Universidad Nacional de Rosario

Facultad de Ciencias Exactas, Ingeniería y Agrimensura

Licenciatura en Ciencias de la Computación

Resumen

Seguridad Informática

Arroyo Joaquín

2024

1. Introducción

Definición de Seguridad Informática.

A partir de esto definimos **Confidencialidad**, **Integridad** y **Disponibilidad** como los tres pilares de la Seguridad Informática. Notamos que Integridad impone 2 restricciones y las otras dos propiedades solo una. Además notamos la diferencia natural entre **Confidencialidad** e **Integridad** contra **Disponibilidad**, a los dos primeros se los puede controlar a partir de retricciones, al último todo lo contrario.

Notamos que en la Seguridad Informática implícitamente trabajamos con un entorno **hostil** (enemigos, amenazas, ataques, etc.) a diferencia de la Ingeniería de Software donde suponemos (implícitamente) que el entorno es benigno.

La existencia de este entorno **hostil** implica diferencias entre **Safety** y **Security**, son propiedades distintas. Esto no quiere decir que no existan propiedades de **Security** que son de **Safety**, pero generalmente no pasa. Un ejemplo es el caso del desbordamiento de arreglo, el código que permite el ataque puede ser verificado formalmente, por lo que sería *safety-correcto*, pero permite un ataque de *buffer-overflow* por lo que **no** es *security-correcto*.

Más allá de estas diferencias entre **safety** y **security**, siguiendo los lineamientos de la Ingeniería de Software se logran desplegar sistemas con seguridad aceptable.

Además de los tres pilares de la Seguridad Informática, **Confidencialidad**, **Integridad** y **Disponibilidad**, agregamos otros tres conceptos: **Autenticación**, **Autorización** y **Auditoría**. Damos sus tres definiciones.

Damos la categorización de **Seguridad** en **Seguridad Interna** y **Seguridad Externa** y explicamos cada una.

Damos la **Hipótesis de Seguridad** bajo la cuál se trabaja en la Seguridad Informática. Esta considera a los usuarios de la organización como usuarios **de confianza**.

> **Hipótesis de Seguridad.** *La organización confía en que los usuarios autorizados no divulgarán, eliminarán ni harán indisponible la información a la cuál tienen acceso.*

A contracara de esto, se supone que todo usuario no autorizado hará lo posible para acceder a información a la cuál no tiene acceso y por lo tanto la divulgará, la eliminará y la hará indisponible.

Damos una categorización para el Software en **Confiable**, **Benigno** y **Hostil**. Explicamos cada uno, y notamos que enrealidad los últimos dos son incluidos en la categoría de **No Confiable**. Esto ya que en general, no es posible determinar que un software benigno con errores, pueda ser usado para realizar ataques.

Se nota que la forma más letal de ataque en la Seguridad Informática son los **Caballos de Troya**

Un **Caballos de Troya** es un software que provee funcionalidad útiles, pero que contiene el código para realizar un ataque hacia el sistema de cómputo el cuál lo ejecuta.

El problema con esto es que no es **decidible** saber si un software es un **Caballo de Troya** o no, por lo que no se debería permitir la instalación de software no confiable.

Igualmente existen contextos en los cuáles pensar que todo el software no confiable de una computadora es un Caballo de Troya, en otros contextos no tanto.

En la actualidad ningún SO puede detectar si un software es un **Caballo de Troya** o no, por lo tanto no pueden detener estos ataques a la confidencialidad.

Notar que si se rompe la **Confidencialidad** de un sistema, eventualmente se romperá tanto la **Integridad** como la **Disponibilidad**.

En consecuencia, los ataques por medio de Caballos de Troya a la Disponibilidad, están en la base de la Seguridad Informática.

2. Confidencialidad en Sistemas de Cómputo

Vimos que la seguridad de los SOs modernos no es suficiente para detener un ataque mediante un Caballo de Troya a la confidencialidad del sistema. Esto sucede principalmente por lo siguiente:

1. El atacante desarrolla un Caballo de Troya T
2. Logra que T se instale en algún sistema de cómputo
3. Un usuario con acceso a un **secreto** U ejecuta T
4. T accede al **secreto** y envía la información al atacante.

Notamos que esto es posible por la forma de como funcionan los permisos en los SOs. En este caso, como T fue ejecutado por U, y este tenía permisos para acceder al **secreto**, entonces T heredó dichos permisos, por lo que tuvo acceso al **secreto**.

A su vez, esto es posible por el tipo de control de acceso que implementan los SOs, el cuál es el *Control de Acceso Discrecional* (DAC). Este control de acceso, permite a los usuarios modificar sus permisos *a discreción*, y los de los objetos que crean/modifican.

Explica como se implementa esto, a partir de una *Tabla de Control de Acceso*, donde cada fila es un **objeto** y cada columna un **sujeto**, en cada entrada de la matriz, están los permisos.

Otra forma de implementar esto es a partir de una *Lista de Control de Acceso*, es lo mismo que una fila de la tabla, pero sin entradas vacías. Y se almacena en los objetos.

