumidor de código pueda comprobar que tal afirmación se verifica en el código que recibe.

Con este fin representamos el programa assembler con un vector Π tal que en casa posición hay una instrucción del programa. O sea que al programa sum le corresponde un Π de 17

[52] Corresponde a arquitectura DEC Alpha. Ejemplo tomado de [22].

[53] Recordar que en este contexto pre-condición, invariante y post-condición refieren a proposiciones de tipos y no a la especificación funcional de la función.

posiciones. O sea que Π incluye las pseudo-instrucciones **pre**, **inv** o **post**. Notamos con Π_i cada una de las posiciones de Π. Asociado a Π hay un *vector de condiciones de verificación*, VC. Cada VC_i contiene una proposición que se obtiene al aplicar una adaptación de la lógica de Hoare al código binario. Concretamente, VC se calcula de la siguiente forma:

$$VC_i = \begin{cases} [n/\mathbf{r}_s]VC_{i+1} & \textbf{si } \Pi_i = \textbf{mov } \mathbf{r}_s, n \\ [\mathbf{r}_s + op/\mathbf{r}_d]VC_{i+1} & \textbf{si } \Pi_i = \textbf{add } \mathbf{r}_s, op, \mathbf{r}_d \\ m \vdash \mathbf{r}_s + n : \mathsf{Addr} \wedge [m(\mathbf{r}_s + n)/\mathbf{r}_d]VC_{i+1} & \textbf{si } \Pi_i = \textbf{ld } \mathbf{r}_d, n(\mathbf{r}_s) \\ (\mathbf{r}_s = 0 \Rightarrow VC_{i+n+1}) \wedge (\mathbf{r}_s \neq 0 \Rightarrow VC_{i+1}) & \textbf{si } \Pi_i = \textbf{beq } \mathbf{r}_s, n \\ VC_{i+n+1} & \textbf{si } \Pi_i = \textbf{br } n \\ \mathcal{A} & \textbf{si } \Pi_i = \textbf{post } \mathcal{A} \wedge \Pi_{i+1} = \textbf{ret} \\ \mathcal{A} & \textbf{si } \Pi_i = \textbf{inv } \mathcal{A} \end{cases}$$

donde $[e/\mathbf{r}_i]P$ indica que en P se sustituye $\mathbf{r}_i$ por e.

Si miran con atención van a ver que la definición de VC está dada sustancialmente por una formalización del assembler siguiendo la lógica de Hoare. Si bien esta definición de VC alcanza para comprender el ejemplo que venimos desarrollando, en el caso real sería necesario extenderla a más instrucciones assembler.

Ahora sea Inv el conjunto de números de línea donde hay una instrucción **pre** o **inv**, y sea Inv_i el predicado de tipos correspondiente a la instrucción de la línea la i. Entonces, se define la condición de verificación para todo el código de la siguiente forma:

$$VC(\Pi, Inv) = \forall \mathbf{r}_j : \bigwedge_{i \in Inv} Inv_i \Rightarrow VC_{i+1}$$

En el ejemplo de sum, tenemos $Inv = \{1, 3\}$. En consecuencia $VC(\mathsf{sum}, Inv)$ tiene dos conjunciones:

$$VC(\Pi, Inv) = \forall \mathbf{r}_j : (Inv_1 \Rightarrow VC_2) \wedge (Inv_3 \Rightarrow VC_4)$$

donde la que corresponde a $Inv_1 \Rightarrow VC_2$ es:

$$m \vdash \mathbf{r}_0 : \mathsf{List(T)} \Rightarrow m \vdash \mathbf{r}_0 : \mathsf{List(T)} \wedge m \vdash 0 : \mathsf{Int}$$

puesto que Inv_1 es $m \vdash \mathbf{r}_0 : \mathsf{List(T)}$ y VC_2 se calcula aplicando la regla para **mov** la cual indica que se debe sustituir $\mathbf{r}_1$ por 0 en VC_3 que es la proposición de la instrucción **inv**, o sea $m \vdash \mathbf{r}_0 : \mathsf{List(T)} \wedge m \vdash \mathbf{r}_1 : \mathsf{Int}$.

El cálculo de $Inv_3 \Rightarrow VC_4$ es más complejo pues requiere derivar todas las condiciones hasta el final de programa. Esencialmente esta condición establece que se preserva el invariante de ciclo y que esta implica la post-condición cuando el ciclo termina. La dejamos como ejercicio.

Es importante observar que el productor de código debe proveer Π, el cual incluye la especificación y los invariantes de ciclo, y una demostración de que vale $VC(\Pi, Inv)$. La dificultad en usar PCC yace tanto en proveer las especificaciones y los invariantes de ciclo, como en proveer la demostración. Para muchas políticas de seguridad de tipos interesantes, es posible demostrar $VC(\Pi, Inv)$ automáticamente si se han dado las especificaciones e invariantes de ciclo. En general, esto se consigue implementando *compiladores de certificación*[54], es decir compiladores

[54] 'certifying compiler', en inglés.

que generan el código objeto junto con una prueba de que este verifica ciertas propiedades. Luego, el consumidor solo debe verificar la prueba respecto del código, lo que es siempre automático y eficiente. Precisamente la ventaja de PCC, frente a otras soluciones, es que la mayor parte del trabajo se hace *off-line*. Otra de las ventajas de PCC es que el consumidor o bien detecta modificaciones en el código y en la prueba que tornan el código peligroso, o bien no las detecta, aunque las haya habido, pero entonces el código no es peligroso.

Esta sección debe completarse leyendo [22]

3.4. Cuando *security* es *safety*: un sistema de tipos para flujo de información seguro

En esta sección vamos a ver un sistema de tipos definido para una cierta clase *restringida* de programas que garantiza que cualquier programa correctamente tipado tiene un flujo de información seguro (es decir, no hay filtración de información de nivel H hacia variables de nivel L). Dicho de otro modo, vamos a resolver el problema de flujo de información planteado por Denning [10] pero esta vez usando un sistema de tipos. Claramente, si podemos tratar el flujo seguro de información como un problema de tipado estamos ante una situación donde *security* es *safety*. En este caso la reducción de un problema de *security* a uno de *safety* es posible porque, como dijimos más arriba, consideramos una clase restringida de programas. Concretamente, el trabajo que vamos a estudiar es el de Volpano, Smith e Irvine [32]. Posteriormente a este artículo se han publicado muchos otros siguiendo más o menos la misma línea de trabajo.

El lenguaje de programación sobre el cual se define el sistema de tipos consta de: asignación (:=), sentencia condicional (if - then - else - fi), sentencia iterativa (while - do - done), concatenación de sentencias (;), expresiones aritméticas (+, −), expresiones booleanas (=, <) y variables y constantes enteras[55]. El 0 se considera como el booleano *false* mientras que el 1 corresponde a *true*.

Cada variable x de un programa tiene una clase de acceso que se denota con $\bar{x}$[56]. La clase de programas para la cual se define el sistema de tipos es aquella en la que la clase de acceso de cualquier variable de un programa se puede determinar en tiempo de compilación (o sea es estática) y en consecuencia no puede alterarse durante la ejecución del programa. Claramente esto deja afuera una clase enorme de programas importantes. Por ejemplo, esta técnica no puede usarse para un programa donde el usuario ingresa el nombre de un archivo a abrir puesto que la clase de acceso de la variable donde se leerá el contenido del archivo se puede determinar una vez leída la entrada del usuario (o sea en tiempo de ejecución). Esta restricción es clave para poder convertir un problema de *security* en uno de *safety*.

Los tipos del sistema de tipos se estratifican en dos niveles. En un nivel están los *tipos de datos*, denotados por τ, los cuales corresponden a las clases de acceso. Es decir hay un tipo de dato por cada clase de acceso. Por ejemplo, si una clase de acceso es $(secret, \{NATO, F35\})$ entonces habrá un tipo cuyo nombre es $(secret, \{NATO, F35\})$. En el otro nivel están los tipos de las sentencias y expresiones del lenguaje, llamados *tipos frase* y denotados por ρ. Los tipos frase incluyen a los tipos de datos que son los que se asignan a expresiones, los tipos de las variables de la forma τ var, y los tipos de las sentencias que tienen la forma τ cmd, llamados *tipos comando*. Es decir, los modificadores var y cmd se usan para saber si el tipo corresponde a una variable

[55] El lenguaje definido por Volpano et al. incluye algunas otras construcciones pero no es necesario tenerlas en cuenta en este curso.

[56] En el artículo usan $\underline{x}$.

$$\gamma \vdash n : \tau \qquad \text{(INT)}$$

$$\gamma \vdash x : \tau \ \mathsf{var}, \text{ if } \gamma(x) = \tau \ \mathsf{var} \qquad \text{(VAR)}$$

$$\frac{\gamma \vdash e : \tau \qquad \gamma \vdash e' : \tau}{\gamma \vdash e + e' : \tau} \qquad \text{(ARITH)}$$

$$\frac{\gamma \vdash e : \tau \ \mathsf{var} \qquad \gamma \vdash e' : \tau}{\gamma \vdash e := e' : \tau \ \mathsf{cmd}} \qquad \text{(ASSIGN)}$$

$$\frac{\gamma \vdash c : \tau \ \mathsf{cmd} \qquad \gamma \vdash c' : \tau \ \mathsf{cmd}}{\gamma \vdash c \ ; \ c' : \tau \ \mathsf{cmd}} \qquad \text{(COMPOSE)}$$

$$\frac{\gamma \vdash e : \tau \qquad \gamma \vdash c : \tau \ \mathsf{cmd} \qquad \gamma \vdash c' : \tau \ \mathsf{cmd}}{\gamma \vdash \mathsf{if}\ e\ \mathsf{then}\ c\ \mathsf{else}\ c'\ \mathsf{fi} : \tau \ \mathsf{cmd}} \qquad \text{(IF)}$$

$$\frac{\gamma \vdash e : \tau \qquad \gamma \vdash c : \tau \ \mathsf{cmd}}{\gamma \vdash \mathsf{while}\ e\ \mathsf{do}\ c\ \mathsf{done} : \tau \ \mathsf{cmd}} \qquad \text{(WHILE)}$$

Figura 16: Reglas de tipado para expresiones y sentencias

o a una sentencia (cuando se trata de una expresión el tipo se escribe sin modificador). Una variable cuyo tipo es τ var guarda información cuya clase de acceso es τ o inferior. A su vez una sentencia c tiene tipo τ cmd solo si está garantizado que cualquier asignación dentro de c se hace a una variable cuya clase de acceso es τ o superior. Esta es una propiedad de *confinamiento* necesaria para controlar los flujos de información implícitos. Finalmente, se extiende la relación de dominación entre clases de acceso ($\preceq$) a una relación de subtipado denotada con $\subseteq$[57].

Las reglas de tipado para las expresiones y sentencias del lenguaje se dan en la Figura 16. Como puede apreciarse las reglas de tipado tienen la misma forma que las proposiciones de tipos de PCC solo que varía su interpretación. En este caso $\gamma \vdash p : \rho$ significa que la sentencia o expresión p tiene tipo ρ asumiendo que γ da los tipos para las variables de p. γ se denomina *contexto de tipado* y es una función tal que si x es una variable $\gamma(x) = \bar{x}$ var, o sea $\gamma(x)$ devuelve el tipo frase de x. Las reglas que definen la lógica de subtipado se dan en la Figura 17. La regla (CMD$^-$) establece que la relación de subtipado es *antimonotónica* para los tipos comando—si $\tau \subseteq \tau'$ entonces τ' cmd $\subseteq \tau$ cmd. Como es usual, existe una regla de coerción de tipos, (SUBTYPE), la cual permite tipar una expresión o sentencia de tipo ρ con un tipo ρ' siempre y cuando se verifique $\rho \subseteq \rho'$.

Estas reglas garantizan la seguridad del flujo (explícito o implícito) de la información dentro de un programa[58]. Consideremos por ejemplo la regla (ASSIGN). Esta regla dice que para que el flujo explícito desde e' a e sea seguro, e y e' tienen que tener la misma clase de acceso lo que se impone al tener τ en ambas hipótesis. De todas formas, un flujo hacia arriba desde e' hacia e es aun posible aplicando la regla (SUBTYPE). En efecto, si $e : H$ var y $e' : L$ entonces el tipo de e' puede ser forzado a H en cuyo caso la regla (ASSIGN) puede ser aplicada aun con $\tau = H$.

[57] O sea que si τ y τ' son tipos de datos (es decir son clases de acceso) entonces $\tau \preceq \tau' \Leftrightarrow \tau \subseteq \tau'$.

[58] En el artículo de Volpano et al. pueden encontrar la demostración de esta afirmación; no la veremos en clase.

$$\frac{\tau \preceq \tau'}{\vdash \tau \subseteq \tau'} \quad \text{(BASE)}$$

$$\rho \subseteq \rho \quad \text{(REFLEX)}$$

$$\frac{\vdash \rho \subseteq \rho' \qquad \vdash \rho' \subseteq \rho''}{\vdash \rho \subseteq \rho''} \quad \text{(TRANS)}$$

$$\frac{\tau \subseteq \tau'}{\tau' \ \mathsf{cmd} \subseteq \tau \ \mathsf{cmd}} \quad (\text{CMD}^-)$$

$$\frac{\gamma \vdash p : \rho \qquad \vdash \rho \subseteq \rho'}{\gamma \vdash p : \rho'} \quad \text{(SUBTYPE)}$$

Figura 17: Reglas de subtipado

De forma similar, la regla (IF) establece que el tipo del condicional es τ cmd con el fin de controlar los flujos implícitos. Por ejemplo, supongamos que x es 0 o 1 y consideremos el programa:

$$\mathsf{if}\ x = 1\ \mathsf{then}\ y := 1\ \mathsf{else}\ y := 0\ \mathsf{fi} \tag{5}$$

Aunque no hay un flujo explícito desde x a y, hay un flujo implícito porque el valor de x se copia en y. Esto es, $y := 1$ e $y := 0$ se ejecutan en un contexto donde se conoce de manera implícita información de nivel $\gamma(x)$. Por esta razón, las dos asignaciones solo pueden hacerse hacia variables de nivel $\gamma(x)$ o superior. Aquí también se pueden usar las reglas de subtipado para poder aplicar la regla. En este caso lo podemos hacer de dos formas: el tipo de $x = 1$ se puede forzar a un tipo mayor o el tipo de las asignaciones a uno menor usando la regla (CMD^-)—o sea la antimonoticidad de $\subseteq$ sobre los tipos comando.

Por otra parte es sencillo comprobar que las coerciones de tipo no van a permitir que la regla se aplique si hay un flujo descendente desde $x = 1$—o sea si $\gamma(x) \not\preceq \gamma(y)$. Por ejemplo, supongamos que $\gamma(x) = H$ var y $\gamma(y) = L$ var. En ese caso tenemos: $\gamma \vdash x = 1 : H$, $\gamma \vdash y := 1 : L$ cmd y $\gamma \vdash y := 0 : L$ cmd. Entonces, si usamos (SUBTYPE) para $x = 1$ no sirve de nada porque aumentaríamos aun más el tipo de la condición; y si aplicamos (CMD^-) a, por caso, $y := 1$ solo reduciríamos aun más el tipo de la asignación. De esta forma podemos concluir que no es posible tipar el programa (5) si el contexto de tipado es $\gamma(x) = H$ var y $\gamma(y) = L$ var.

Para concluir, el trabajo de Volpano et al. nos dice que dado el lenguaje de programación, el sistema de tipos y las reglas de tipado que ellos proponen es posible construir un *typechecker* que determina si un programa solo tiene flujos de información seguros. Dicho de otro modo, solo programas seguros pasarán el control de tipos; solo programas seguros tiparán correctamente; solo es posible tipar programas seguros. Por lo tanto vemos cómo una propiedad de *safety* (el tipado) nos permite garantizar una propiedad de *security* (flujo de información seguro) pero solo para una clase restringida de programas.

4. Unidad IV: Introducción a la criptografía aplicada

En esta sección veremos una muy breve introducción a la criptografía pero, sobre todo, lo haremos desde el punto de vista de la criptografía como una herramienta más de la seguridad informática. Es decir, no veremos a la criptografía como una rama de la matemática aplicada sino sus aplicaciones a la informática. En particular no profundizaremos en los detalles matemáticos de la criptografía sino que estudiaremos cómo utilizarla para la seguridad informática.

La criptografía existe desde mucho antes que se implementara por primera vez como software. El Imperio Romano utilizaba sistemas criptográficos para enviar mensajes militares de forma tal que los mensajeros y los posibles enemigos que aparecieran en el camino no pudieran comprender su contenido. Aun así, la criptografía se vuelve una disciplina de gran importancia con el advenimiento de las comunicaciones a distancia como el telégrafo, la radio, la telefonía, etc. a fines del siglo XIX. En general hasta hace solo un par de décadas su uso estuvo restringido al ámbito militar, los servicios de inteligencia o diplomáticos. Solo en las últimas décadas pasó a ser una tecnología al alcance de casi cualquier persona con una computadora conectada a alguna red. En la actualidad prácticamente toda la población mundial, aun sin saberlo, usa criptografía más o menos frecuentemente e incluso diariamente (por ejemplo WhatsApp encripta los mensajes sin que el usuario lo note).

Definición 7 (Criptografía)**.** *Actualmente se considera que la* criptografía *es una disciplina que combina la matemática y la informática con el fin de crear y estudiar métodos para transmitir datos de forma segura a través de canales de comunicación públicos. La seguridad se logra por medio de métodos que cambian la apariencia de un mensaje sin cambiar su significado.*

Más precisamente, dos o más sujetos acuerdan en usar un *sistema criptográfico*[59] para comunicarse entre ellos, el cual tiene como propiedad que solo ellos pueden devolverle a los mensajes su apariencia original (usando el mismo sistema criptográfico). En realidad, los sujetos *asumen* que solo ellos pueden devolverle a los mensajes su apariencia original puesto que es posible que el sistema tenga alguna vulnerabilidad a resultas de la cual un atacante también podría hacerlo.

En este sentido hay tres niveles en los cuales un sistema criptográfico puede ser atacado:

1. A nivel matemático-computacional. Todo sistema criptográfico empieza por una descripción matemática y computacional (por ejemplo, como veremos en la Sección 4.2, la criptografía asimétrica se basa en propiedades del álgebra modular y conjeturas sobre la complejidad computacional de ciertos algoritmos). Este modelo matemático-computacional podría tener errores o estar basado en hipótesis que no se verifican siempre.
2. La implementación como solución de (hardware +) software. Eventualmente el modelo matemático-computacional se implementa como un (circuito más posiblemente un) programa. Si no se siguen prácticas de desarrollo sólidas y prolijas no es raro que la implementación tenga errores que se convierten en vulnerabilidades. Además pueden aparecer efectos laterales que nunca fueron considerados en el modelo matemático-computacional. Por ejemplo, existen ataques que miden la radiación o temperatura de los circuitos electrónicos lo que brinda información sobre las operaciones que se están efectuando lo que a su vez permite deducir (al menos) partes de la clave.
3. El uso que se hace de la implementación. Finalmente ese programa se instala en una cierta computadora sobre un cierto sistema operativo y es utilizado desde ciertas aplicaciones

[59] 'cryptosystem', en inglés.

y por los usuarios. En general es aquí donde muchas de las hipótesis y principios sobre los cuales se basa el sistema a nivel matemático-computacional dejan de ser válidas. Por ejemplo, cualquier sistema criptográfico se basa en que las claves de encriptación permanecen en secreto pero en un sistema operativo tipo DAC un caballo de Troya invalidaría esta condición. Es decir, no se ataca al sistema en sí sino el uso que se hace de él.

Este modelo de amenazas demuestra que la criptografía *no es la bala de plata de la seguridad informática* como muchos creen. O sea, no basta con encriptar todo para mejorar la seguridad. De hecho, la criptografía es, fundamentalmente (ver definición), una herramienta para proteger comunicaciones (o sea información en tránsito) y no tanto para proteger información dentro de una computadora.

Ejemplo 28 (La firma electrónica no es tan segura)**.** *Una de las aplicaciones rutilantes de la criptografía moderna es la denominada* firma electrónica *o* digital*. Es decir, de alguna forma (ver Sección 4.2), se usa criptografía para firmar digitalmente un documento en el mismo sentido de la firma manuscrita. Algunos sostienen que la firma digital es más segura que la firma manuscrita dada la fortaleza matemática de los métodos utilizados.*

Entonces una persona sentada frente a su computadora con sistema operativo Windows o Linux usa una aplicación para firmar digitalmente un documento, por ejemplo un cheque. Supongamos que la aplicación de firma electrónica ha sido formalmente verificada contra la especificación matemática del sistema criptográfico para firmar. Más aun supongamos que el método matemático no tiene vulnerabilidades. El problema, sin embargo, no está ni en la aplicación ni en la matemática sino en el entorno computacional donde es utilizada. Windows o Linux no pueden mantener el nivel de seguridad de la criptografía. En particular, cualquier proceso podría invocar a la aplicación de firma electrónica para firmar cualquier documento (por ejemplo otro cheque). Aun si el usuario debe ingresar una contraseña para validar la firma, un caballo de Troya podría capturarla.

¿Qué mecanismos de seguridad de los que ya hemos visto sería necesario implementar para preservar el nivel de seguridad de la firma electrónica?

De hecho la criptografía no impide, por ejemplo, que el archivo encriptado en tu computadora sea borrado por un caballo de Troya. Tampoco impide que el archivo sea leído antes de ser encriptado y después de ser desencriptado (después de todo los humanos no podemos leer texto encriptado y en general las computadoras tampoco). Peor aun, existen ataques que usan criptografía.

Ejemplo 29 (Ransomware)**.** *El* ransomware *es un tipo de software hostil que, por ejemplo, puede encriptar información almacenada en la computadora de una persona con una clave que solo conoce el atacante. En general el ransomware puede usar las primitivas criptográficas instaladas en la computadora de la víctima. En consecuencia la víctima no puede acceder a su información. Para desencriptar la información el atacante solicita un rescate. De aquí a que este tipo de ataques se los llame* secuestro de información*.*

Notar que habiendo puesto en práctica procedimientos de seguridad razonables el ataque no es tan dañino como parece. Con solo efectuar respaldos diarios de la información el daño que un ataque por ransomware puede producir es limitado y la víctima no necesariamente debería pagar el rescate.

El ataque por ransomware, ¿es un ataque a la confidencialidad, la integridad o la disponibilidad?

La aplicación fundamental de la criptografía es asegurar la confidencialidad de los datos en tránsito. Sin embargo, también se la usa para asegurar la integridad y autenticidad de datos, y el no repudio de transacciones (ver Sección 4.2.3). Opuesto a estos requerimientos está, como casi

siempre, la eficiencia o desempeño del sistema criptográfico. En general, a mayor seguridad del sistema criptográfico menor eficiencia. Cualquier persona puede esperar unos segundos más al encriptar o desencriptar sus documentos dado que lo hace un par de veces al día. Sin embargo, en aplicaciones industriales esos segundos de más se deben multiplicar por millones de corridas del sistema criptográfico lo que puede tornarlo impracticable.

Observar que la definición de criptografía hace referencia a *canales públicos*. Podemos preguntarnos, entonces, ¿por qué no usar canales privados? La respuesta es muy simple: porque muchas veces no es posible y cuando lo es, suele ser muy caro. Por ejemplo, tener un canal privado entre Rosario y Buenos Aires requeriría tender un cable altamente protegido (en el sentido de que no pueda ser intervenido) entre ambas ciudades. Más allá de los costos que esto implica, asegurar la protección del cable a lo largo de 300 km es casi imposible. Las comunicaciones de radio no son un canal privado puesto que cualquiera con una antena puede escucharlas. Usar la red de telefonía fija tampoco es un canal privado porque entre un teléfono en Rosario y otro en Buenos Aires hay multitud de equipos de comunicación en manos de terceros y cables que pasan por lugares públicos donde es trivial intervenirlos. Internet tampoco es un canal privado por los mismos motivos que la red telefónica; al igual que la red de telefonía celular por las mismas razones que las comunicaciones de radio. Por otra parte, es imposible tener un canal de comunicación privado entre el comando de tierra y una escuadra de aviones de combate en vuelo.

El elemento primordial de un sistema criptográfico es el *cifrador*[60]. Un cifrador es un par de algoritmos uno de los cuales transforma el *texto legible*[61] en *texto cifrado* y el otro realiza el procedimiento inverso[62]. A estos algoritmos se los suele llamar algoritmos criptográficos. Al proceso de transformar el texto legible en texto cifrado se lo llama *encriptación* o *cifrado*; y al proceso inverso *desencriptación* o *descifrado*. Entonces, un algoritmo criptográfico recibe dos entradas: el texto (legible o cifrado) y la *clave*. La clave es una cadena de caracteres que debería alterar notablemente el funcionamiento del algoritmo. Es decir, un cambio mínimo en la clave debería producir un cambio muy grande en la salida del algoritmo (para el mismo texto). Al menos una de las claves debe permanecer en secreto para garantizar la seguridad del sistema. Una característica importante de una clave es su *longitud*, que habitualmente se mide en bits. Se considera que un algoritmo de encriptación debería ser más seguro a mayor longitud de la clave. Todas las claves de una misma longitud definen un *espacio de claves*.

Desde el punto de vista estrictamente criptográfico, la fortaleza de un sistema criptográfico se mide como la cantidad de tiempo que toma obtener el texto legible a partir del texto cifrado usando la mejor tecnología disponible pero sin utilizar los posibles efectos laterales de una implementación. De todas formas, se debe tener presente que un sistema criptográfico que se puede romper en, digamos, pocos minutos, no necesariamente es inútil. La utilidad de un sistema criptográfico depende también del *tiempo de vida del mesaje*. Por ejemplo, supongamos que a una escuadra de aviones de combate en vuelo se le comunica su objetivo, protegiendo los mensajes con un sistema criptográfico que se puede romper en *cinco* minutos. Ahora, digamos que el mensaje con el objetivo se transmite cuando restan *dos* minutos para que los aviones

[60] 'cipher', en inglés.

[61] 'plain text', en inglés.

[62] Tener en cuenta que para un cifrador no hay diferencia entre texto legible y cifrado; toman y devuelven cadenas de bytes. En otras palabras, 'legible' no significa que un humano lo pueda leer, es simplemente el nombre que le damos a la entrada del algoritmo que encripta y a la salida del que desencripta (aunque es cierto que se justifica por el uso más clásico de la criptografía).

lleguen a él. Si el enemigo descifra el mensaje a los cinco minutos ya será tarde pues el objetivo ya habrá sido atacado (tres minutos antes). Es decir el tiempo de vida del mensaje (dos minutos) es menor que el tiempo necesario para romper el sistema (cinco minutos). Aun así, ¿por qué alguien usaría tal sistema criptográfico en esa situación pudiendo usar uno mucho más fuerte? La razón suele ser la eficiencia. En general cuando un sistema criptográfico es más difícil de romper que otro, también sus procesos de cifrado y descifrado consumen más recursos. En algunos contextos estos recursos pueden ser escasos y por lo tanto se utilizan sistemas criptográficos más débiles pero seguros en ese contexto de uso.

La criptografía suele dividirse en *criptografía simétrica* o *de clave privada* o *de clave compartida* y *criptografía asimétrica* o *de clave pública*. En la primera la clave de cifrado y la de descifrado son la misma; en la segunda, son diferentes. En la criptografía asimétrica, si k es una clave (de cifrado o descifrado), la otra clave que reversa la operación no puede ser cualquiera sino que debe mantener una cierta relación matemática con k. En este sentido se habla de *par de claves* donde una de ellas se llama *clave privada* y la otra *clave pública*. Precisamente, la clave pública puede ser divulgada sin que esto comprometa la seguridad del sistema.

Los algoritmos de cifrado y descifrado deben ser tales que su composición matemática sea la identidad. Es decir, si $\mathcal{E}$ es un algoritmo de cifrado y $\mathcal{D}$ el de descifrado del mismo cifrador, debe valer:

$$\mathcal{D}(\mathcal{E}(t,k_1),k_2) = t \wedge \mathcal{E}(\mathcal{D}(t,k_1),k_2) = t$$

para todo texto t y para todo par de claves (k_1,k_2) (si se trata de un cifrador simétrico las claves son iguales). O sea que en realidad son dos algoritmos que deben usarse de una cierta forma para obtener el resultado esperado. No hay nada sustancial que diferencie el proceso de encriptar del proceso de desencriptar. Cualquiera de los dos algoritmos de un cifrador se pueden usar para cualquiera de las dos operaciones siempre y cuando se los use de manera consistente.

Cualquier sistema criptográfico se debe basar en el siguiente principio.

Principio de la criptografía *La seguridad de un cifrador debe basarse en mantener la clave en secreto y no en mantener en secreto los algoritmos.*

Es decir que la descripción matemática del algoritmo (e incluso su implementación) pueden ser públicos. Justamente, como dijimos más arriba, se espera que el algoritmo sea tal que pequeñas diferencias en la clave produzcan grandes diferencias en el texto de salida. Por lo tanto, aunque el atacante conozca los detalles del algoritmo, al probar con una clave que no es la que corresponde obtendrá una salida muy diferente a la esperada. Si el espacio de claves es suficientemente grande un ataque por fuerza bruta, en general, no será factible. El problema con mantener en secreto el cifrador es que los atacantes disponen de una cantidad de tiempo arbitraria para descubrirlo, o bien se lo debe cambiar periódicamente lo que implica tener que proponer nuevos cifradores con la misma periodicidad (tarea que no es nada simple).

Este principio se considera tan importante que cuando el gobierno de EE.UU. debió reemplazar el algoritmo de encriptación usado para comunicaciones oficiales sensibles (i.e. DES), organizó un concurso público internacional donde diversos grupos de criptógrafos propusieron sus algoritmos. Muchos de estos algoritmos habían sido publicados en conferencias y revistas científicas del área y por consiguiente eran ampliamente conocidos. El ganador del concurso fue el algoritmo Rijndael diseñado por los criptógrafos belgas Vincent Rijmen y Joan Daemen[63].

[63] Los otros finalistas del concurso fueron: RC6, MARS, Serpent y Twofish.

Rijndael es el primer cifrador público aprobado por la NSA para comunicaciones de nivel de seguridad *top secret* (ver Unidad 2). Es decir, el cifrador usado en la actualidad por el gobierno de EE.UU. para comunicaciones sensibles no fue diseñado por ciudadanos de ese país e incluso los enemigos de EE.UU. conocen en detalle su diseño.

4.1. Criptografía simétrica

Como dijimos más arriba, en la criptografía simétrica se utiliza la misma clave para cifrar y descifrar cada mensaje. Esta forma de criptografía fue la que se utilizó desde siempre hasta que en los años setenta del siglo pasado se descubrió la criptografía asimétrica (ver Sección 4.2).

Los primeros cifradores simplemente cambiaban el orden de las letras del mensaje; técnicamente se los llama *algoritmos de transposición* o *de permutación*. El remitente y el receptor debían acordar la permutación a utilizar y por tanto debían usar la misma por largos períodos de tiempo. Estos cifradores no tenían clave y por lo tanto el algoritmo debía permanecer en secreto, violando el principio de la criptografía (moderna) que vimos más arriba. Durante el Renacimiento se comenzaron a utilizar cifradores más sofisticados denominados *cifradores polialfabéticos*. En este caso se definían una o más sustituciones de letras por letras (por ejemplo, la 'a' siempre se sustituye por la 'b', la 'b' por 'c', y así sucesivamente hasta que la 'z' se sustituye por la 'a'). En el caso extremo se definía una sustitución para cada letra del texto legible de forma tal que, por caso, dos letras 'a' del mismo mensaje se sustituían por dos letras diferentes. Como en el caso anterior, estos cifradores no usaban una clave y por lo tanto el algoritmo mismo debía permanecer en secreto. Un poco más tarde se propuso lo que luego se conoció como *cifrador de Vigenère* el cual utilizaba una *palabra clave* para seleccionar la sustitución a efectuar. Estos cifradores se consideraron indescifrables hasta mediados del siglo XIX. El principio de la criptografía (moderna) se comenzó a pensar también a mediados del siglo XIX y fue explícitamente propuesto por primera vez en 1883 por Auguste Kerckhoffs (Claude Shannon lo reformularía durante el siglo XX en su teoría de la información)[64].

Como en la criptografía simétrica se usa la misma clave para encriptar y desencriptar, dos o más sujetos que quieran usar esta forma de criptografía deberán *compartir* la clave entre ellos y a la vez mantenerla *privada* (con respecto a los demás). De aquí que a esta criptografía también se la llame de clave compartida o de clave privada.

Los cifradores modernos se suelen clasificar en *cifradores de bloque* y *cifradores de flujo*[65]. Un cifrador de bloques opera con bloques de texto legible de longitud fija, es decir siempre encripta y desencripta bloques de la misma longitud; los de flujo operan sobre flujos de bits de longitud arbitraria. De aquí en más nos focalizaremos en los cifradores de bloque.

Los cifradores de bloque ejecutan las viejas técnicas de permutación y sustitución pero de formas más complejas. En la actualidad, la mayoría de los cifradores de bloque siguen un diseño propuesto por Shannon que se denomina *cifrador producto iterativo*[66]. La idea básica es iterar una operación que combina (o efectúa el *producto* de) una secuencia de transformaciones simples como sustituciones y permutaciones. Además, en cada iteración se utiliza solo una parte de la

[64] Nótese la contemporaneidad del principio de la criptografía con el desarrollo de las primeras comunicaciones a distancia. Por ejemplo, el telégrafo transcontinental de EE.UU., que conectaba ambas costas, comenzó a operar en 1861.

[65] 'stream cipher' en inglés.