2.1. Seguridad Multi Nivel

Política de **Seguridad Multi Nivel** (MLS). Creada y utilizada por el DoD de EEUU. Lo más importante es:

- Forma de las **clases de acceso** (n, C) donde n es el nivel de acceso, y C es un conjunto de categorías denominado **departamentos**.
- La propiedad que impone esta política sobre la lectura de información:

  *Un usuario **U** con clase de acceso (n_1, C_1) puede acceder a la información **I** con con clase de acceso (n_2, C_2) sii la clase de acceso de **U domina** a la clase de acceso de **I***

  Cuando hablamos de dominación nos referimos a

  $$(n_1, C_1) \textit{ dominates } (n_2, C_2) \textit{ sii } n_2 \leq n_1 \wedge C_2 \subseteq C_1$$

- El control de acceso es Obligatorio (MAC). Esto debido a que cuando se quiere aumentar la clase de acceso de un sujeto, este pasa por un proceso minucioso en el cuál se le realizan interrogatorios, entrevistas, se buscan sus antecedentes, etc.

  Y lo mismo para cuando se quiere bajar la clase de acceso de una información, además de pasar por un proceso minucioso, también deben pasar una cantidad de años establecida (gralmente. 50).

  Es por esto que vemos claramente, que el control de acceso no queda a cargo de los usuarios, si no a cargo de un grupo de personas y de un proceso bien definido y minucioso.

2.2. TCB

Se menciona que uno de los problemas de la Seguridad Informática es definir una Arquitectura que minimice el código y lo simplifique. Para resolver (o sanar) este problema, tenemos el concepto de **TCB** (base de cómputo confiable)

TCB. *Es el conjunto de componentes de Software y Hardware responsables de la seguridad del sistema.*

Esta debe ser desarrollada por personal de máxima confianza, ya que cualquier error, o código hostil, puede implicar una violación a la seguridad del sistema.

Además se espera que sea mínima y simple para que pueda ser verificada formalmente.

Debe intermediar todos los accesos de sujetos a objetos.

Una forma probada de implementar la **TCB** es dividirla en **Núcleo de Seguridad** y **Aplicaciones Confiables**.

Dar definición de **Camino Confiable**.

Se menciona que la **política de seguridad** corresponde a los **Requerimientos**, y el **modelo de seguridad** a la **Especificación**. Vemos tres modelos de seguridad.

- **Bell-LaPadula** (BLP)
- **No Interferencia**
- **Multi-ejecución segura**

2.3. BLP

BLP fue desarrollado por Bell y LaPadula en 1972 para especificar llamadas al sistema de un SO previo a UNIX. Utilizaron una matemática simple pero ad-hoc. Definieron a estas llamadas como operaciones/transiciones, y por último definieron dos invariantes los cuales son necesarios para que el modelo implemente MLS. Además el modelo implementa DAC.

Nosotros vimos las operaciones de *write* y *read* y los dos invariantes: **Condición de Seguridad** y la **Propiedad Estrella**

La **Condición de Seguridad** es básicamente la condición que impone MLS sobre la lectura de información pero formalizada.

Y la **Propiedad Estrella** fue necesaria debido a que el sistema de cómputo especificado puede presentar Caballos de Troya, y con esta propiedad se evita su función principal: Desclasificar un secreto.

La propiedad estrella trabaja de la siguiente forma:

Un usuario puede abrir un objeto con clase de acceso c, en modo escritura, siempre y cuando para todos los objetos que tenga abierto en modo lectura, c domine a todas sus clases de acceso. Con esto evitamos el *write-down*, es decir, que se escriba información clasificada, en archivos con menor nivel de seguridad.

Los problemas de BLP son tres

1. Es **innecesariamente restrictivo**. Esto principalmente debido a la propiedad estrella. Se prohíbe utilizar ciertos objetos debido a esta propiedad, cuando no necesariamente estos desclasificaran información
2. **Escalada de Nivel**. Esto debido a la propiedad estrella, los usuarios quedan asignados al supremo de las clases de acceso de todos los objetos que abrió en modo lectura. Obviamente, esto tiende al supremo de todas las clases de acceso, por lo que el usuario termina teniendo acceso a poca información. Para solucionar esto, deberíamos borrar el usuario para que comience con el mínimo de las clases de acceso, pero claramente esto no es una solución
3. Es muy dificil **determinar todos los objetos del sistema**, por lo que un contenedor de información confidencial, que no fue catalo

2.4. No Interferencia

El segundo modelo es **No Interferencia**. Fue desarrollado por Goguen y Meseguer en 1982. Hoy en día **No Interferencia** se la conoce como **la** definición de confidencialidad en la Seguridad Informática.

Este simple concepto trabaja de la siguiente forma:

Sin perdida de la generalidad, podemos suponer que tenemos dos clases de acceso **L** y **H**. Supongamos que tenemos dos grupos de usuarios G_L yG_H. Decimos que G_H **no interfiere** G_L, si toda salida vista por los usuarios de G_L es la misma si los usuarios de G_H realizaron acciones sobre el sistema, como si no lo hicieron.

Para realizar esta definición formalmente, definen un **sistema de capacidades** como un sistema cuyas acciones permitidas varían en el tiempo.

Un **sistema de capacidades** M contiene:

1. Un conjunto de **Usuarios** U
2. Un conjunto de **Estados** E
3. Un conjunto de **transiciones de estado** T
4. Un conjunto de **salidas** del sistema Out
5. Un conjunto de **tablas de capacidades C**
6. Un conjunto de **comandos de capacidades** P
7. Una función $O : U \times E \times C \to Out$. Devuelve las salidas posibles.
8. Una función $T : U \times E \times T \times C \to E$. Transiciona de estado.
9. Una función $P : U \times P \times C \to C$. Cambia la tabla de capacidades.
10. Dos constantes s_0 y c_0 que son el estado inicial, y la tabla de capacidades inicial respectivamente.

Además se define una función E la cuál engloba las transiciones de estado y los cambios en la tabla de capacidades.

$$E : U \times E \times C \times (T \cup P) \to E \times C$$

Luego definimos otra función E que generalizamos para secuencias de comandos de entrada, pueden ser de T o de P.

$$E : U \times E \times C \times seq(T \cup P) \to E \times C$$

Por último definimos, $[\![w]\!]_u$ donde w es una secuencia de $(T \cup P)$ y $u \in U$. $[\![w]\!]_u$ quiere decir, todo lo que puede "ver" el usuario u luego de que se ejecutó la secuencia w.

A partir de estas definiciones, se define formalmente la **no interferencia**.

Otros conceptos importantes son:

- U_{+x} que significa: $\forall u \in U, sc(u)$ *dominates* x ($sc()$ es clase de acceso y x es una clase de acceso)
- Decimos que un grupo de usuarios $G_1 \subseteq U$ es **invisible** para un grupo de usuarios G_2 sii G_1 no interfiere con G_2
- Decimos que un sistema de capacidades M es **MLS** sii para toda clase de acceso x, U_{+x} es un grupo de usuarios **invisible**.

2.4.1. Como probamos la seguridad de un sistema?

Luego de esta definición se habla sobre que aún cuando estamos seguros de tener una buena definición de seguridad (en esta sección *confidencialidad*), no es fácil determinar que un sistema cumple con ella. Esto está relacionado con el hecho de que *Safety* Y *Security* son propiedades distintas, como ya vimos, no son disjuntas, pero son distintas.

La **no interferencia** no es de *Safety*, es más, no es una propiedad en la forma que Alpern y Schneider definieron a las propiedades. Según ellos, una propiedad depende de una ejecución, y no interferencia depende de un conjunto de ejecuciones.

A partir de esto, Clarkson y Schneider, definen las *hiperpropiedades*, propiedades que dependen de múltiples ejecuciones. Bajo esta definición, se puede definir a la **no interferencia**. Además definieron *hypersafety* e *hyperliveness* como generalizaciones naturales de *safety* y vitalidad.

También muestran que el refinamiento preserva *safety*, pero si puede invalidad una hiperpropiedad.

No pueden demostrar que **no interferencia** es de *hypersafety* pero por un tecnicismo, según ellos sería lógico que lo fuera.

Y por último, muestran que el refinamiento preserva *hypersafety*, por lo que sería lógico que *no necesariamente* preserve **no interferencia** (Dado que no es de *hypersafety*).

Esto tampoco significa que la **no interferencia** nunca sea de *safety*, existen definiciones, menos generales, de **no interferencia** a las cuáles se las puede expresar como propiedades de *safety*. Usualmente estas se expresan a partir de un sistema de tipos, por lo que la verificación es básicamente un chequeo de tipos, que es de *safety*. Estas definiciones igualmente generalmente son innecesariamente restrictivas.

2.4.2. Canales Encubiertos

Concepto de **canal encubierto**. Canales que pueden transmitir información, pero no fueron creados para ello.

Este concepto se da debido a que un Caballo de Troya puede aprovechar la existencia de este tipo de canales en un SO.

Estos canales tienen un ancho de banda definido por una unidad de tiempo, podemos pensar que si estos canales tienen anchos de banda *absurdos*, como por ejemplo **1 bit**, se los puede desestimar. Pero esto no es así, debido a que la idea detrás del uso de estos canales es enviar información (no de gran tamaño) al atacante para que este pueda acceder al sistema, por ejemplo, una credencial. Una credencial tranquilamente se podría enviar por un canal encubierto con ancho de banda de **1 bit** $\times$ **min**. Es más, hasta el tiempo podría ser mayor, 1 semana, 1 mes, 1 año quizás ya no tiene mucho sentido.

De todas formas, estos canales, como todo canal, tienen presencia de **ruido**. Es decir, lo que se envía por el canal no siempre llega a destino. Puede parecer que esto nos da un respiro, pero simplemente con reintentar enviar la información, sería suficiente para el Caballo de Troya y el espía.