[66] 'iterated product cipher', en inglés.

clave, llamada *subclave*. A cada iteración se la suele llamar *ronda*[67] y a las transformaciones que se efectúan en una ronda se las llama *función de ronda*. Una implementación muy utilizada de este diseño se debe a Horst Feistel, un físico alemán devenido en criptógrafo norteamericano que trabajó para IBM. La implementación de Feistel, llamada *cifrador de Feistel*, tiene la ventaja de que los procedimientos de cifrado y descifrado son muy similares (idealmente idénticos) requiriendo únicamente reversar la *selección de la subclave*[68]. Esta similitud entre los procedimientos implica que la implementación requiere menos recursos (memoria o circuitos) lo que en los años setenta del siglo XX era muy apreciado (y aun sigue siéndolo pero a otra escala). La selección de la subclave es otro de los elementos del diseño propuesto por Shannon que consiste en un algoritmo, llamado *seleccionador de subclaves*, que determina qué parte de la clave (i.e. qué subclave) se utiliza en cada ronda. En este sentido, reversar la selección de la subclave significa que la subclave utilizada en la primera ronda será utilizada en la última, la utilizada en la segunda lo será en la penúltima, y así sucesivamente. Una forma trivial de incrementar la seguridad de un cifrador producto iterativo es aumentar el número de rondas. Sin embargo, normalmente esto va en detrimento de la eficiencia por lo cual la definición de una función de ronda segura es crucial para lograr un cifrador seguro y eficiente.

Un cifrador de Feistel tiene una estructura matemática bien definida. Sea $\mathbb{T}$ el conjunto de todos los textos posibles[69] y sea $\mathbb{K}$ el espacio de claves. Sea $\mathcal{F} : \mathbb{T} \times \mathbb{K} \to \mathbb{T}$ la función de ronda (en este caso también llamada *función de Feistel*), $\mathcal{K} : [1,n] \times \mathbb{K} \to \mathbb{K}$ el seleccionador de subclaves y n el número de rondas. Para encriptar, el texto legible se divide en dos partes iguales, (l_1, r_1). Para cada ronda i, se calcula:

$$
\begin{aligned}
l_{i+1} &= r_i \\
r_{i+1} &= l_i \oplus \mathcal{F}(r_i, \mathcal{K}(i,k))
\end{aligned}
$$

donde $\oplus$ simboliza XOR y k es la clave. Esto implica que el texto cifrado es (r_{n+1}, l_{n+1}). Entonces, para desencriptar, el texto cifrado se divide en dos partes, (r_{n+1}, l_{n+1}). Luego, para cada ronda i $(= n, n-1, \ldots, 1)$ se calcula:

$$
\begin{aligned}
l_i &= r_{i+1} \oplus \mathcal{F}(l_{i+1}, \mathcal{K}(i,k)) \\
r_i &= l_{i+1}
\end{aligned}
$$

donde el texto legible es (l_1, r_1). Observar que la selección de claves se reversa durante el descifrado pues la primera ronda usa k_n, la segunda k_{n-1} y así sucesivamente.

4.1.1. DES

A continuación veremos con cierto detalle el que tal vez sea el cifrador simétrico de bloques más conocido, estudiado, atacado, implementado y usado del mundo, y probablemente el primero de la criptografía moderna. Se trata del Data Encryption Standard (DES) desarrollado por un grupo de IBM, liderado por Feistel, a principios de los años setenta del siglo pasado. DES fue propuesto por IBM, a pedido del National Institute of Standards and Technology (NIST) de EE.UU., como cifrador para proteger información sensible del gobierno de ese país. Luego de consultar con la NSA, el NIST tomó una versión ligeramente diferente que convirtió en estándar

[67] 'round', en inglés.
[68] 'key schedule', en inglés.
[69] Recordar que no hay una verdadera distinción entre textos legibles y cifrados.

para el gobierno en 1977. Rápidamente se convirtió en un cifrador usado en todo el mundo y ha influenciado gran parte de la investigación y desarrollo en criptografía desde aquella época.

Debido a ciertas características técnicas del cifrador y al involucramiento de la NSA, la comunidad científica tuvo sospechas de que la versión publicada por el gobierno de EE.UU. era más débil de la que aquel realmente utilizaba. Sin embargo, luego de décadas de escrutinio público solo se pudieron descubrir ataques teóricos que no podían ser usados en la práctica. De todas formas, con el avance en la capacidad de cómputo el ataque por fuerza bruta comenzó a ser factible. En 1999 una clave DES pudo ser descubierta en 22h15m. En la actualidad DES es considerado inseguro y fue reemplazado (como estándar de EE.UU.) por el Advanced Encryption Standard (AES, también conocido como Rijndael) que entró en servicio en 2002.

Aun así, la versión conocida como *Triple DES* (3DES) es considerada segura. Si DES_e representa el cifrador de DES, DES_d su descifrador, y k_1, k_2 y k_3 son tres claves DES y k es tal que $k = k_1 \frown k_2 \frown k_3$, entonces *TripleDES* encripta de la siguiente forma:

$$TripleDES_e(t,k) = DES_e(DES_d(DES_e(t,k_1),k_2),k_3)$$

y desencripta aplicando la inversa de la fórmula anterior:

$$TripleDES_d(t,k) = DES_d(DES_e(DES_d(t,k_3),k_2),k_1)$$

Si bien 3DES se sigue usando hay otros cifradores que tienen al menos el mismo nivel de seguridad pero que son más eficientes al no tener que realizar tres operaciones de cifrado.

DES es un cifrador de bloques de Feistel de 16 rondas que utiliza claves de 56 bits[70] y bloques de 64 bits. La longitud de la clave es el factor que hace que en la actualidad DES sea inseguro (3DES es más o menos equivalente a DES con claves de 112 bits y por lo tanto sigue siendo seguro). Antes de la primera ronda y luego de la última se efectúan dos permutaciones (llamadas *inicial* y *final*) pero que no tienen significado criptográfico (se las incluyó por cuestiones del hardware de la época).

Dado que DES es un cifrador de Feistel solo debemos dar la definición del seleccionador de subclaves, $\mathcal{K}$, y de la función de ronda, $\mathcal{F}$. La estructura general del seleccionador de subclaves se puede ver en la Figura 18. $\mathcal{K}$ toma como entrada un número natural entre 1 y 16 y la clave de 64 bits, y devuelve un bloque de 48 bits (k_i). La selección permutada 1 está dada por una matriz de 8×7 dividida en dos mitades horizontales de 4×7 tal que la mitad superior corresponde a C_0 y la inferior a D_0. Por ejemplo, la primera fila de esta matriz es:

57	49	41	33	25	17	9

lo que significa que el bit 57 de k será el primer bit de C_0, el 49 de k será el segundo de C_0 y así sucesivamente. Cada *left shift* está definido en una tabla donde se indica, para cada ronda, cuántos corrimientos a izquierda se deben realizar (siempre 1 o 2). Por ejemplo, en la primera ronda se debe realizar un corrimiento lo que significa que C_1 y D_1 se obtienen moviendo una posición a la izquierda[71] los bits de C_0 y D_0, respectivamente. Finalmente, la selección permutada 2 es una matriz de 8×6 que indica la posición que debe tomar cada bit de la matriz $\binom{C_i}{D_i}$. Ver los detalles en [21].

[70] En un sentido las claves son de 64 bits pero los bits 8, 16,. . . , 64 se usan para control de paridad y no participan del cifrado.

[71] Obviamente el primer bit pasa a la última posición.

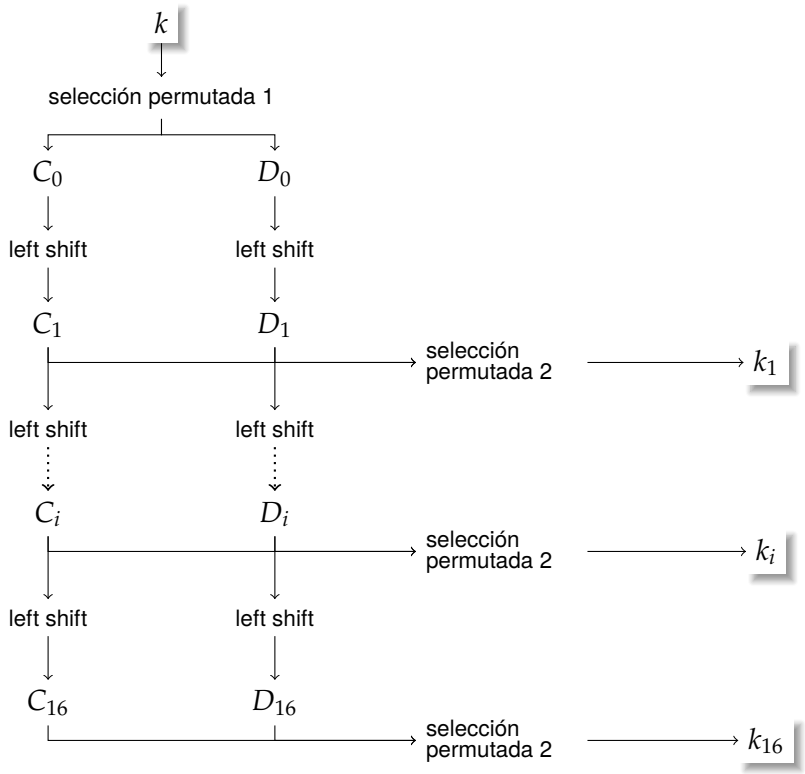


Figura 18: Estructura general del seleccionador de subclaves de DES

La función $\mathcal{F}$ recibe y devuelve bloques de 32 bits y está definida por la siguiente fórmula (ver detalles en [21]):

$$\mathcal{F}(r,\bar{k}) = P(S_1(B_1)S_2(B_2)\ldots S_8(B_8))$$

donde

- Cada B_i es un bloque de 6 bits tal que:

  $$B_1 \ldots B_8 = \bar{k} \oplus E(r)$$

  donde E es una función que recibe bloques de 32 bits y retorna bloques de 48 bits (notar que r es la parte derecha del texto legible la cual tiene 32 bits y $\bar{k}$ es una subclave que tienen 48 bits). E se llama *función de expansión*.

- Cada S_i es una matriz de 4×16 tal que si B es un bloque de 6 bits, luego:

  $$S_i(B) = S_i[m,j]_2$$

  donde

  - $S_i[m,j]_2$ es la representación binaria de la celda (m,j) de S_i
  - m es el número (de 0 a 3) que se obtiene al convertir a base 10 el número formado por el primer y último bit de B

- j es el número (de 0 a 15) que se obtiene al convertir a base 10 el número formado por los 4 bits centrales de B

- P es una permutación que trabaja sobre bloques de 32 bits

Las operaciones representadas por las funciones P, S_i y E se denominan caja-P[72] (i.e. permutación), caja-S (i.e. sustitución) y caja-E (i.e. expansión). Estas (junto a un par más) son consideradas las operaciones básicas de cualquier cifrador de bloques, algunas de las cuales fueron conceptualmente propuestas por Shannon. Cada una de estas funciones debe tener ciertas propiedades. Por ejemplo, una buena caja-S debe ser tal que: a) al cambiar un bit de la entrada se modifiquen cerca de la mitad de los bits de la salida, y b) cada bit de la salida depende de todos los bits de la entrada.

Precisamente, la controversia desatada a raíz del involucramiento de la NSA en la definición final de DES se dio debido a que esta introdujo modificaciones en las cajas-S. Sin embargo, al contrario de lo que se pensó durante años, la NSA fortaleció las cajas-S de forma tal que fuesen resistentes al criptoanálisis diferencial (técnica de ataque descubierta por aquellos años). Importantes criptógrafos, como Adi Shamir (ver Sección 4.2), demostraron que aun pequeñas modificaciones en las cajas-S pueden debilitar considerablemente a DES.

4.1.2. Modos de operación de DES

Dado que DES encripta bloques de 64 bits, ¿cómo se lo debe usar para encriptar bloques más largos? A las diversas formas de uso para encriptar textos de longitud arbitraria se las denomina *modos de operación*. Un estándar del NIST [20] define cuatro modos de operación para DES pero aquí solo veremos dos de ellos. Estos modos y algunos más pueden ser usados con cualquier cifrador de bloques.

- Electronic Code Book (ECB). El texto legible se divide en bloques de 64 bits y cada bloque se (des)encripta con la misma clave. El resultado es la concatenación de los bloques de texto cifrado. Por lo tanto, de bloques legibles iguales se obtienen bloques cifrados iguales.

- Cipher Block Chaining (CBC). El funcionamiento de CBC se grafica en la Figura 19. El texto legible se divide en bloques de 64 bits. El primer bloque se suma (XOR) con un vector de inicialización y el resultado se encripta con la clave provista; el segundo bloque se suma (XOR) con el texto cifrado del primero y el resultado se encripta con la clave provista, y así sucesivamente. De esta forma porciones iguales de texto legible no resultan en porciones iguales de texto cifrado. Este es el modo por defecto en la mayoría de las implementaciones de software disponibles.

  ¿Es necesario comunicar el vector de inicialización para más tarde poder desencriptar? ¿Hay alguna forma en que no sea necesario hacerlo?

4.2. Criptografía asimétrica

¿Por qué se inventó la criptografía asimétrica? En otras palabras, ¿qué problema hay con la criptografía simétrica que hizo necesario inventar una nueva criptografía? El problema se llama *distribución de claves*. Efectivamente, cuando dos sujetos quieren utilizar un sistema criptográfico simétrico deben acordar la clave a utilizar. Este acuerdo no puede darse sobre un canal público

[72] 'P-box', en inglés.

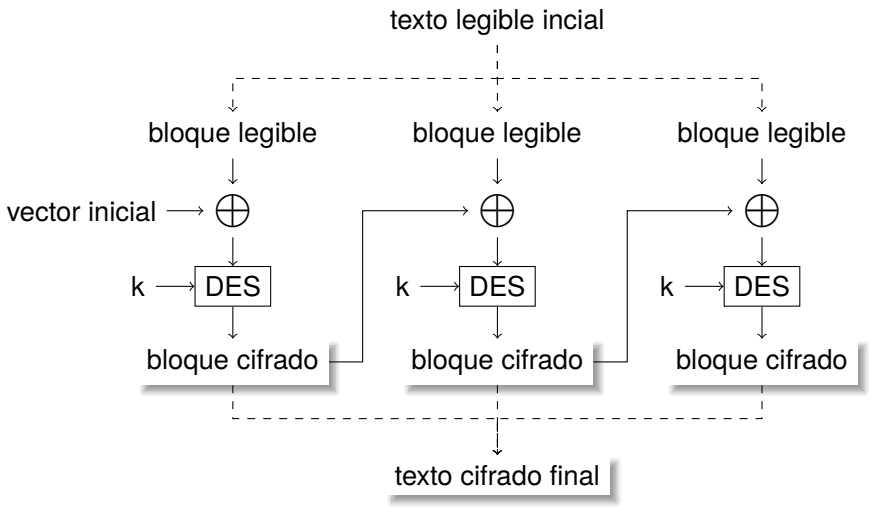


Figura 19: Representación gráfica del modo CBC (cifrado)

porque tornaría todo el sistema vulnerable. En consecuencia, deben reunirse personalmente, usar un correo confiable o negociar la clave sobre un canal criptográfico existente. Las dos primeras posibilidades son en general impracticables, sobre todo cuando se trata de un número importante de sujetos y claves (por ejemplo el sistema consular de un país como la Argentina). La tercera depende de la seguridad de una negociación de claves anterior. Precisamente, por medio de la clave pública de los sistemas asimétricos el problema de la distribución de claves se reduce considerablemente. Además, la criptografía asimétrica provee la noción de firma digital la cual habilita la propiedad de no repudio. Sin embargo, la criptografía simétrica en general es más eficiente que la asimétrica por lo que se busca combinarlas para obtener lo mejor de ambas (ver Sección 4.4.3).

La idea original de la criptografía asimétrica fue publicada por primera vez en 1976 por los criptógrafos norteamericanos Whitfield Diffie y Martin Hellman. De todas formas, más tarde se supo que tres criptógrafos del servicio secreto británico, James H. Ellis, Clifford Cocks y Malcolm J. Williamson, habían obtenido resultados similares unos años antes pero que debieron permanecer en secreto. Por otro lado, los primeros en proponer públicamente un sistema asimétrico práctico fueron tres criptógrafos norteamericanos, Ronald Rivest, Adi Shamir y Len Adleman, quienes en 1978 inventaron lo que hoy se conoce como el algoritmo RSA.

La idea básica de la criptografía asimétrica es la siguiente. Sean A y B dos sujetos que quieren comunicarse a través de un canal público. Supongamos que A quiere mandarle el mensaje t de forma confidencial a B. Sea $(\mathcal{E},\mathcal{D})$ un cifrador asimétrico y sea (s_B,p_B) el par de claves de B correspondiente a dicho cifrador (s_B es la clave secreta o privada y p_B la pública). Entonces A debe enviar a B el resultado de $\mathcal{E}(t,p_B)$ en tanto que B lo debe descifrar calculando $\mathcal{D}(\mathcal{E}(t,p_B),s_B)$ (notar que se usan claves diferentes para cifrar y descifrar).