Por lo que vimos que, los canales de 1 bit, y de 1 bit ruidoso son intolerables.

También existen los canales de medio bit, en los cuales el Caballo de Troya solo puede enviar un 1 o un 0. Podemos pensar que no tiene sentido preocuparnos por este tipo de canales, pero juntando dos canales de medio bit, obtenemos un canal de 1 bit, y ya vimos que estos son intolerables.

2.4.3. Conclusiones finales sobre No Interferencia

Se ve el **determinismo**, **interrupciones** y **concurrencia** sobre **no interferencia**.

Este modelo se basa en que el sistema es: **determinista** y **no admite interrupciones**.

Vemos que la salida del modelo de **no interferencia** es función de la entrada, por lo que claramente se espera que el sistema sea **determinista**. Es más, no se tiene noción del **tiempo**

de respuesta, ni de **interrupciones**, las cuales transformar al sistema en **no determinista**.

Estas hipótesis tienen la ventaja de que lo que el modelo describe como **salida**, es realmente todo lo que el usuario puede ver. Si se implementa este modelo, entonces ya sabremos que el sistema será no interferente, debido a que la salida está descrita a nivel de especificación. Pero esto tiene otro problema, que es que en la realidad, los sistemas, en su mayoría, son **no deterministas** y **admiten interrupciones**.

Una simple red de dos computadoras ya es **no determinista**, un SO que simule la **concurrencia** también se vuelve **no determinista**.

Existe un modelo que adaptó **no interferencia** al **no determinismo**, llamado **no interferencia generalizada**, por McCullough.

2.5. Sistemas Manifiestamente Seguros

Noción de sistema manifiestamente seguro. Este concepto nace a partir de la complejidad de implementar la política MLS sin hacer un modelo innecesariamente restrictivo.

Este concepto se presenta con un ejemplo muy simple, un diagrama compuesto de **cajas** y **flechas**. Las **cajas** son máquinas que ejecutan procesos, y las **flechas** son mecanismos de comunicación (unidireccionales) entre las máquinas.

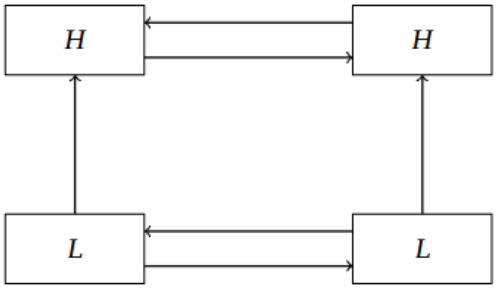

Figura 1: Descripción gráfica de un sistema manifiestamente seguro. Las flechas representan comunicaciones unidireccionales

La idea de esto, es que las máquinas de nivel **X**, solo puedan enviar información a máquinas de su nivel, o nivel mayor. Vemos que las máquinas de nivel **H** se envían información solo entre ellas, pues no hay máquinas de mayor nivel, y las máquinas de nivel **L** se envían información entre ellas, y a las máquinas de nivel **H**.

Cualquier sistema que implemente este modelo, será seguro e implementará la política MLS.

2.6. Multi Ejecución Segura

A partir de los **Sistemas Manifiestamente Seguros**, es que se desarrolló el modelo de **Multi-Ejecición segura**. Fue propuesto por Maxi en las 5tas JCCs en 2007, luego publicado por Maxi y Mata en 2009. Obtuvo mayor popularidad cuando dos belgas lo publicaron.

Este modelo se basa en el modelo de **Sistemas Manifiestamente Seguros**, donde en este caso, las cajas son procesos, y las flechas son mecanismos que provee el sistema operativo para que estos se comuniquen (envíen información).

La idea principal detrás de su funcionamiento es la siguiente:

SPDLG suponemos que tenemos dos clases de acceso H y L.

1. Cuando un proceso se lanza, se lo etiqueta como L.
2. Sigue etiquetado de la misma forma, siempre y cuando no acceda a información H.

3. Si accede a información H, entonces se hace un *fork* del proceso: Obtenemos P_L, esencialmente es el mismo proceso que el anterior, y P_H es un proceso nuevo que puede acceder a información H.

4. Un proceso clasificado como L puede escribir información en objetos L y H, pero un proceso clasificado como H solo puede escribir información en objetos H. Así, es trivial ver que este modelo implementa MLS.

Formalmente, este modelo se define de la siguiente forma:

1. Se define la sintáxis y semántica de un lenguaje de programación simple, pero Turing completo. A partir de esto definimos una función $T : M \times E \times P \to M \times E$ de transiciones, que recibe una memoria, un entorno y un programa, y devuelve la memoria y entorno modificados por el programa.

2. Se especifíca la seguridad del sistema mediante una máquina de estados S, la cuál ejecuta los programas descritos en el paso anterior.

   La idea principal de esta máquina, es que se interponga entre los procesos y el hardware, funcionando como una TCB. S controla todas las llamadas al sistema realizadas por los procesos.