La fortaleza del sistema yace en que se *conjetura* que es computacionalmente muy costoso (NP) obtener s_B a partir de p_B y t a partir de $\mathcal{E}(t,p_B)$ y p_B. Como podemos ver, el sistema (*casi* [ver Sección 4.2.4]) resuelve el problema de la distribución de claves dado que B puede usar un canal inseguro para enviarle p_B a A, puesto que la seguridad del sistema no depende de las claves públicas.

En general, la matemática que utiliza la criptografía asimétrica es bastante diferente de la usada por la simétrica. Los sistemas asimétricos suelen estar basados en el álgebra modular, la teoría de números y resultados de complejidad computacional[73]. Por esta razón repasaremos brevemente algunos resultados en estas áreas. De ahora en más todos los números que utilizaremos serán enteros, a menos que se indique otra cosa. Si a y b son dos números, ab indica su producto.

Decimos que $b \neq 0$ divide a a sí y solo sí $a = mb$ para algún natural m; es decir a es múltiplo de b. Sean a, b y n tres números con $n > 0$. Decimos que a es congruente con b módulo n, simbolizado $a \equiv b(\mathsf{mod}\, n)$, sí y solo sí n divide a $a - b$; o sea existe un natural k tal que $a - b = kn$, que es lo mismo que decir que $a - b$ es múltiplo de n. Sean p y q dos números primos y $n = pq$, entonces definimos $\varphi(n) = (p-1)(q-1)$. El número a es primo relativo a (o coprimo de) b si el único divisor en común de ambos es 1. A continuación enunciamos el siguiente teorema.

Teorema (de Euler)**.** *Sean p y q dos números primos y $n = pq$. Sea a primo relativo a n. Entonces:*

$$a^{\varphi(n)} \equiv 1(\mathsf{mod}\, n)$$

4.2.1. RSA

A continuación veremos con cierto detalle el algoritmo RSA, que es uno de los más utilizados en la actualidad para todo tipo de comunicaciones cifradas. RSA opera con varias longitudes de claves pero actualmente se considera que es seguro a partir de los 2048 bits. La presentación responde al artículo de Rivest, Shamir y Adleman [26], aunque no tenemos en cuenta cuestiones prácticas que hacen a la seguridad del algoritmo (por ejemplo la longitud y otras características de los números primos usados para inicializar el algoritmo). En general, muchas de las consideraciones que se verán a continuación aplican a otros sistemas asimétricos.

La inicialización del algoritmo es de la siguiente forma:

- Sean p y q dos números primos
- Sea $n = pq$
- Sean e y d tales que $ed \equiv 1(\mathsf{mod}\, \varphi(n))$
- La clave pública es (n, e)
- La clave privada es d (aunque p, q y $\varphi(n)$ también deben permanecer secretos, ver más abajo)
- Sea M la representación binaria del mensaje a encriptar
- Sean m_i con $i \in [1, k]$ tales que:
  - $M = m_1 m_2 \ldots m_k$
  - $m_i < n$
  - m_i es primo relativo a n

Entonces el mensaje M se encripta cifrando cada m_i por separado calculado el número c_i tal que:

$$c_i \equiv m_i^e(\mathsf{mod}\, n)$$

[73] Aunque hay excepciones como por ejemplo el esquema de firma digital propuesto por Lamport (Wikipedia.org: Lamport signature).

siendo el texto cifrado la concatenación de los c_i. Notar que para encriptar se usan e y n que constituyen la clave pública del destinatario.

A su vez, el destinatario, cuya clave privada es d, desencripta cada c_i calculando:

$$m_i \equiv c_i^d (\mathsf{mod}\, n) \text{ con } m_i < n$$

Veamos ahora cómo comprobar que el cálculo anterior efectivamente devuelve el mensaje original.

$$\begin{aligned}
& c_i^d (\mathsf{mod}\, n) \equiv m_i^{ed} (\mathsf{mod}\, n) && [\text{por } c_i^d = (m_i^e)^d = m_i^{ed}] \\
& = m_i^{k\varphi(n)+1} (\mathsf{mod}\, n) && [\text{por } ed = k\varphi(n)+1, \text{ ver más abajo}] \\
& = m_i m_i^{k\varphi(n)} (\mathsf{mod}\, n) \\
& = m_i \overbrace{m_i^{\varphi(n)} \ldots m_i^{\varphi(n)}}^{k\text{-veces}} (\mathsf{mod}\, n) \\
& \equiv m_i \overbrace{1(\mathsf{mod}\, n) \ldots 1(\mathsf{mod}\, n)}^{k\text{-veces}} && [\text{por T. de Euler y prop. (ver abajo) } k\text{-veces; } m_i \text{ coprimo } n \text{ por constr.}] \\
& = m_i (\mathsf{mod}\, n) \\
& \equiv m_i
\end{aligned}$$

Veamos el primer paso que falta justificar: $ed = k\varphi(n)+1$. En efecto, recordar que por construcción $ed \equiv 1(\mathsf{mod}\, \varphi(n))$. Entonces existe $k > 0$ tal que $ed - 1 = k\varphi(n)$, lo que implica el paso.

El paso donde se aplica el Teorema de Euler se justifica también por la siguiente propiedad del álgebra modular:

$$y \equiv a(\mathsf{mod}\, c) \wedge z \equiv b(\mathsf{mod}\, c) \Rightarrow (x \equiv ab(\mathsf{mod}\, c) \Leftrightarrow x \equiv yz(\mathsf{mod}\, c))$$

4.2.2. Análisis de RSA

Si un atacante conoce c_i y (n,e), ¿qué le impide obtener los m_i? Claramente cada m_i sería el entero $\sqrt[e]{c_i}(\mathsf{mod}\, n)$. El problema es que la complejidad algorítmica *conjeturada* para calcular raíces modulares es al menos NP[74]. Otro ataque posible sería obtener p y q partiendo de n; es decir factorizar n[75]. En efecto, si el atacante lograra obtener p y q tendría $\varphi(n)$ y como tiene e podría obtener fácilmente d (i.e. la clave privada). Nuevamente, el problema de este ataque es la complejidad computacional *conjeturada* para la factorización de números enteros, la cual es al menos NP. De todas formas, actualmente este último sería el ataque más prometedor. Con un poco más de precisión podemos decir que si h es la cantidad de cifras del entero n, entonces:

- La suma modular requiere h operaciones
- El producto modular requiere h^2 operaciones
- La factorización de n requiere e^n operaciones aunque hay métodos que tienen complejidades levemente menores (e.g. el método de la criba cuadrática).

[74] Todas las consideraciones sobre complejidad computacional asumen un modelo computacional *clásico*; los resultados podrían variar drásticamente en un modelo *cuántico*. De hecho, se presume que la criptografía asimétrica sería inútil en entornos cuánticos, no así la simétrica.

[75] Entendemos por 'factorizar' a la factorización de n en números primos.

- Calcular la raíz de $k(\mathsf{mod}\, n)$ requiere e^n operaciones

Por lo tanto la fortaleza de RSA se basa en elegir dos primos p y q suficientemente grandes. ¿Es tan fácil encontrar dos primos de, por ejemplo, 512 bits cada uno? No es tan fácil porque requiere comprobar que son realmente primos. Para hacerlo se utilizan algoritmos llamados *pruebas de primalidad*. Existe una prueba de primalidad que ejecuta en tiempo polinomial (el algoritmo AKS) pero que no se utiliza en la práctica debido a que las pruebas de primalidad probabilísticas son mucho más eficientes (e.g. Baillie-PSW; Fermat; Miller-Rabin; Solovay-Strassen; etc.). Es decir, son pruebas de primalidad que determinan si un entero es primo con cierta probabilidad (i.e. un *probable primo*). Las implementaciones de sistemas asimétricos suelen tener listas precalculadas de números primos hasta cierta longitud para mejorar la eficiencia. En cualquier caso, lo que queremos resaltar es que generar las claves de un sistema asimétrico no es algo que puedan hacer los usuarios (no es como elegir una contraseña para Gmail), sino que deben utilizar programas para este propósito.

Dependiendo del hardware utilizado, una clave RSA de 1024 bits es aproximadamente equivalente a una clave simétrica de 80 bits; una de 2048 bits lo es de una de 112 bits (o sea 3DES); una de 3072 bits lo es de una de 128 bits; y una de 15360 bits lo es de una de 256 bits. El sistema de Diffie-Hellman es más o menos equivalente en fortaleza a RSA.

4.2.3. Firma digital o electrónica

Sea A un sujeto cuyo par de claves (s_A, p_A) corresponde al cifrador asimétrico $(\mathcal{E}, \mathcal{D})$. Decimos que el documento electrónico c está *firmado digitalmente* o *electrónicamente*[76] por A si $c = \mathcal{E}(t, s_A)$ para algún texto t. Es decir, firmar digitalmente un documento no es más que encriptarlo con la clave privada. De esta forma, cualquiera puede corroborar lo que firmó A con solo calcular $\mathcal{D}(c, p_A)$.

Si bien en teoría la firma digital garantiza el no repudio, en la práctica esto no es razonable en ciertos contextos por razones ajenas a la criptografía asimétrica (ver Ejemplo 28). Aun así, la firma digital es un plus con respecto a la criptografía simétrica dado que esta *no* puede garantizar el no repudio en *ningún* contexto.

Lo que sí puede garantizar la firma digital es preservar la integridad del mensaje *luego* de haber sido firmado (ver Sección 4.4.2). Es decir que si un atacante modifica el mensaje en tránsito hacia su destinatario, este lo podrá detectar. Esto es así puesto que alterar el mensaje requeriría volverlo a firmar lo cual es imposible dado que se necesita la clave privada. Otra posibilidad sería que el atacante modificara el mensaje en su forma cifrada, pero el destinatario lo comprobaría porque obtendría un valor basura (puesto que la modificación se hace totalmente a ciegas). Por ejemplo, si se firma un documento PDF y se lo modifica sin desencriptarlo, cuando el destinatario lo desencripta no obtiene un PDF puesto que seguramente habrá perdido el formato.

No obstante, en la práctica y por razones de eficiencia, en realidad solo se firma un *resumen criptográfico*[77] del mensaje que se quiere firmar (ver Sección 4.3). Es decir, se envía el mensaje en su forma legible acompañado del resumen firmado. Entonces el destinatario recalcula el resumen, desencripta el resumen que recibió y los compara. Además, esto permite comprobar la integridad del mensaje de forma automática (ver Sección 4.4.2). Como mencionamos más

[76] En realidad existe una diferencia entre firma digital y firma electrónica pero no vale la pena entrar en esos detalles.

[77] 'message digest', en inglés

arriba, en general, los sistemas asimétricos son poco eficientes por lo que firmar un mensaje de gran tamaño puede llevar mucho tiempo y recursos, de aquí que en la práctica se firmen resúmenes criptográficos.

4.2.4. Certificados digitales y autoridades de certificación

Como dijimos más arriba la criptografía asimétrica *casi* resuelve el problema de la distribución de claves. Claramente, A y B pueden enviarse mutuamente sus claves públicas a través de canales inseguros. No obstante, ¿qué seguridad tiene A de que la clave que recibe es la de B (y viceversa)? Si el canal es inseguro, podría haber un atacante, C, que captura la clave de B y le envía la suya a A diciéndole que es la de B. Por lo tanto, cuando A encripta un mensaje secreto para B lo hace, en realidad, con la clave pública de C lo que le permite a este conocer el contenido del mensaje (y además reenviar otro a B). A este tipo de ataque se lo denomina *ataque de intermediario*[78].

Ejemplo 30 (Servidores de claves)**.** *En la práctica las claves públicas se pueden subir a sitios web (por ejemplo* `http://pgp.mit.edu/`*) que no controlan la identidad de los usuarios.*

Por otro lado este problema es menos severo que el de la distribución de claves en el caso simétrico. Por ejemplo, dos personas que se conocen podrían intercambiar sus claves públicas telefónicamente (aunque un atacante esté escuchando la conversación). En el caso simétrico el atacante solo debe intervenir el canal; en el caso asimétrico debe además ser capaz de eliminar, agregar y rerutear mensajes. En el caso asimétrico al atacante no le basta con comprometer la integridad de la clave pública sino que debe cambiarla por una bajo su control, sin que los usuarios legítimos lo noten. En un sentido podemos decir que se cambia un problema de confidencialidad por otro de autenticidad. En otro sentido, se cambia el problema de la distribución de claves por el *problema de la autenticidad de claves*.

La criptografía asimétrica provee la base para reducir considerablemente el problema de la autenticidad de claves. A este fin se puede crear una estructura jerárquica (del tipo del sistema de nombres de dominio, DNS) conocida como *infraestructura de clave pública* (PKI[79]). Dos elementos claves de una PKI son los *certificados digitales* (CD) y las *autoridades de certificación* (AC). La función de una PKI es gestionar CDs (emitir [o crear], distribuir, almacenar y revocar).

Un CD es un documento electrónico con un cierto formato estándar que esencialmente contiene los siguientes datos:

- Nombre (identificador) del propietario del CD
- Clave pública del propietario
- Identificación de la AC que lo emite
- Fecha de expiración o caducidad
- Un número de serie que identifica el CD

La función de una AC es *certificar la autenticidad de claves públicas* contenidas en CDs. Con este fin, un usuario (persona u organización) le solicita a una AC que certifique su clave pública de forma tal de poder distribuirla de forma segura. La AC puede generar la calve pública a nombre del solicitante o tomar la que este haya generado. Seguidamente la AC comprueba la identidad

[78] 'man-in-the-middle attack', en inglés.
[79] 'public key infrastructure', en inglés.

del usuario. Esta comprobación va desde un proceso automático hasta una auditoría sobre la organización solicitante. Uno de los usos más difundidos de los CD es demostrar propiedad sobre un nombre de dominio (e.g. `amazon.com`). En este caso la CA solo comprueba que el solicitante tiene control sobre ese dominio, y no necesariamente comprueba la identidad del usuario.

Una vez que la identidad del solicitante ha sido comprobada, la AC *firma digitalmente* el CD. Es decir, cada AC tiene su propio par de claves y su propio CD. Con esta firma los usuarios pueden comprobar la autenticidad del CD de la siguiente forma. Digamos que R es una AC que ha firmado el CD c a nombre del usuario A. Ahora cuando el usuario B recibe c comprueba su autenticidad corroborando la firma 'escrita' en c. Es decir, B debe desencriptar c con la clave pública de R. Pero claramente el problema se traslada a que B pueda corroborar la clave pública de R. Es aquí donde aparece la estructura jerárquica de la PKI. Para que B pueda corroborar la clave pública de R deberá tener el CD a nombre de R el cual: *(i)* está firmado por una AC de mayor nivel; o *(ii)* R es la AC raíz. Si es el caso *(i)*, B repite el proceso recursivamente hasta llegar al caso *(ii)*. Si es el caso *(ii)* B debe recibir el CD raíz por un canal seguro; el CD raíz no está firmado. Si a B le falta alguno de los CDs intermedios los puede solicitar o buscar en la PKI; en caso de que alguno venga con una firma que no se puede corroborar la clave pública de A no es confiable y B la usa bajo su propio riesgo.

Como prácticamente todas las soluciones de seguridad, la PKI no es perfecta. Suelen ser software más o menos complejos que pueden tener errores y, como consecuencia, vulnerabilidades. Además cuando se gestionan grandes cantidades de CDs puede haber errores humanos de todo tipo. También es posible que se comprometa la clave privada de alguna AC lo que comprometerá la seguridad de una parte de la PKI. Otra posibilidad es que haya errores en el proceso de comprobar la identidad de los solicitantes de CDs. Finalmente, aun son necesarios canales seguros (no criptográficos) para comunicar, al menos, el CD raíz.

En la práctica existen AC mundialmente reconocidas (Comodo, Symantec, etc.) que son empresas globales cuyo negocio es emitir y firmar CDs. También recientemente se han creado iniciativas no comerciales como Let's Encrypt. En estos casos los CD raíz se distribuyen con los sistemas operativos de uso masivo o los navegadores más populares. Además los gobiernos suelen tener sus propias ACs para CDs que se usan en actividades oficiales o legales. La forma de distribución depende de cada caso. Finalmente, cualquier usuario u organización puede utilizar software PKI (abiertos y gratuitos como OpenSSL, EJBCA, OpenCA, etc.; o comerciales) para gestionar sus propios CDs. En este caso la distribución de los CD raíz debe hacerse de alguna forma definida por el usuario.

4.3. Funciones *hash* criptográficas

La última primitiva criptográfica que veremos son las denominadas *funciones hash criptográficas*. La entrada de una función hash criptográfica se llama *mensaje* y la salida *resumen de mensaje* o solo *resumen*[80]. Las funciones de este tipo se utilizan para verificar la integridad y autenticidad de mensajes (ver Secciones 4.4.1 y 4.4.2). Una función hash criptográfica es una función *hash* que además, idealmente, debe tener las siguientes propiedades:

- Son eficientes al calcular el resumen de un mensaje cualquiera

[80]En este contexto un mensaje debe verse como cualquier texto que incluso no necesariamente es transmitido entre dos sujetos.