   $S : M \times M \times E \times P \to M \times M \times E$

   En este caso, vemos que utilizamos dos memorias. Esto es necesario debido a que tenemos dos clases de acceso, y se utiliza una memoria para cada una.

3. Finalmente, se adapta el concepto de no interferencia a esta especificación, es decir, esencialmente a S.

   Para realizar esto, 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. Necesitamos 2 entornos que coincidan cuando son evaluados en las entradas de nivel L. Luego tres memorias, ya que cada ejecución se hará sobre la misma memoria de nivel L, pero probablemente disintintas memorias de nivel H. Por último, tomamos un programa $p \in P$ y definimos:

   $$\begin{gathered}\forall e_1, e_2 \in E\ \forall m_1, m_2, m_3 \in M; p \in P : \\ e_1(i_L) = e_2(i_L) \land S(m_1, m_2, e_1, p) = (m_1', m_2', e_1') \land \\ S(m_1, m_3, e_2, p) = (m_1', m_3', e_2') \implies e_1'(o_L) = e_2'(o_L)\end{gathered}$$

   Este modelo no esta exento de problemas. Por ejemplo, si tenemos clases de acceso A, B *y* C tales que $A < B < C$, y abrimos un editor de textos en nivel A, y luego abrimos un archivo de nivel B, se nos abrirá otro editor de texto que accede a dicho archivo, pero, que hacemos con el primer editor? Se lo puede cerrar? Nacen todas estas preguntas.

   Otro problema es la adaptación de sistemas actuales a modelos que cumplan con MLS. Vimos que en los casos de BLP y No Interferencia, los sistemas deberían sufrir cambios significativos para cumplir con su especificación y sus propiedades. Para SME nos hacemos las mismas preguntas.

3. Seguridad en Lenguajes de Programación

En esta sección veremos ataques que son posibles debido a malas prácticas de programación, y algunas soluciones posibles. Estos ataques evidencian la necesidad de suponer que trabajamos en un entrono hostil, ya que algunas de las vulnerabilidades son producidas a partir de errores de programación, y estos no tendrían mayor impacto sobre el sistema si estuvieramos en un entorno benigno. Es más, estos programas pueden ser safety-correctos, pero inseguros.

Veremos **Buffer Overflow** y **SQL Injection**

3.1. Buffer Overflow

Buffer Overflow se trata básicamente de encontrar un error de programación en un programa, que además reciba algún tipo de entrada. Este error se aprovecha, logrando sobreescribir la dirección de retorno de una función para ejecutar código malicioso, que en definitiva, termina dando el control del programa al atacante.

Este ataque es más conveniente en algunas situaciones:

1. **Ataques remotos**: El atacante realiza el ataque desde otra computadora. Principalmente se atacan: Servidores, Aplicaciones que reciben datos externos o APIs que se utilizan en estos últimos dos
2. **Ataques locales**: El atacante ataca programas dentro de su computadora, que le permiten realizar una escalada de privilegios.

Yendo a lo técnico, el error que hay que encontrar en un programa es que no se controle la longitud de un arreglo definido localmente (longitud fija), y se inserte información en él, donde esa información es externa. Por ejemplo:

```
def void f(char[] array_a, char[] array_b) {
  int i = 0;
  while (array_b[i] != '\0') {
    array_a[i] = array_b[i];
    i += 1;
  }
  array_a[i] = array_b[i];
}

def main() {
  char[4] array_a, char[8] array_b;
  // Llenamos array_b de informacion
  f(array_a, array_b);
  // Aqui array_a llenara memoria que en principio no esta asignada
  // a nadie, debido a que f no controla bien la longitud de 4
}
```

Para ver porque esto es importante para el ataque, debemos ver la estructura de los procesos, y en principal la estructura de la pila. Un proceso se forma de:

1. **Texto**
2. **Datos**
3. **Pila**

La pila contiene marcos de pila, las cuales se van apilando y desapilando luego de cada ejecución de una función, por ej.

1. **marco de main()**
2. **marco de f()**

Cada uno de estos marcos se forma de:

1. **Parámetros de la función**
2. **Dirección de retorno**

3. **Variables locales**

Notar que las direcciones crecen hacia arriba.

Con esto vemos la importancia de encontrar un arreglo definido localmente con longitud fija, ya que este va a parar por debajo de la dirección de retorno, y crece hacia arriba. Si se guarda información en este arreglo, y no se controla su longitud, si logramos insertar un código malicioso en él, y luego sobreescribir la dirección de retorno para que esta apunte al código malicioso, entonces habremos realizado un ataque de Buffer Overflow.

En relación a la defensa contra este tipo de ataques se utilizan técnicas de **prevención**, **detección** y **respuesta**.

Las medidas de **prevención** buscar evitar que existan desbordes de arreglos, por lo que principalmente se focalizan en mejorar las prácticas de programación. Además se puede:

1. Utilizar lenguajes de programación fuertemente tipados
2. Definir longitud máxima para las entradas del usuario, y controlar esta a lo largo del programa
3. No utilizar funciones de bibliotecas que sabemos que tienen este error, como por ej strcpy
4. etc.