- Dados dos mensajes muy similares, sus resúmenes deben ser muy diferentes.
- Es computacionalmente muy complejo (o intratable) obtener el mensaje partiendo del resumen
- Es computacionalmente muy complejo (o intratable) encontrar dos mensajes con el mismo resumen. Si la función cumple esta propiedad se dice que es *fuertemente resistente a colisiones*.

Obviamente la idea detrás de las últimas tres propiedades es que un atacante sea incapaz de reemplazar un mensaje por otro con el mismo resumen.

Una forma de encontrar funciones hash criptográficas es buscando *funciones de compresión unidireccionales libres de colisiones*[81] (llamado diseño de Merkle-Damgård[82]). Las funciones de compresión unidireccionales libres de colisiones son básicamente algoritmos de cifrado simétrico de bloque (por ejemplo DES en modo CBC). Esto es útil en contextos donde los recursos son muy escasos y se debe proveer tanto un cifrador de bloques y una función hash criptográfica. Dos de las funciones hash criptográficas más usadas y basadas en el diseño de Merkle-Damgård son SHA-1 (diseñada por la NSA) y MD5 (diseñada por Rivest; aunque actualmente se la considera insegura, sigue siendo popular).

4.4. Aplicaciones

En esta sección veremos algunas aplicaciones típicas de las primitivas criptográficas que hemos estudiado más arriba.

4.4.1. Autenticación en UNIX (y algo más)

Usualmente se dice que las contraseñas en UNIX (y en general en los sistemas operativos) se guardan encriptadas. En realidad no es del todo correcto. Históricamente, el procedimiento era el siguiente:

- Cuando el usuario ejecuta el comando passwd, este encripta con DES una cadena de ocho ceros usando como clave la contraseña suministrada por el usuario[83]. Simbólicamente:

$$DES_e(\overbrace{00000000}^{8}, \mathit{password})$$

- Luego passwd guarda el resultado de esa encriptación en el archivo /etc/passwd.
- Cuando el usuario se logea, el programa login encripta con DES una cadena de ocho ceros usando como clave la contraseña suministrada por el usuario y compara este resultado con lo que está almacenado en /etc/passwd.

O sea que en realidad nunca se guarda la contraseña encriptada. Observar que antiguamente las contraseñas de UNIX estaban limitadas a ocho caracteres, es decir 64 bits, que es la longitud de las claves de DES.

[81] Wikipedia.org: One-way compression function
[82] Wikipedia.org: Merkle-Damgård construction
[83] Esta es una versión simplificada. En realidad se ejecutaban 25 encriptaciones sucesivas con una versión ligeramente diferente de DES.

En las versiones actuales de UNIX el procedimiento es básicamente el mismo con un par de modificaciones. Como hemos visto las 'contraseñas' no se guardan más en /etc/passwd sino en /etc/shadow. Además, en general, se usa una función hash criptográfica en lugar de un cifrador de bloques. Es decir se calcula un hash criptográfico de la contraseña del usuario en lugar de encriptar una cadena fija. La función hash queda determinada por parámetros de configuración (en distribuciones de Linux puede ser MD5, Blowfish, eksblowfish, SHA-256, SHA-512).

En cualquiera de los dos casos, en realidad, también se utiliza una cadena aleatoria, llamada *salt*, que influencia la encriptación o el hasheo final. El salt lo elije passwd y lo almacena en texto legible en /etc/password o /etc/shadow, según de qué versión se trate. El salt se agrega como una forma de reducir las posibilidades de éxito de los *ataques por diccionario* (es decir usar un diccionario de posibles contraseñas y probar una tras otra[84]). En el caso de usar funciones hash criptográficas el salt simplemente se concatena a la contraseña provista por el usuario. Simbólicamente se calcula lo siguiente (por ejemplo con SHA):

$$SHA(salt \frown password)$$

Mientras que en el caso de DES, Robert Morris y Ken Thompson en 1979 modificaron la función E (ver Sección 4.1.1) de manera tal que recibe como parámetro el salt. Entonces simbólicamente se calcula lo siguiente:

$$\overline{DES}_e(salt, \overbrace{00000000}^{8}, password)$$

donde $\overline{DES}_e$ es la versión modificada de DES.

No guardar claves. Un corolario del principio de la criptografía es que la clave nunca se transmite junto al texto cifrado, aunque se la ofusque de diversas formas. Esta característica sería parte del sistema criptográfico y entonces se debe asumir que eventualmente será descubierta haciendo que el sistema se vuelva inseguro. De todas formas, las implementaciones de herramientas criptográficas destinadas a ser usadas por personas (por ejemplo GNU Privacy Guard) usualmente guardan junto al texto cifrado la 'contraseña encriptada', más o menos como hace UNIX.

Ejemplo 31 (GPG)**.** *GNU Privacy Guard (GPG) es un software libre para que los usuarios puedan realizar diversas operaciones criptográficas*[85]*. Para encriptar con GPG el archivo* apunte.tex *con el cifrador simétrico por defecto (AES-128, i.e. Rijndael con claves de 128 bits), hacemos:*

```
$ gpg --symmetric apunte.tex
Introduzca contraseña: hola
```

[84]Los ataques por diccionario son más o menos factibles debido a que las personas tienden a elegir contraseñas 'fáciles'. Es decir tienden a utilizar solo un subconjunto (relativamente pequeño) del espacio de claves. Para evitar estos ataques, además del salt, se usan otras técnicas como filtros de contraseñas que exigen que las contraseñas tengan números, mayúsculas, signos de puntuación, etc.

[85]GPG, de alguna forma, recuerda a Pretty Good Privacy (PGP) desarrollado por Phil Zimmermann con el fin de permitirle al usuario común tener seguridad en sus comunicaciones y datos. En 1993 Zimmermann fue acusado por el gobierno de EE.UU. por 'exportar municiones sin licencia' debido a que PGP incluía algoritmos criptográficos protegidos por ese país y era usado fuera de EE.UU. Parecería que para el gobierno de EE.UU. la criptografía es un arma.

Repita contraseña: hola

lo que genera el archivo apunte.tex.gpg.

Ahora intentemos desencriptarlo pero con una contraseña errónea:

```
$ gpg -d -o apunte.tex apunte.tex.gpg
gpg: datos cifrados AES
Introduzca contraseña: chau
gpg: cifrado con 1 contraseña
gpg: descifrado fallido: clave incorrecta
```

o sea que GPG de alguna forma sabe cuál fue la contraseña que se usó al momento de encriptar el archivo.

Como era de esperarse, GPG no guarda la contraseña sino que calcula un hash criptográfico de ella y guarda este hash junto al texto cifrado (similar a lo que hace UNIX con las contraseñas). Cuando el usuario quiere desencriptar el archivo, GPG pide la contraseña, vuelve a calcular el hash y lo compara con el que está almacenado. Si son iguales desencripta el archivo; caso contrario muestra el error de más arriba. ¿Por qué GPG hace esto?

4.4.2. Integridad de datos

Proveer integridad de datos en tránsito significa que el destinatario pueda detectar si los datos fueron modificados[86]. Más precisamente, se supone que el atacante puede modificarlos pero no de manera útil y cuando lo hace el destinatario puede detectarlo. En este caso asumimos que no se requiere proveer confidencialidad; si así fuera, se debería simplemente (además) encriptar los datos.

Ejemplo 32 (Transacciones bancarias)**.** *Un ejemplo típico de comunicaciones que necesitan integridad pero no necesariamente requieren confidencialidad son las transacciones bancarias. En particular las transferencias de dinero de una cuenta a otra. La mayor parte de la actividad bancaria requiere transparencia por lo que la confidencialidad de algún modo la contradice.*

Para proveer integridad de datos se puede utilizar criptografía simétrica o asimétrica. Cuando se usa criptografía simétrica se calcula un *código de autenticación del mensaje* (MAC[87]). Un MAC es similar a un resumen de mensaje solo que se usa un cifrador de bloques (en general en modo CBC) y una clave de encriptación para generarlo[88]. El modo CBC de DES con un vector de inicialización solo de ceros es una forma de calcular MACs. En este caso el MAC de los datos es el último bloque encriptado, el cual depende de todos los bloques anteriores. En consecuencia un cambio en cualquier bit del texto legible modificará sensiblemente el MAC. Entonces el remitente envía el mensaje formado por el texto legible y el MAC correspondiente. Cuando el destinatario lo recibe, recalcula el MAC (sobre el texto legible que recibió) y lo compara con el MAC recibido. Si coinciden, el texto no fue modificado en tránsito; caso contrario el texto recibido no corresponde al enviado. Observar que el destinatario necesita la clave que fue usada para calcular el MAC (se supone que es una clave previamente acordada, como siempre en criptografía simétrica). Simbólicamente, si t son los datos a enviar y k es la clave de encriptación

[86] En este contexto 'datos' pueden ser también archivos y 'tránsito' puede ser también un archivo que es guardado en algún medio que en algún momento está fuera del control de su propietario (e.g. un *pendrive* o incluso una *laptop*).

[87] 'message authentication code', en inglés.

[88] Wikipedia: CBC-MAC

para calcular el MAC, el mensaje que el remitente debe enviar es el siguiente[89]:

$$(t, MAC^{DES}(t,k))$$

donde $MAC^{DES}(t,k)$ se calcula de la siguiente forma. Sean $t_1, \ldots, t_n$ bloques de 64 bits[90] tales que $t = t_1 \frown \cdots \frown t_n$ y sean $c_1, \ldots, c_n$ tales que:

$$c_1 = DES_e^{CBC}(t_1, k) \qquad \text{[se usa } \mathbf{0} \text{ como vector de inicialización]}$$
$$c_i = DES_e^{CBC}(c_{i-1} \oplus t_i, k), \text{ para } 1 < i \leq n$$

entonces

$$MAC^{DES}(t,k) = c_n$$

Cuando se usa criptografía asimétrica se utilizan funciones hash criptográficas para calcular el resumen de los datos a enviar. En este caso, el remitente calcula el hash de los datos, lo encripta con la clave pública del destinatario y lo envía junto a los datos. El destinatario recalcula el hash (sobre el texto legible que recibió), desencripta (con su clave privada) el resumen de mensaje recibido y los compara. Simbólicamente, si t son los datos a enviar, $(\mathcal{E}, \mathcal{D})$ es el cifrador asimétrico, $\mathcal{H}$ es la función hash criptográfica y (s_B, p_B) es el par de claves del destinatario, el mensaje que el remitente debe enviar es el siguiente:

$$(t, \mathcal{E}(\mathcal{H}(t), p_B))$$

en tanto, asumiendo que B recibe (t', h), la comprobación que debe hacer es la siguiente:

$$\mathcal{D}(h, s_B) = \mathcal{H}(t')$$

¿Qué ventajas tiene un método sobre el otro? ¿Por qué el vector de inicialización del método simétrico es **0**? ¿Cómo se podría proveer autenticidad en lugar de integridad usando métodos similares? ¿Cómo se podrían proveer ambas?

Otra posibilidad para proveer integridad de datos es usar la firma digital. ¿Cómo se sería el proceso en este caso? ¿Cuáles serían los ataques posibles (y por qué en la práctica no son efectivos)? ¿Se podría proveer integridad y autenticidad al mismo tiempo?

4.4.3. Negociación de una clave simétrica

Cuando introdujimos la criptografía asimétrica mencionamos que para cifrar es menos eficiente que la simétrica, aunque es más eficiente en relación a la distribución de claves. Consecuentemente lo que se busca es combinarlas para tener lo mejor de ambas. La aplicación canónica de esta combinación es la *negociación de una clave simétrica de sesión* para proveer confidencialidad.

Una sesión es una secuencia finita de mensajes que intercambian dos o más sujetos. Por ejemplo, durante una sesión de chat, o una compra on-line, pero fundamentalmente en *protocolos criptográficos* (ver Secciones 4.4.4 y 4.4.5 y 5). El tiempo de duración de la sesión puede ir

[89] Observar que los datos se envían legibles.

[90] Usamos bloques de 64 bits porque usamos DES para calcular el MAC. Para otros cifradores de bloque se deben usar las longitudes de bloque correspondientes.

de unos pocos segundos a pocas horas. En este contexto cada sujeto tiene su par de claves correspondientes a un cierto cifrador asimétrico. Estos sujetos podrían encriptar los mensajes con las claves públicas de los destinatarios pero si deben transmitir mensajes con *payloads* grandes los tiempos de cifrado y descifrado de la criptografía asimétrica pueden volverse rápidamente inaceptables.

Por lo tanto, primero negocian una clave simétrica para la sesión usando criptografía asimétrica, y luego usan esa clave simétrica para encriptar los mensajes de la sesión en sí mediante criptografía simétrica. En otras palabras, la sesión se divide en dos etapas:

1. Negociación de la clave simétrica de sesión
2. Comunicaciones semánticas (o la sesión en sí)

Digamos que los sujetos A y B quieren negociar una clave simétrica de sesión para el cifrador simétrico $(\mathcal{E}_s, \mathcal{D}_s)$. Cada uno tiene su par de claves, (s_A, p_A) y (s_B, p_B), correspondientes al cifrador asimétrico $(\mathcal{E}_a, \mathcal{D}_a)$. Asumiendo que A inicia la sesión, proceden de la siguiente forma:

M1. A $\rightarrow$ B: p_A (¿Cómo está seguro B que p_A es la clave de A?)

M2. B $\rightarrow$ A: p_B

M3. A $\rightarrow$ B: $\mathcal{E}_a(k, p_B)$ (B desencripta con $\mathcal{D}_a(_, s_B)$)

M4. B $\rightarrow$ A: *ok*

M5. A $\rightarrow$ B: $\mathcal{E}_s(secreto, k)$ (B desencripta con $\mathcal{D}_s(_, k)$)

M6. ... $\rightarrow$ ... : ...

M7. B $\rightarrow$ A: $\mathcal{E}_s(otro_secreto, k)$ (A desencripta con $\mathcal{D}_s(_, k)$)

M8. ... $\rightarrow$ ... : ...

donde cada línea describe un mensaje entre los dos sujetos mencionados. Este intercambio de mensajes no pretende ser un protocolo criptográfico sino solamente mostrar la combinación de criptografía asimétrica y simétrica para establecer y usar una clave simétrica de sesión. Un protocolo criptográfico real debe contener más información para evitar diversos ataques (ver Secciones 4.4.4 y 4.4.5 y 5).

Hechas estas aclaraciones y salvedades, explicamos el intercambio de mensajes. En los dos primeros mensajes A y B intercambian sus claves públicas en texto legible puesto que es información que no compromete la seguridad de la sesión. Luego, A genera una clave simétrica aleatoria y se la envía a B encriptándola con su clave pública. De esta forma B es el único que la puede desencriptar. A partir del quinto mensaje comienza la etapa semántica de la sesión donde A y B intercambian sus mensajes secretos (des)encriptándolos con la clave k. Como k solo la conocen ellos, los mensajes están seguros. De esta forma la criptografía asimétrica se usa solo para negociar la clave (pocos y pequeños mensajes) y la simétrica se usa en la etapa semántica (muchos y grandes mensajes).

4.4.4. El protocolo de Needham-Schroeder (versión simétrica)

Establecer una clave simétrica de sesión requiere de un *protocolo criptográfico*. Un protocolo criptográfico es un protocolo de comunicación de datos tal que algunos de sus mensajes contienen datos encriptados y cuyo objetivo es proveer una o más propiedades de seguridad

(confidencialidad, integridad, autenticidad, etc.). En la descripción de un protocolo criptográfico se asume que las primitivas criptográficas son perfectas o infalibles, y en general solo se indica si se usa criptografía simétrica o asimétrica sin mencionar los cifradores particulares que se utilicen. Más precisamente, se asume que el único ataque posible sobre la criptografía es el de fuerza bruta y por lo tanto existe una cota superior de tiempo para el uso de las claves. O sea que una de las propiedades básicas de cualquier protocolo criptográfico es no permitir el uso de claves que hayan sido usadas en corridas anteriores del protocolo.

Lo importante en la definición de un protocolo criptográfico está en proveer: a) una secuencia mínima de mensajes; b) con la cantidad mínima de información; c) con la cantidad mínima de operaciones criptográficas; y d) tales que garanticen las propiedades de seguridad pretendidas. La exigencia de proveer mínimos se basa en la necesidad de que el protocolo sea eficiente (cuantos más mensajes o más datos o más operaciones criptográficas, más uso de la red, los equipos de comunicación, etc.). Como veremos en la Unidad 5, esto ha demostrado ser más complejo que lo que la intuición indica.

En esta sección y en la siguiente veremos sendos protocolos criptográficos cuyo objetivo es que dos sujetos negocien una clave simétrica de sesión. El de esta sección es el *protocolo de Needham-Schroeder* (NS, versión simétrica) y el de la sección siguiente es el *protocolo Secure Sockets Layer* (SSL, que usa criptografía asimétrica).

La negociación de una clave simétrica de sesión que mostramos en la sección anterior también se puede hacer exclusivamente con criptografía simétrica pero requiere de un *servidor de claves*[91] o *servidor de autenticación*. Un servidor de claves es un sujeto que provee claves de sesión a otros sujetos que confían en él; con este fin los sujetos comparten claves que les permiten comunicarse de forma segura con el servidor de claves (pero no entre ellos).

Entonces, tenemos un conjunto de sujetos, $A, B, C, \ldots$, y un sujeto especial, S, que actúa como servidor de claves. Cada uno de los sujetos del primer conjunto comparte una clave con S. La clave compartida entre los sujetos X e Y se nota con K_{xy}. Para decir que los textos legibles $t_1, \ldots, t_n$ están encriptados con la clave K escribimos $\{t_1, \ldots, t_n\}_K$ (dado que no nos importa el cifrador que se use). Esta misma notación la usaremos en la sección siguiente y en la Unidad 5.

Uno de los protocolos más usados y estudiados para negociar una clave de sesión teniendo un servidor de claves es el *protocolo de Needham-Schroeder* (NS) desarrollado por Roger Needham y Michael Schroeder en 1978. Por ejemplo, el protocolo Kerberos, muy usado en ambientes Windows, se basa en NS. Existe una versión de NS que se basa en criptografía asimétrica. La descripción del protocolo, en su versión simétrica, es la siguiente:

M1. A $\rightarrow$ S: A, B, N_a

M2. S $\rightarrow$ A: $\{N_a, B, K_{ab}, \{K_{ab}, A\}_{K_{bs}}\}_{K_{as}}$

M3. A $\rightarrow$ B: $\{K_{ab}, A\}_{K_{bs}}$

M4. B $\rightarrow$ A: $\{N_b\}_{K_{ab}}$

M5. A $\rightarrow$ B: $\{N_b - 1\}_{K_{ab}}$

El mensaje M1 no aporta a la seguridad del protocolo y simplemente sirve para que el sujeto que quiere iniciar la sesión (A) le indique al servidor de claves (S) quién es, con quién

[91] En esta sección todas las claves son simétricas.

se quiere comunicar (B) y que le está enviando un *fresco*[92] (N_a). Los frescos son un componente esencial en el diseño de protocolos criptográficos. La intención al incluir un fresco en un mensaje es indicar que es un mensaje nuevo, de esta ejecución o corrida del protocolo; que no es un mensaje de otra corrida. Esto es importante porque uno de los ataques típicos contra protocolos criptográficos, denominado *ataque por repetición*[93], consiste en reutilizar elementos y mensajes de corridas anteriores del protocolo para engañar a alguno de los participantes. Los frescos suelen ser números enteros.

En M2 se inicia la etapa de seguridad del protocolo. Notar que el mensaje que S le envía a A va encriptado con la clave que ellos comparten (K_{as}). Es decir es un mensaje privado para A, nadie más lo puede desencriptar. Recordar que en el diseño de protocolos criptográficos se asume que los cifradores son infalibles; los ataques serían contra la secuencia de mensajes. En este mensaje S le dice a A que este mensaje es en respuesta al que él mandó antes, debido a la presencia de N_a y B. Luego le manda la clave que deberá usar para comunicarse con B. Finalmente S le envía un *certificado* encriptado con la clave que este comparte con B. El certificado le dice a B que debe usar K_{ab} para comunicarse con A.

Entonces A desencripta este mensaje, extrae el certificado para B y se lo reenvía en M3. Cuando B recibe este mensaje confía en la clave K_{ab} pues viene certificada por S.

Luego, en M4, B desafía a A con un fresco para constatar que este está en conocimiento de la clave K_{ab}. Es decir, B intenta asegurarse que la clave que recibió en el mensaje anterior no es una repetición[94] de alguna corrida anterior del protocolo, posiblemente enviada por alguien que intenta hacerse pasar por A. En M5, A responde modificando el fresco lo que implica que fue capaz de desencriptar el mensaje anterior y por lo tanto conoce la clave. Observar que, implícitamente, esto excluye la posibilidad de que un atacante haya desencriptado el mensaje porque rompió el cifrador. En la descripción del protocolo la respuesta esperada al desafío es $N_b - 1$, pero cualquier modificación previamente acordada serviría. El punto es no permitir que la respuesta al desafío sea el mismo mensaje, puesto que así sería trivial sustituir a A. A este par de mensajes se le suele llamar *desafío de actualidad*[95].

A partir del mensaje M5, los sujetos A y B pueden comenzar a comunicarse confidencialmente entre sí usando la clave K_{ab}.

Ataque por repetición. NS es vulnerable a ataques por repetición. En otras palabras, un atacante, C, puede reutilizar una clave de una ejecución vieja del protocolo y así interponerse en la comunicación entre A y B. Esto es un error del protocolo, no de la criptografía. Por ejemplo, digamos que en los años ochenta del siglo XX DES podía romperse en dos semanas usando fuerza bruta. Ahora supongamos que una implementación de NS de aquella época usaba DES como cifrador. Entonces, C podía escuchar una ejecución del protocolo, capturar algunos mensajes, romper la clave por fuerza bruta en dos semanas, y volverla a usar en una nueva sesión del protocolo. NS no debería permitir usar claves viejas porque todas las claves tienen un tiempo máximo de vida.

[92] 'nonce', en inglés. La traducción literal de esta palabra es 'presente', 'momento', 'ocasión' o 'esta vez'. Ninguna me satisface para este contexto. Otras traducciones que he visto más cerca de este contexto son 'semilla' (en el sentido de una semilla para la generación de números aleatorios), 'valor único', 'número aleatorio', etc.

[93] 'replay attack', en inglés.

[94] 'replay', en inglés.

[95] 'nonce handshake', en inglés.

Entonces, el ataque es el siguiente. Supongamos que C tiene la clave K'_{ab} usada en una corrida vieja de NS entre A y B. Entonces inicia la siguiente secuencia de mensajes:

M1. C $\rightarrow$ S: A, B, N_a (C se hace pasar por A)

M2. S $\rightarrow$ C: $\{N_a, B, K_{ab}, \{K_{ab}, A\}_{K_{bs}}\}_{K_{as}}$

M3. C $\rightarrow$ B: $\{K'_{ab}, A\}_{K_{bs}}$ (C le envía a B la clave vieja)

M4. B $\rightarrow$ C: $\{N_b\}_{K'_{ab}}$ (B desafía con la clave vieja)

M5. C $\rightarrow$ B: $\{N_b - 1\}_{K'_{ab}}$ (C puede responder porque conoce la clave vieja)

El resultado es que B cree estar hablando con A cuando en realidad lo hace con C. El problema de fondo es que B no tiene forma de comprobar que la clave es vieja. Una de las propiedades de las claves negociadas en protocolos criptográficos es que todos los participantes puedan determinar que son frescas (es decir que tengan las propiedades de los frescos). Específicamente, a C no le interesa M2 porque viene encriptado con la clave entre A y S pero él sabe que para seguir NS debe tomar el certificado y reenviarlo a B. Justamente C tiene el certificado de la corrida vieja cuya clave conoce, y entonces le reenvía este certificado a B. Como dijimos, el problema es que B no puede comprobar que el certificado es viejo y cae en la trampa. En otras palabras, *B debe asumir o confiar* que K'_{ab} es fresca, *no lo puede comprobar*. Esta conjetura no fue tenida en cuenta por Needham y Schroeder cuando diseñaron el protocolo.

Soluciones. Existen al menos dos soluciones a este ataque. Kerberos incluye *time-stamps* en algunos mensajes pero esto tiene la desventaja de requerir una forma global y precisa de medir el tiempo en la red. En 1987 Needham y Schroeder [24] propusieron una modificación del protocolo donde B puede corroborar que la clave generada por S es fresca:

M1. A $\rightarrow$ B: A

M2. B $\rightarrow$ A: $\{A, N_b\}_{K_{bs}}$

M3. A $\rightarrow$ S: $A, B, N_a, \{A, N_b\}_{K_{bs}}$

M4. S $\rightarrow$ A: $\{N_a, B, K_{ab}, \{K_{ab}, A, N_b\}_{K_{bs}}\}_{K_{as}}$

M5. A $\rightarrow$ B: $\{K_{ab}, A, N_b\}_{K_{bs}}$

M6. B $\rightarrow$ A: $\{N'_b\}_{K_{ab}}$

M7. A $\rightarrow$ B: $\{N'_b - 1\}_{K_{ab}}$

con la desventaja de requerir más mensajes, más datos (N_b y N'_b) y que B mantenga más estado. Ahora en M5 B puede corroborar que K_{ab} es fresca porque el certificado incluye el fresco que él mismo generó al inicio de la corrida. Recordar lo que decíamos al inicio de la sección sobre las dificultades que entraña requerir mínimos en el diseño de protocolos.

4.4.5. El protocolo Secure Sockets Layer

El protocolo *Secure Sockets Layer* (SSL) fue durante años 'el' protocolo de la era de Internet. Fue concebido cerca de 1995 por Taher Elgamal trabajando para Netscape y rediseñado (SSL 3.0) por Paul Kocher y Netscape al año siguiente debido a problemas de seguridad. SSL fue desarrollado para que el navegador web Navigator de Netscape soportara HTTPS. En 1999 algunos ingenieros que trabajaron en el desarrollo de SSL (Christopher Allen y Tim Dierks de

Certicom), definieron el *Transport Layer Security* (TLS) que en actualidad debería usarse en lugar de SSL. El objetivo primario de SSL y TLS es proveer confidencialidad, integridad y autenticidad en las comunicaciones de aplicaciones tales como navegación web, correo electrónico, mensajería instantánea y voz sobre IP. Tanto SSL como TLS soportan una variedad de métodos para negociar claves, encriptar datos, autenticar a los participantes y corroborar la integridad de los datos, que pueden ser seleccionados por los participantes (e.g. Diffie-Hellman, RSA, FORTEZZA, MD5, SHA, AES, 3DES, etc.).

En esta sección veremos una de las configuraciones más simples de SSL para negociar una clave simétrica de sesión. Los sujetos participantes son el cliente C (normalmente un navegador web) y el servidor S (normalmente un servidor web). En esta configuración cliente y servidor presentan sus CDs. Las partes de mensajes notadas con N (más algún subíndice) representan frescos. Si X es un participante entonces (K_x^{-1}, K_x) denota su par de claves privada y pública y C_x su CD. La descripción del protocolo en esa configuración es la siguiente (ver explicación detallada más abajo):

M1. C $\rightarrow$ S: N_c, P (P: lista de algoritmos a usar)

M2. S $\rightarrow$ C: C_s, OK, N_s (OK: S acepta P)

M3. C $\rightarrow$ S: $C_c, \{K\}_{K_s}, \{CV\}_{K_c^{-1}}$

M4. S $\rightarrow$ C: SF (server finish)

M5. C $\rightarrow$ S: CF (client finish)

M6. ... $\rightarrow$ ... : ... (inicio comunicación privada)

M7. C $\rightarrow$ S: $\{$caja ahorros: $672903/6, MD5(F_c)\}_{K_{cw}}$

M8. S $\rightarrow$ C: $\{$saldo: $12790, MD5(F_s)\}_{K_{sw}}$

En M3 el cliente calcula K (llamada *pre-secreto maestro*[96]) generando 48 bytes aleatorios. A partir de K se generan las claves, como veremos a continuación. Es decir que el protocolo le permite al cliente ser él quien provea la semilla para asegurar la comunicación. Luego, C y S calculan el *secreto maestro*[97] K_{ms} de la siguiente forma (+ indica concatenación):

$$\begin{aligned} K_{ms} = & MD5(K + SHA(\text{'}A\text{'} + K + N_c + N_s)) \\ & + MD5(K + SHA(\text{'}BB\text{'} + K + N_c + N_s)) \\ & + MD5(K + SHA(\text{'}CCC\text{'} + K + N_c + N_s)) \end{aligned}$$

O sea que el secreto maestro depende de los frescos y de la cadena aleatoria generada por el cliente, además de ser el resultado de componer dos funciones hash criptográficas. La composición de las funciones hash se usa para evitar que el protocolo se vuelva inseguro si una de ella tiene fallas no críticas.

A partir del secreto maestro, C y S generan dos claves, K_{cw} y K_{sw}, que serán las que cliente y servidor, respectivamente, usen para encriptar los datos salientes[98] a partir del mensaje M6. Estas claves se calculan de la siguiente forma:

[96] 'pre-master secret', en el original en inglés.
[97] 'master secret', en el original en inglés.
[98] Y para desencriptar los entrantes.

1. Se genera el *material de claves*[99] con una fórmula similar a la usada para K_{ms}:

$$\begin{aligned} key_material = & MD5(K_{ms} + SHA(\text{'}A\text{'} + K_{ms} + N_c + N_s)) \\ & + MD5(K_{ms} + SHA(\text{'}BB\text{'} + K + N_c + N_s)) \\ & + MD5(K_{ms} + SHA(\text{'}CCC\text{'} + K_{ms} + N_c + N_s)) \\ & + \ldots \end{aligned} \tag{†}$$

   hasta que se haya generado suficiente material.

2. Luego, la cadena así generada se divide en bloques de los cuales el tercero corresponde a K_{cw} y el cuarto a K_{sw}. Los dos primeros se usan más adelante.

El cliente en M3 calcula CV (*certificate verify*) y se lo envía al servidor con el fin de que este pueda verificar el CD del cliente (notar que CV va firmado por el cliente). La fórmula para CV es[100]:

$$CV = FHC(K_{ms} + FHC(M1 + M2 + K_{ms}))$$

donde FHC puede ser MD5 o SHA. Notar que se usa toda la información contenida en los dos primeros mensajes.

El contenido de los mensajes M4 y M5 se envía cada vez que se negocia un nuevo paquete de cifradores y claves[101]. CF y SF se calculan con:

$$MD5(K_{ms} + MD5(M1 + M2 + M3 + sender + K_{ms}))$$

donde *sender* es una constante diferente para CF y SF. Notar que estos son los primeros mensajes que se protegen con todos los secretos y algoritmos negociados hasta el momento. La protección se genera al enviarse un hash criptográfico de la concatenación de todo lo que se ha generado hasta el momento.

A partir del mensaje M6, cliente y servidor intercambian mensajes a nivel de aplicación. Suponiendo que se usa un cifrador de bloques, cada mensaje debe contener un control de integridad[102] y debe ser encriptado como se indica a continuación. El control de integridad se incluye para evitar ataques por repetición.

- Los mensajes enviados por C incluyen un control de integridad, F_c, que se calcula con la siguiente fórmula:

  $$F_c = FHC(K_{cm} + FHC(K_{cm} + \textit{número mensaje} + \textit{contenido mensaje}))$$

  donde K_{cm} es el primer bloque que se obtiene de la fórmula (†).

  Luego, C encripta con K_{cw} y desencripta con K_{sw} (también llamada K_{cr})[103].

  El mensaje M7 muestra un ejemplo de esta situación.

[99] 'key material', en inglés.

[100] En esta y otras fórumulas, se omiten algunos detalles demasiado técnicos.

[101] Es decir, después de haber usado los cifradores y claves negociados inicialmente, las partes pueden volver a negociarlos (por ejemplo si la sesión es muy larga).

[102] Según el vocabulario usado en la descripción de SSL, a estos controles los llaman MAC (ver Sección 4.4.2), aunque se usan funciones hash criptográficas y no cifradores de bloque para calcularlos.

[103] w es por 'write' (es decir 'enviar') y r es por 'read' (es decir 'recibir').

- Los mensajes enviados por S incluyen un control de integridad, F_s, que se calcula con la siguiente fórmula:

  $$F_s = FHC(K_{sm} + FHC(K_{sm} + \textit{número mensaje} + \textit{contenido mensaje}))$$

  donde K_{sm} es el segundo bloque que se obtiene de la fórmula (†).

  Luego, S encripta con K_{sw} y desencripta con K_{cw} (también llamada K_{sr}).

  El mensaje M8 muestra un ejemplo de esta situación.

Como conclusión podemos destacar que el protocolo incluye información de toda la sesión para evitar ataques por repetición o por modificación de los datos. Aun así SSL 3.0 tiene algunas vulnerabilidades relacionadas más que nada con el uso de MD5 para calcular material de clave, secretos, controles de integridad, etc.; MD5 se considera inseguro desde el inicio del milenio (aunque sigue siendo bastante usado). De todas formas, SSL constituye un protocolo que fue diseñado concienzudamente y que aseguró los primeros años del despegue de Internet como plataforma para el comercio electrónico.

Esta sección debe completarse leyendo [13, secciones 1 y 2; partes relevantes de la 5 y 6; y apéndice F]

4.4.6. Criptografía y no interferencia

Como colofón, en esta sección queremos dejar planteado un problema para pensar. ¿Cuál es la relación entre la criptografía y la no interferencia? ¿Tiene algún papel la criptografía en sistemas que implementan alguna forma de no interferencia? ¿Puede la criptografía colaborar en preservar la no interferencia? ¿En qué situaciones? ¿Cuál criptografía, la simétrica o la asimétrica?