Otras formas de **prevención** vienen definidas dentro del SO, como pueden ser:

1. **DEP (Data Execution Prevention)**: Evita que ciertas regiones de la memoria (como el stack y el heap) se ejecuten como código.
2. **ASLR (Address Space Layout Randomization)**: Aleatoriza la ubicación de secciones clave de la memoria (stack, heap, librerías, etc.), dificultando exploits que dependen de direcciones fijas.

Las medidas de **detección** se centran en detectar este ataque cuando está por ocurrir, para así evitarlo, una muy conocida es:

1. **Canarios en el stack (Stack Canaries)**: Se insertan valores aleatorios (canarios) antes de los punteros de retorno en la pila. Si un desbordamiento los sobrescribe, se detecta y se aborta la ejecución.

Por último las medidas de **respuesta** en general pasan primero por mantener bitácoras de seguridad. Ya que, si un ataque no se pudo prevenir, ni detectar, entonces es fundamental revisar esta información para ver si es necesario contraatacar con alguna respuesta. Esta puede ser desde reforzar la seguridad del sistema, hasta intentar encontrar la identidad del atacante.

Como **conclusiones** se nota que este ataque es posible debido a errores de programación, esto se resalta ya que los ataques por medio de caballos de troya no requieren esto. Igualmente, parecería que este ataque es muy común, ya que hoy en día hay muchísimo código, y probablemente mucho con este tipo de errores, pero realmente no es cosa de todos los días encontrar la situación ideal para poder realizar este ataque. Además, los SOs hoy en día ya implementan condiciones de seguridad para evitar este tipo de ataques.

Si un SO implementara alguna forma de **no interferencia**, este tipo de ataque sería inofensivo pues las protecciones implementadas por el sistema asumen que todo el software por fuera de la **TCB** es hostil. En este caso, el único problema sería que haya un desbordamiento de arreglo dentro de la **TCB**, pero la idea de esta es justamente que sea lo más pequeña y simple posible, para que pueda ser verificada formalmente, es decir, podríamos evitar este error antes de generarlo.

3.2. SQL Injection

Un ataque por medio de **SQL Injection** consiste en la inserción de sentencias SQL a través de los datos de entrada de una aplicación. Mediante este tipo de ataques se puede obtener acceso a información confidencial (credenciales) y a partir de ahí violar la integridad del sistema.

Estos ataques son comunes en aplicaciones implementadas en PHP o ASP, debido a la presencia de interfaces antiguas. Este tipo de ataques se dan cuando:

1. Hay ingreso de datos de una fuente no confiable
2. Esos datos se utilizan para construir dinámicamente sentencias SQL

El ataque consiste en insertar un meta-caracter (# o ;) en una cadena de entrada que luego se utiliza 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.

Por ejemplo, si tenemos un formulario en el cual ingresamos **Usuario** y **Contraseña**, y sabemos que la query que nos da acceso es algo parecido a

```
SELECT * FROM users WHERE username = <username> AND password = <password>;
```

entonces podemos aprovechar para insertar algún meta caracter, que corte la query y la moldee a nuestro favor, como por ejemplo:

```
SELECT * FROM users WHERE username = <username> # AND password = <password>;
```

con esto, sabiendo un usuario existente en el sistema, ya no es necesario ingresar contraseña, pues esta quedó en un código comentado. Luego, tendremos acceso con el usuario insertado en el formulario.

Para prevenir este tipo de ataques, como en buffer overflow, se puede mejorar las prácticas de programación. Además, existen otras dos técnicas para evitar este problema:

1. Sentencias pre-programadas: Estas obligan al desarrollador a definir todo el código SQL para luego pasarle solo sus parámetros, esto permite al DBMS distinguir código de datos, y así realizar chequeos sobre estos últimos para evitar este tipo de ataques.
2. Procedimientos almacenados: Estos se definen a nivel del DBMS. Estos tienen el mismo efecto que las sentencias pre-programadas si se los implementa de manera segura (lo que significa no usar SQL dinámico). Primero definimos la sentencia SQL, y luego se le pasan los parámetros. Entonces en el DBMS se programa un método que tiene un identificador y parámetros, y este es invocado desde el código.

Algunas conclusiones sobre este ataque es que, como buffer overflow, necesitan de un error de programación (en general no funcional). Por lo tanto aplican consideranciones similares al otro ataque. Además, dada la cantidad de aplicaciones web que existen, y que a través de ellas generalmente se llegan al corazón de los sistemas de una organización, y además lo fácil que es detectar y explotar este tipo de errores, entonces es muy dificil evitar estos ataques, y cuando ocurren tienen consecuencias muy graves.

3.3. Proof-Carrying-Code

Proof-Carrying-Code es una técnica de verificación de Software inventada por Necula y Lee en 1997. PCC apunte sobre todo a verificar propiedades de seguridad de tipos de programas, por lo que fue pensada para un problema de *safety*. Pero como vimos, algunos problemas de *segurity* son de *safety*, por lo que también fue estudiada por la comunidad de la Seguridad Informática. El problema a resolver es el siguiente:

1. Tenemos un programa F que va a invocar rutinas de las cuales no se tiene el código fuente
2. 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.