5. Unidad V: Introducción al análisis de protocolos criptográficos

¿Cómo podemos estar seguros que un protocolo criptográfico es seguro? Es decir, ¿existe alguna técnica que pueda garantizar que un protocolo criptográfico verifica ciertas propiedades? Ya hemos visto que protocolos criptográfico que se consideraron seguros durante años finalmente se demostró que no lo eran tanto. La respuesta que da la comunidad de Seguridad Informática a estas preguntas es lo que se conoce como *análisis formal de protocolos criptográficos*. Es decir, el protocolo criptográfico se formaliza en alguna lógica y se estudian formalmente sus propiedades de seguridad. Posiblemente el análisis formal de protocolos criptográficos haya nacido con el trabajo de Michael Burrows, Martín Abadi y Roger Needham que hoy se conoce como la *lógica BAN* [5]. Este formalismo *ad-hoc* para el análisis de protocolos criptográficos dio origen a una línea de investigación que aun hoy continúa activa. Aunque la publicación formal es [5] nosotros utilizaremos el reporte técnico de DEC-SRC[104] [4].

BAN se basa en dos nociones centrales: *creencias* y *mensajes vistos*. La noción de creencia se relaciona con que un *sujeto cree que cierto dato es verdadero*. En un sentido los mensajes vistos representan lo que el sujeto *conoce*. Los sujetos van desarrollado o aumentando sus creencias y

[104] Systems Research Center de Digital Equipment Corporation. En SRC trabajaron investigadores como Butler Lampson y Leslie Lamport, ambos ganadores del premio Turing de la ACM.

conocimientos a medida que se ejecuta el protocolo[105]. Ambos se modelan como proposiciones de la lógica.

El lenguaje. El lenguaje distingue tres *sorts*: sujetos, claves y sentencias. Los mensajes de un protocolo criptográfico se asocian con las sentencias. En BAN el único conector proposicional es la conjunción que se simboliza con la coma. En lo que sigue, los símbolos P, Q y R denotan sujetos; X e Y denotan sentencias; y K denota claves. El lenguaje de BAN consta de las siguientes construcciones[106]:

- $P \overset{\text{cree}}{\Longrightarrow} X$: el sujeto P puede actuar como si X fuera verdadera.
- $P \overset{\text{ve}}{\Longrightarrow} X$: alguien le ha mandado a P un mensaje conteniendo X, quien puede leerlo (posiblemente luego de desencriptarlo) y reenviarlo.
- $P \overset{\text{dijo}}{\Longrightarrow} X$: el sujeto P alguna vez mandó un mensaje conteniendo la sentencia X. El mensaje puede haberlo enviado mucho antes o durante la corrida actual del protocolo.
- $P \overset{\text{tjs}}{\Longrightarrow} X$: el sujeto P *tiene jurisdicción sobre* X. P tiene autoridad sobre X y los otros sujetos deben tenerle confianza sobre X. Por ejemplo se usa para establecer propiedades de los servidores de claves sobre las claves que ellos generan.
- $\#(X)$: la sentencia X es *fresca*[107]; es decir, nunca fue incluida en un mensaje antes de la ejecución actual del protocolo. Usualmente esto es así para los frescos.
- $P \overset{K}{\leftrightarrow} Q$: K es una clave compartida entre P y Q. Esta clave nunca será conocida por otros sujetos salvo aquel que la haya generado.
- $\overset{K}{\mapsto} P$: K es la clave pública de P; su clave privada es K^{-1}. La clave privada jamás será conocida por otro sujeto distindo de P excepto aquel que la haya generado.
- $\{X\}_K$: tiene casi el mismo significado que le dimos en secciones anteriores. La única diferencia es que esta construcción es una abreviación de $\{X\}_K \,\mathsf{from}\, P$ con el fin de poder saber quién originó el mensaje. De todas formas, cuando el contexto lo permite la parte from se omite.

La semántica formal de estas construcciones pueden verla en el trabajo original de Burrows, Abadi y Needham [4] pero no es obligatorio para este curso.

Reglas de inferencia. En BAN se distingue entre pasado, todo aquello que ocurrió antes de la corrida actual del protocolo; y presente, que es lo que ocurre durante la corrida actual del protocolo. Todos los mensajes enviados antes de la corrida actual están en el pasado y el diseño del protocolo debe ser tal que estos nunca sean considerados como pertenecientes al presente. Todas las creencias del presente son estables durante la corrida del protocolo; se asume que si P dice X entonces también cree en X. Sin embargo, las creencias del pasado no necesariamente

[105] Estas nociones estarían lejanamente relacionadas con las lógicas de conocimiento y creencias que se utilizan en varias áreas de las ciencias de la computación (ver por ejemplo [30]).

[106] En el original se usan otros símbolos y se incluyen dos construcciones que nosotros decidimos no mostrar.

[107] 'fresh', en inglés.

lo son del presente. Se asume que los sujetos pueden detectar e ignorar los mensajes que ellos mismos envían. Dicho esto es posible presentar las reglas de inferencia de BAN[108].

Las primeras reglas de inferencia tratan sobre el significado de los mensajes. Dos de ellas dan significado a los mensajes encriptados (la tercera no la veremos). Todas explican cómo derivar creencias partiendo de mensajes. La primera de las dos reglas que veremos se encarga de mensajes encriptados con claves compartidas:

$$\frac{P \overset{\text{cree}}{\Longrightarrow} Q \overset{K}{\leftrightarrow} P, \qquad P \overset{\text{ve}}{\Longrightarrow} \{X\}_K}{P \overset{\text{cree}}{\Longrightarrow} Q \overset{\text{dijo}}{\Longrightarrow} X} \tag{R1}$$

Es decir que si P cree que comparte la clave K con Q y ve el mensaje X encriptado con esa clave, entonces cree que Q alguna vez dijo X. Para que la regla sea consistente es necesario que P y Q no sean el mismo sujeto (es decir, que P se haya enviado el mensaje a sí mismo). Esto queda implícito pues $\{X\}_K$ es en realidad $\{X\}_K \,\mathsf{from}\, Q$ y entonces basta con pedir $Q \neq P$.

La segunda regla sobre el significado de los mensajes es similar a la anterior solo que trata sobre claves públicas:

$$\frac{P \overset{\text{cree}}{\Longrightarrow} \overset{K}{\mapsto} Q, \qquad P \overset{\text{ve}}{\Longrightarrow} \{X\}_{K^{-1}}}{P \overset{\text{cree}}{\Longrightarrow} Q \overset{\text{dijo}}{\Longrightarrow} X} \tag{R2}$$

Observar que al desencriptar un mensaje no se sabe si fue dicho en el presente o en el pasado; es decir, P no puede creer en él así como así. Simbólicamente no podemos deducir $P \overset{\text{cree}}{\Longrightarrow} X$, solo podemos deducir que P cree que Q lo dijo. . . alguna vez.

Luego tenemos una regla para verificar frescos, que permite deducir que el remitente aun cree en él:

$$\frac{P \overset{\text{cree}}{\Longrightarrow} \#(X), \qquad P \overset{\text{cree}}{\Longrightarrow} Q \overset{\text{dijo}}{\Longrightarrow} X}{P \overset{\text{cree}}{\Longrightarrow} Q \overset{\text{cree}}{\Longrightarrow} X} \tag{R3}$$

donde por simplicidad se asume que X es texto legible, es decir no contiene ninguna sentencia de la forma $\{Y\}_K$.

La regla anterior es la única que deduce $\overset{\text{cree}}{\Longrightarrow}$ partiendo de $\overset{\text{dijo}}{\Longrightarrow}$. Esta regla formaliza una práctica común en el diseño de protocolos criptográficos que consiste en tomar como válidos los mensajes que contienen un fresco o algo que se deriva de él. Notar que los frescos, cuando son emitidos como desafíos, no necesitan estar encriptados, aunque sí cuando son parte de la respuesta.

Ahora veremos el significado del conector $\overset{\text{tjs}}{\Longrightarrow}$:

$$\frac{P \overset{\text{cree}}{\Longrightarrow} Q \overset{\text{tjs}}{\Longrightarrow} X, \qquad P \overset{\text{cree}}{\Longrightarrow} Q \overset{\text{cree}}{\Longrightarrow} X}{P \overset{\text{cree}}{\Longrightarrow} X} \tag{R4}$$

es decir que si P le adjudica a Q jurisdicción sobre X entonces P cree en la veracidad de X.

[108] Por brevedad no presentamos todas las reglas.

Luego tenemos reglas que de alguna forma distribuyen el operador $\overset{\text{cree}}{\Longrightarrow}$:

$$\frac{P \overset{\text{cree}}{\Longrightarrow} X, \qquad P \overset{\text{cree}}{\Longrightarrow} Y}{P \overset{\text{cree}}{\Longrightarrow} (X,Y)} \qquad \frac{P \overset{\text{cree}}{\Longrightarrow} (X,Y)}{P \overset{\text{cree}}{\Longrightarrow} X} \qquad \frac{P \overset{\text{cree}}{\Longrightarrow} Q \overset{\text{cree}}{\Longrightarrow} (X,Y)}{P \overset{\text{cree}}{\Longrightarrow} Q \overset{\text{cree}}{\Longrightarrow} X} \tag{R5}$$

Otras reglas similares se pueden agregar si fuese necesario.

Una regla similar vale para el operador $\overset{\text{dijo}}{\Longrightarrow}$:

$$\frac{P \overset{\text{cree}}{\Longrightarrow} Q \overset{\text{dijo}}{\Longrightarrow} (X,Y)}{P \overset{\text{cree}}{\Longrightarrow} Q \overset{\text{dijo}}{\Longrightarrow} X} \tag{R6}$$

aunque no es cierto que de $P \overset{\text{cree}}{\Longrightarrow} Q \overset{\text{dijo}}{\Longrightarrow} X$ y $P \overset{\text{cree}}{\Longrightarrow} Q \overset{\text{dijo}}{\Longrightarrow} Y$ se deduzca que $P \overset{\text{cree}}{\Longrightarrow} Q \overset{\text{dijo}}{\Longrightarrow} (X,Y)$ porque ambas premisas podrían no haberse dicho al mismo momento.

Algo similar vale para los frescos:

$$\frac{P \overset{\text{cree}}{\Longrightarrow} \#(X)}{P \overset{\text{cree}}{\Longrightarrow} \#(X,Y)} \tag{R7}$$

También se podrían agregar reglas que establecieran que si X es fresco entonces también lo es $\{X\}_K$.

Además tenemos reglas que determinan que si un sujeto ve una sentencia entonces puede ver sus componentes siempre que tenga las claves apropiadas.

$$\frac{P \overset{\text{ve}}{\Longrightarrow} (X,Y)}{P \overset{\text{ve}}{\Longrightarrow} X} \qquad \frac{P \overset{\text{cree}}{\Longrightarrow} Q \overset{K}{\leftrightarrow} P, \qquad P \overset{\text{ve}}{\Longrightarrow} \{X\}_K}{P \overset{\text{ve}}{\Longrightarrow} X} \tag{R8}$$

$$\frac{P \overset{\text{cree}}{\Longrightarrow} \overset{K}{\mapsto} P, \qquad P \overset{\text{ve}}{\Longrightarrow} \{X\}_K}{P \overset{\text{ve}}{\Longrightarrow} X} \qquad \frac{P \overset{\text{cree}}{\Longrightarrow} \overset{K}{\mapsto} Q, \qquad P \overset{\text{ve}}{\Longrightarrow} \{X\}_{K^{-1}}}{P \overset{\text{ve}}{\Longrightarrow} X} \tag{R9}$$

La primera regla de la segunda línea se basa en el hecho de que si P cree que su clave pública es K entonces también cree que su clave privada es K^{-1} y por lo tanto puede desencriptar $\{X\}_K$. Recordar que las sentencias de la forma $\{X\}_K$ son en realidad $\{X\}_K \,\mathsf{from}\, R$ y siempre se pide $P \neq R$.

Finalmente damos reglas que indican que las claves simétricas se pueden usar en ambas 'direcciones'.

$$\frac{P \overset{\text{cree}}{\Longrightarrow} R \overset{K}{\leftrightarrow} R'}{P \overset{\text{cree}}{\Longrightarrow} R' \overset{K}{\leftrightarrow} R} \qquad \frac{P \overset{\text{cree}}{\Longrightarrow} Q \overset{\text{cree}}{\Longrightarrow} R \overset{K}{\leftrightarrow} R'}{P \overset{\text{cree}}{\Longrightarrow} Q \overset{\text{cree}}{\Longrightarrow} R' \overset{K}{\leftrightarrow} R} \tag{R10}$$

Metodología para analizar protocolos. Para analizar un protocolo criptográfico usando BAN se siguen los siguientes pasos (todos los predicados que se mencionan son predicados BAN):

1. Se especifica el protocolo partiendo de una descripción como la que hicimos para NS (Sección 4.4.4) y SSL (Sección 4.4.5)[109]. Cada mensaje que sea relevante desde el punto de vista de la seguridad se traduce a una o más sentencias BAN. En particular, los mensajes legibles no se traducen pues no aportan a la seguridad.

   Este es, tal vez, el paso más complejo de la metodología BAN porque requiere comprender la esencia del protocolo y deducir algunas propiedades que no están explícitas. En general, primero se debe comprender el protocolo intuitivamente como un todo. No se puede especificar el protocolo traduciendo cada mensaje a BAN aisladamente. La traducción de un mensaje a BAN solo se puede hacer si se comprende el protocolo como un todo.

   En general, el mensaje m se puede traducir a la sentencia BAN X si siempre que el destinatario recibe m él puede deducir que el remitente creía en X cuando emitió m.

2. Se establecen los supuestos o hipótesis iniciales del protocolo. Es decir las proposiciones que se consideran válidas justo antes de que se inicie la ejecución del protocolo. Por ejemplo, en NS sabemos que S tiene autoridad sobre las claves que usarán A y B, cosa que no está dicha en los mensajes del protocolo sino en su contexto. Esto se formaliza de la siguiente forma:

   $$A \overset{\text{cree}}{\Longrightarrow} S \overset{\text{tjs}}{\Longrightarrow} A \overset{K_{ab}}{\leftrightarrow} B, \qquad B \overset{\text{cree}}{\Longrightarrow} S \overset{\text{tjs}}{\Longrightarrow} A \overset{K_{ab}}{\leftrightarrow} B$$

   En general las hipótesis establecen cuáles claves compartidas existen, cuáles sujetos han generado frescos, y cuáles sujetos tienen autoridad o jurisdicción sobre ciertas cuestiones. En la mayoría de los casos estas hipótesis quedan más o menos determinadas por el tipo de protocolo (de clave pública o privada, para negociar una clave, para autenticar a algún sujeto, etc.). Sin embargo, en ciertos casos los diseñadores originales, posiblemente sin darse cuenta, han efectuado otras suposiciones. En ocasiones estas suposiciones más ocultas dan lugar a vulnerabilidades.

3. Se asocian predicados que valen antes y después de cada mensaje de la especificación del protocolo. Aquí se aplica la idea clásica de la lógica de Hoare. En otras palabras se *anota* el protocolo.

   Más precisamente, un protocolo es una secuencia de sentencias *send* de la forma $P \to Q : X$ con $P \neq Q$. Entonces, anotar el protocolo consiste en asociar una secuencia de predicados antes y después de cada sentencia del protocolo; los predicados antes de la primera sentencia son los supuestos (paso 2) y los predicados después de la última sentencia son las conclusiones (paso 4). Los predicados BAN son conjunciones de fórmulas de la forma $P \overset{\text{cree}}{\Longrightarrow} X$ y $P \overset{\text{ve}}{\Longrightarrow} X$. En particular si el predicado Y vale antes de la sentencia (ya formalizada) $P \to Q : X$, entonces la anotación $\{Y\}P \to Q : X\{Y, Q \overset{\text{ve}}{\Longrightarrow} X\}$ es correcta.

4. Se aplican las reglas de inferencia para determinar las creencias de los participantes al finalizar la ejecución del protocolo.

Objetivos de seguridad de un protocolo. El último paso de la metodología BAN nos sugiere abordar el problema de determinar cuáles son los objetivos de un protocolo criptográfico en términos de BAN. Es decir, ¿cuáles deberían ser las conclusiones (formales) que obtengamos en

[109] Los autores de BAN a este paso lo llaman *idealización* del protocolo.

el paso 4 que nos permitan tener confianza en que el protocolo es seguro? En otras palabras, ¿cómo sabemos que un protocolo criptográfico es seguro? ¿Qué propiedades deben verificar estos protocolos?

Aunque estas propiedades son materia de debate existen algunas casi ineludibles u obvias. En general el objetivo más buscado es compartir una clave simétrica; o sea, la conclusión del paso 4 debería ser:

$$A \overset{\text{cree}}{\Longrightarrow} A \overset{K}{\leftrightarrow} B, \qquad B \overset{\text{cree}}{\Longrightarrow} A \overset{K}{\leftrightarrow} B$$

Algunos protocolos garantizan, además:

$$A \overset{\text{cree}}{\Longrightarrow} B \overset{\text{cree}}{\Longrightarrow} A \overset{K}{\leftrightarrow} B, \qquad B \overset{\text{cree}}{\Longrightarrow} A \overset{\text{cree}}{\Longrightarrow} A \overset{K}{\leftrightarrow} B$$

Otros protocolos suelen garantizar menos:

$$A \overset{\text{cree}}{\Longrightarrow} B \overset{\text{cree}}{\Longrightarrow} X$$

para algún X, lo que dice que A cree que B existe en el presente (seguramente porque ha enviado mensajes recientes).

En el caso de protocolos que usan criptografía asimétrica, el objetivo podría ser que A tenga certeza de que K es la clave pública de B:

$$A \overset{\text{cree}}{\Longrightarrow} \overset{K}{\mapsto} B$$

Aplicación de BAN al protocolo NS Con todos estos elementos podemos aplicar BAN al análisis del protocolo NS (ver Sección 4.4.4). Entonces, siguiendo la metodología de cuatro pasos comentada más arriba, comenzamos por formalizar el protocolo en BAN:

M2. S $\rightarrow$ A: $\{N_a, A \overset{K_{ab}}{\leftrightarrow} B, \#(A \overset{K_{ab}}{\leftrightarrow} B), \{A \overset{K_{ab}}{\leftrightarrow} B\}_{K_{bs}}\}_{K_{as}}$

M3. A $\rightarrow$ B: $\{A \overset{K_{ab}}{\leftrightarrow} B\}_{K_{bs}}$

M4. B $\rightarrow$ A: $\{N_b, A \overset{K_{ab}}{\leftrightarrow} B\}_{K_{ab}}$ from B

M5. A $\rightarrow$ B: $\{N_b, A \overset{K_{ab}}{\leftrightarrow} B\}_{K_{ab}}$ from A

El primer mensaje del protocolo no se formaliza porque, como dijimos más arriba, los mensajes de texto legible no aportan a la seguridad. Luego, por ejemplo, vemos que se formaliza el hecho de que K_{ab} es una clave compartida entre A y B. Ciertamente decir que K_{ab} es una clave compartida entre A y B solo por el hecho de que su subíndice es ab no constituye una sentencia formal. Por el contrario, la construcción $A \overset{K_{ab}}{\leftrightarrow} B$ es una sentencia con una semántica formal (ver [4, sección 13]). Además se incluye el hecho de que K_{ab} es una clave nueva ($\#(A \overset{K_{ab}}{\leftrightarrow} B)$). Finalmente, la respuesta de A al desafío planteado por B (i.e. en M5) se simplifica a N_b puesto que retornar $N_b - 1$ es solo a los efectos de evitar la repetición (ciega) del desafío. Esto se logra agregando el modificador from que permite distinguir ambos mensajes. La inclusión de $A \overset{K_{ab}}{\leftrightarrow} B$ en los dos últimos mensajes captura el hecho (implícito en la descripción informal) de que A y B creen que la clave es solo para ellos.

Ahora podemos formalizar las hipótesis del protocolo (paso 2).

$$A \overset{\text{cree}}{\Longrightarrow} A \overset{K_{as}}{\leftrightarrow} S, \qquad B \overset{\text{cree}}{\Longrightarrow} B \overset{K_{bs}}{\leftrightarrow} S, \qquad S \overset{\text{cree}}{\Longrightarrow} A \overset{K_{as}}{\leftrightarrow} S, \qquad S \overset{\text{cree}}{\Longrightarrow} B \overset{K_{bs}}{\leftrightarrow} S, \qquad S \overset{\text{cree}}{\Longrightarrow} A \overset{K_{ab}}{\leftrightarrow} B, \tag{H1}$$

$$A \overset{\text{cree}}{\Longrightarrow} S \overset{\text{tjs}}{\Longrightarrow} A \overset{K_{ab}}{\leftrightarrow} B, \qquad B \overset{\text{cree}}{\Longrightarrow} S \overset{\text{tjs}}{\Longrightarrow} A \overset{K_{ab}}{\leftrightarrow} B, \qquad A \overset{\text{cree}}{\Longrightarrow} S \overset{\text{tjs}}{\Longrightarrow} \#(A \overset{K_{ab}}{\leftrightarrow} B), \tag{H2}$$

$$A \overset{\text{cree}}{\Longrightarrow} \#(N_a), \qquad B \overset{\text{cree}}{\Longrightarrow} \#(N_b), \qquad S \overset{\text{cree}}{\Longrightarrow} \#(A \overset{K_{ab}}{\leftrightarrow} B), \qquad \boxed{B \overset{\text{cree}}{\Longrightarrow} \#(A \overset{K_{ab}}{\leftrightarrow} B) \quad \text{(H34)}} \tag{H3}$$

La mayoría de estas hipótesis son obvias dados el contexto y la definición del protocolo. Notar que A cree que S tiene jurisdicción sobre la frescura de K_{ab}. B no puede asumir lo mismo porque no interactúa con S. Resaltamos la última hipótesis, (H34), porque sin ella no se puede demostrar una de las propiedades básicas del protocolo (i.e. $B \overset{\text{cree}}{\Longrightarrow} A \overset{K_{ab}}{\leftrightarrow} B$) lo que en definitiva permite el ataque visto en la Sección 4.4.4. Es decir, los autores del protocolo no se dieron cuenta que esta hipótesis era necesaria para garantizar la seguridad del protocolo y por lo tanto su diseño no lo refleja.

Según los pasos 3 y 4 de la metodología debemos anotar cada mensaje de la formalización del protocolo y aplicar reglas de inferencia para deducir propiedades. De esta forma vamos a deducir dos propiedades: (a) $A \overset{\text{cree}}{\Longrightarrow} \#(A \overset{K_{ab}}{\leftrightarrow} B)$; y (b) para demostrar que sin la hipótesis (H34) no es posible deducir $B \overset{\text{cree}}{\Longrightarrow} A \overset{K_{ab}}{\leftrightarrow} B$. Notar que, como vimos en el apartado anterior, (b) es una de las propiedades típicas de los protocolos criptográficos.

Empezamos con (a). La formalización comienza con el mensaje M2 que pueden ver más arriba. Antes del mensaje M2 valen todas las hipótesis sobre el protocolo que acabamos de mencionar. Luego de M2 A ve el contenido del mensaje que le envía S y por lo tanto podemos agregar el predicado:

$$A \overset{\text{ve}}{\Longrightarrow} \{N_a, A \overset{K_{ab}}{\leftrightarrow} B, \#(A \overset{K_{ab}}{\leftrightarrow} B), \{A \overset{K_{ab}}{\leftrightarrow} B\}_{K_{bs}}\}_{K_{as}} \tag{6}$$

del cual, considerando la primera hipótesis de (H1) y la regla (R1), podemos deducir:

$$A \overset{\text{cree}}{\Longrightarrow} S \overset{\text{dijo}}{\Longrightarrow} (N_a, A \overset{K_{ab}}{\leftrightarrow} B, \#(A \overset{K_{ab}}{\leftrightarrow} B), \{A \overset{K_{ab}}{\leftrightarrow} B\}_{K_{bs}}) \tag{7}$$

del cual, considerando la primera hipótesis de (H3) y la regla (R7), podemos deducir:

$$A \overset{\text{cree}}{\Longrightarrow} \#(N_a, \#(A \overset{K_{ab}}{\leftrightarrow} B)) \tag{8}$$

A su vez también de (7) pero aplicando la regla (R6) podemos deducir:

$$A \overset{\text{cree}}{\Longrightarrow} S \overset{\text{dijo}}{\Longrightarrow} (N_a, \#(A \overset{K_{ab}}{\leftrightarrow} B)) \tag{9}$$

del cual, considerando (8) y la regla (R3), podemos deducir:

$$A \overset{\text{cree}}{\Longrightarrow} S \overset{\text{cree}}{\Longrightarrow} (N_a, \#(A \overset{K_{ab}}{\leftrightarrow} B)) \tag{10}$$

del cual, aplicando la tercera regla de (R5), podemos deducir:

$$A \overset{\text{cree}}{\Longrightarrow} S \overset{\text{cree}}{\Longrightarrow} \#(A \overset{K_{ab}}{\leftrightarrow} B) \tag{11}$$

del cual, considerando la tercera hipótesis de (H2) y aplicando la regla (R4), podemos deducir (a).

Ahora vamos a demostrar (b). Iniciamos como cuando demostramos (a), es decir anotamos M3 con:

$$B \overset{\text{ve}}{\Longrightarrow} \{A \overset{K_{ab}}{\leftrightarrow} B\}_{K_{bs}} \tag{12}$$

de cual, considerando la segunda hipótesis de (H1) y aplicando la regla (R1), podemos deducir:

$$B \overset{\text{cree}}{\Longrightarrow} S \overset{\text{dijo}}{\Longrightarrow} A \overset{K_{ab}}{\leftrightarrow} B \tag{13}$$

Pero aquí comienzan los problemas porque de este $\overset{\text{dijo}}{\Longrightarrow} A \overset{K_{ab}}{\leftrightarrow} B$ deberíamos pasar a $\overset{\text{cree}}{\Longrightarrow} A \overset{K_{ab}}{\leftrightarrow} B$. El problema es que la única regla, (R3), que a partir de $\overset{\text{dijo}}{\Longrightarrow} X$ permite deducir $\overset{\text{cree}}{\Longrightarrow} X$ requiere que el sujeto crea que X es fresco. En este caso requiere que B crea que $A \overset{K_{ab}}{\leftrightarrow} B$ es fresco; o sea $B \overset{\text{cree}}{\Longrightarrow} \#(A \overset{K_{ab}}{\leftrightarrow} B)$, que es precisamente (H34). En resumen, si no se asume (H34) no se puede demostrar (b). Tampoco podemos demostrar (H34) como hicimos con (a) (notar que es el mismo predicado pero con A como sujeto) porque M2 le permite a A deducir cosas que B no puede puesto que él no ve ese mensaje (comprobar esto como ejercicio). De hecho, la solución planteada por Needham y Schroeder que vimos en la Sección 4.4.4 agrega mensajes que le permiten a B comprobar la frescura K_{ab} (ejercicio).

Ahora, asumiendo (H34) veamos cómo se demuestra (b). Entonces, considerando (H34) y aplicando la regla (R3), podemos deducir:

$$B \overset{\text{cree}}{\Longrightarrow} S \overset{\text{cree}}{\Longrightarrow} A \overset{K_{ab}}{\leftrightarrow} B \tag{14}$$

del cual, considerando la segunda hipótesis de (H2) y aplicando la regla (R4), podemos deducir (b).

Dejamos como ejercicio demostrar que $A \overset{\text{cree}}{\Longrightarrow} A \overset{K_{ab}}{\leftrightarrow} B$, lo que en este punto debería resultarles muy simple.

Claramente, aunque aquí hayamos procedido a mano con las demostraciones, BAN puede ser codificada como una teoría en asistentes de prueba tales como Coq e Isabelle/HOL, lo que permite automatizar grandes porciones de la mayoría de las pruebas (e incluso, en algunos casos, automatizarlas completamente).

Esta sección debe completarse leyendo [4, secciones 1-3, 5 y 12-13]

Referencias

[1] Aleph One. Smashing the stack for fun and profit. *Phrack*, 7(49), November 1996. http://www.phrack.com/issues.html?issue=49&id=14.

[2] D. Elliot Bell and Leonard LaPadula. Secure computer systems: Mathematical foundations. MTR 2547, The MITRE Corporation, May 1973.

[3] D. Elliot Bell and Leonard LaPadula. Secure computer systems: Mathematical model. ESD-TR 73-278, The MITRE Corporation, November 1973.

[4] Michael Burrows, Martín Abadi, and Reger Needham. A logic of authentication. Technical Report no. 39, Digital Systems Research Center, Digital Equipment Corporation, 1989. http://www.hpl.hp.com/techreports/Compaq-DEC/SRC-RR-39.pdf.

[5] Michael Burrows, Martín Abadi, and Roger M. Needham. A logic of authentication. *ACM Trans. Comput. Syst.*, 8(1):18–36, 1990.

[6] Shawn A. Butler. Security attribute evaluation method: a cost-benefit approach. In Will Tracz, Michal Young, and Jeff Magee, editors, *Proceedings of the 24th International Conference on Software Engineering, ICSE 2002, 19-25 May 2002, Orlando, Florida, USA*, pages 232–240. ACM, 2002.

[7] Michael R. Clarkson and Fred B. Schneider. Hyperproperties. *Journal of Computer Security*, 18(6):1157–1210, 2010.

[8] Maximiliano Cristiá and Pablo Mata. Runtime enforcement of noninterference by duplicating processes and their memories. In *Workshop de Seguridad Informática WSEGI 2009*, Mar del Plata, Argentina, 2009. SADIO.

[9] Maximiliano Cristiá and Gianfranco Rossi. Automated proof of Bell-LaPadula security properties. *J. Autom. Reason.*, 65(4):463–478, 2021.

[10] Dorothy E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5):236–243, 1976.

[11] Dominique Devriese and Frank Piessens. Noninterference through secure multi-execution. In *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA*, pages 109–124. IEEE Computer Society, 2010.

[12] Sergio Ferrari. La Crypyo CIA desata un terremoto. *El Cohete a la Luna*, 2020. https://www.elcohetealaluna.com/la-crypto-cia-desata-un-terremoto/.

[13] A. Freier, P. Karlton, and P. Kocher. The secure sockets layer (ssl) protocol version 3.0. RFC 6101, RFC Editor, August 2011.

[14] Morrie Gasser. *Building a Secure Computer System*. Van Nostrand Reinhold Co., New York, NY, USA, 1988.

[15] Joseph A. Goguen and José Meseguer. Security policies and security models. In *1982 IEEE Symposium on Security and Privacy, Oakland, CA, USA, April 26-28, 1982*, pages 11–20. IEEE Computer Society, 1982.

[16] Lawrence A. Gordon and Martin P. Loeb. The economics of information security investment. *ACM Transactions on Information and System Security*, 5(4):438–457, November 2002.

[17] Daryl McCullough. Specifications for multi-level security and a hook-up property. In *Proceedings of the 1987 IEEE Symposium on Security and Privacy, Oakland, California, USA, April 27-29, 1987*, pages 161–166. IEEE Computer Society, 1987.

[18] Daryl McCullough. Noninterference and the composability of security properties. In *IEEE Symposium on Security and Privacy*, pages 177–186, Oakland, CA, April 1988.

[19] Greg Miller. The intelligence coup of the century. *The Whashington Post*, 2020. https://www.washingtonpost.com/graphics/2020/world/national-security/cia-crypto-encryption-machines-espionage/.

[20] National Institute of Standards and Technology. *FIPS PUB 81: DES modes of operation*. December 1980. https://csrc.nist.gov/csrc/media/publications/fips/81/archive/1980-12-02/documents/fips81.pdf.

[21] National Institute of Standards and Technology. *FIPS PUB 46-3: Data Encryption Standard (DES)*. October 1999. Supersedes FIPS 46-2, https://csrc.nist.gov/csrc/media/publications/fips/46/3/archive/1999-10-25/documents/fips46-3.pdf.

[22] George C. Necula. Proof-carrying code. In Peter Lee, Fritz Henglein, and Neil D. Jones, editors, *Conference Record of POPL'97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, Paris, France, 15-17 January 1997*, pages 106–119. ACM Press, 1997.

[23] George C. Necula and Peter Lee. Research on proof-carrying code for untrusted-code security. In *1997 IEEE Symposium on Security and Privacy, May 4-7, 1997, Oakland, CA, USA*, page 204. IEEE Computer Society, 1997.

[24] Roger M. Needham and Michael D. Schroeder. Authentication revisited. *Operating Systems Review*, 21(1):7, 1987.

[25] OWASP. Sql injection. https://www.owasp.org/index.php/SQL_Injection.

[26] Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, 1978.

[27] Chris Salter, O. Sami Saydjari, Bruce Schneier, and Jim Wallner. Toward a secure system engineering methodology. In *Proceedings of the New Security Paradigms Workshop*, pages 2–10, Charlottsville, VA, September 1998.

[28] Bruce Schneier. *Secrets and Lies: digital security in a networked world*. Wiley Computer Publishing, 2000.

[29] Ian Sutherland and et. al. Romulus: A computer security properties modeling enviroment. The theory of security. Technical Report RL-TR-91-36 Vol IIa, Rome Laboratory, April 1991.

[30] Hans van Ditmarsch, Joseph Y. Halpern, Wiebe van der Hoek, and Barteld P. Kooi. An introduction to logics of knowledge and belief. *CoRR*, abs/1503.00806, 2015.

[31] Guido Vassallo. *Las Malvinas y el Plan Cóndor, en el centro de un escándalo mundial de espionaje*, 2020.

[32] Dennis M. Volpano, Cynthia E. Irvine, and Geoffrey Smith. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2/3):167–188, 1996.