Por lo que queremos una forma de ver que los binarios respetan el sistema de tipos de F.

Para esto, hay que trabajar sobre código *assembler*. Generalizando la situación tenemos:

1. 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.
2. Tenemos un productor de código P que envía sus binarios a C.
3. C establece propiedades de tipos que el código programado por P debe verificar. A estas las llama *política de seguridad de tipos*.
4. P suministra el código *assembler* junto a una demostración de que este verifica la política de tipos definida por C.
5. Cuando C recibe el código, ejecuta la demostración que envió P en relación al código binario recibido.

Aunque la comprobación que realiza C sobre la demostración es automática, la parte de realizar la demostración no siempre lo es.

Cuando un programa escrito en un lenguaje L se compila, cada tipo se implementa a nivel de la máquina mediante *palabras* y punteros. Como C debe dar la política de tipos a nivel de código de máquina, debe tener en claro como se representa cada tipo de su lenguaje a nivel de máquina. Dar ejemplo de un lenguaje básico con enteros, listas, etc.

El consumidor del código formaliza las representaciones binarias de los tipos a traves de *proposiciones de tipos* de la forma: $m \vdash e : \tau$. Esto se lee como, en la memoria m la expresión e tiene tipo τ. Además estas proposiciones se dan mediante reglas de inferencia que se definen según cada lenguaje y representación binaria de sus tipos.

Estas reglas permiten al **productor** del código verificar si su código cumple con las condiciones impuestas por el **consumidor**. Esto lo hace a través de una adaptación de la lógica de Hoare, donde, si se verifica la pre-condición, entonces vale la post-condición. Estas pre y post condiciones son proposiciones de tipos.

Para esto tenemos un **vector** el cuál contiene en cada posición una instrucción assembler del programa del productor, y otro vector asociado a este llamado *vector de condiciones de verificación* donde en cada posición hay una proposición que se obtiene al aplicar una adaptación de la lógica de Hoare al código binario.

Es importante observar que es el productor el cuál debe proveer el vector que contiene las instrucciones assembler, y que además incluye la especificación y los invariantes de ciclo, y una demostración de que valen las condiciones de verificación sobre el primer vector y sobre la pre condición e invariantes definidos.

Luego de que el consumidor recibe toda la información, el proceso de verificar la prueba respecto a su código es automático y efeciente, todo se realiza *offline*.

La dificultad de PCC yace en proveer esta especificación y su demostración. Esto se resuelve implementando *compiladores de certificación* que generen esto automáticamente.

Un ejemplo de **cuando *Security* es *Safety*** lo damos a partir de la definición deun lenguaje de programación definido para una clase restringida de programas, el cuál garantiza que cualquier programa correctamente tipado tiene un flujo de información seguro. Dentro del lenguaje se definen clases de acceso para cada variable, y esta no puede ser modificada en tiempo de ejecución, es por esto que el lenguaje queda restringido. Pero esta restricción es clave para convertir un problema de *security* en *safety*.

4. Introducción a la Criptografía Aplicada

Cripografía: *Disciplina que combina la matemática y la informática para crear y estudiar métodos para transmitir datos de manera segura en un canal público. Estos métodos consisten en modificar la estrucura del mensaje pero no su significado.*

La definición de **criptografía** se realiza sobre *canales públicos*, esto debido a que no siempre es posible tener *canales privados* y son costosos. Además, hoy en día, la mayoría de los canales de comunicación, como el Internet, la radio, etc. son públicos.

Dos o más sujetos acuerdan en utilizar un **sistema criptográfico** para comunicarse entre ellos, el cuál tiene como propiedad que solo ellos pueden devolver los mensajes a su forma original, utilizando dicho sistema.

Existen **tres niveles** sobre los cuales un sistema criptográfico puede ser atacado

1. A nivel de **modelo** (matemático-computacional): Estos ataques suceden si se encuentra algún error en el modelo, o estar basado en hipótesis que no siempre se cumplen.
2. A nivel de **Software**: Los modelos son implementados de una forma, y pueden no haber sido implementados correctamente, llevando a vulnerabilidades que pueden ser aprovechadas por atacantes.
3. A nivel del **uso que se le da a la implementación**: El modelo y la implementación pueden ser correctas, pero si se utilizan en una situación donde no valen las hipótesis del modelo, entonces puede ser atacado.

Esto demuestra que la criptografía **no soluciona** todos los problemas de seguridad. Es decir, encriptar toda la información no hace que los sistemas sean más seguros, de hecho, la criptografía es fundamentalmente una herramienta para proteger comunicaciones (información en tránsito), y no tanto para información estática.

De hecho, la criptografía no impide que información encriptada sea eliminada por un Caballo de Troya, violando así la disponibilidad. Tampoco impide que este sea leído antes de ser encriptado, y después de ser desencriptado.

Opuesto a los requerimientos de la criptografía tenemos la **eficiencia** o el **costo computacional**. Es cierto que, en general, a mayor seguridad del sistema criptográfico, menor eficiencia tiene este. Por lo que no siempre se pueden utilizar todos los sistemas criptográficos en cualquier momento, ya que no siempre se tienen los recursos necesarios.

El elemento primordial de un sistema criptográfico es el **cifrador**. Este se compone de un par de algoritmos de los cuales el primero se encarga de tranformar el *texto legible* en *texto cifrado*, y el segundo realiza la operación inversa. Además este recibe una *clave*, la cuál utiliza para realizar estas operaciones. Esta es una cadena de caracteres la cuál debería realizar modificaciones notables sobre el *texto legible*, es decir, un pequeño cambio en ella, se debe traducir en grandes cambios en el *texto cifrado*, partiendo de un mismo *texto legible*.

Una característica importante de la *clave* es su longitud, la cuál se mide en bits. Aumentando esta longitud, aumenta el conjunto de claves posibles, por lo que nos podemos dar cuenta que aumentar esta clave implica aumentar la seguridad del sistema.

La fortaleza de un sistema criptográfico se mide en la cantidad de tiempo que toma obtener *texto legible* a partir de *texto cifrado*, utilizando fuerza bruta. Igualmente, se debe tener en claro que un sistema criptográfico que se rompa en 5/10 minutos **no es inservible**, existen situaciones en las cuáles la información tiene un tiempo de **vida útil** menor al dicho, por lo que se podría utilizar un sistema criptográfico de estas características para enviarla por un canal público sin problemas. Aun así, podemos pensar que teniendo la posibilidad de utilizar un mejor sistema criprográfico, porqué usuaríamos uno peor. La respuesta es el costo computacional, y que no siempre se tienen los recursos computacionales necesarios.

Los sistemas de encriptación se dividen en dos según el tipo y la cantidad de *claves* que utilizan

1. **Encriptación simétrica**: Esta utiliza una sola *clave* la cuál debe ser **privada**, es decir, solo la deben conocer las personas involucradas en la comuniación.

2. **Encriptación asimétrica**: Esta utiliza dos claves, una **pública**, y otra **privada**. Claramente tiene que existir una relación matemática entre estas dos para que el sistema de encriptación funcione correctamente.

Principio de la criptografía *La seguridad de un sistema criptográfico se basa en mantener en secreto la clave, y en no mantener en secreto los algoritmos.*

Es decir que la descripción matemática del algoritmo (e incluso su implementación) pueden ser públicos. 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.

4.1. Encriptación simétrica

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.

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).

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).

Los cifradores modernos se suelen clasificar en **cifradores de bloque** y **cifradores de flujo**, veremos los de bloque.

En la actualidad, la mayoría de los cifradores de bloque siguen un diseñopropuesto por Shannon que se denomina *crifrador producto iterativo*. La idea básica es la siguiente:

Se realizan un número de iteraciones, llamadas **rondas**, en las cuáles se aplican transformaciones denominadas **funciones de ronda**.

En cada una de estas rondas se utiliza solo una parte de la clave, denominada **subclave**.

Una implementación muy conocida es el *Cifrador de Heistel*, el cuál tiene la propiedad de que la encriptación y la desencriptación son muy similares, requiriendo solamente reversar la **selección de las subclaves**. Es decir, si comenzamos con una subclave k_0 para encriptar, esta será utilizada última al momento de desencriptar.

El **seleccionador de subclaves** es el que se encarga de seleccionar dichas subclaves.

Una forma trivial de aumentar la seguridad de un cifrador producto iterativo, es aumentar la cantidad de rondas. Pero como vimos, esto va en contra de la eficiencia del algoritmo.

4.1.1. DES

DES fue desarrollado por un grupo de investigadores de IBM a principio de los 70s. Fue solicitado por el NIST de EEUU con el fin de ser utilizado por el gobierno.

En la actualidad DES es considerado inseguro, y fue reemplazado por el AES en 2002. Aún así, la versión de 3DES se sigue considerando segura.

Si bien 3DES es seguro, existen otros algoritmos igualmente seguros, pero más eficientes, ya que 3DES se basa en aplicar tres veces el algoritmo DES.

DES es un cifrador de bloques de Feistel de 16 rondas, que utiliza claves de 64 bits. La longitud de la clave es la razón por la cuál hoy en día DES sea inseguro. 3DES es el equivalente a DES con una clave de 112 bits.

Las operaciones básicas de un cifrador de bloque son:

1. caja-P: Conjunto de operaciones de permutación.
2. caja-S: Conjunto de operaciones de sustitución.
3. caja-E: Conjunto de operaciones de expansión.
4. etc.

Estas deben cumplir unas ciertas propiedades, como por ej. la caja-P debe ser tal que un cambio en un bit de la entrada, se debe traducir en un cambio en más de la mitad de los bits de su salida. Cada bit de la salida depende de todos los bits de la entrada, etc.

A las distintas formas de encriptar texto de longitud arbitraria se las denomina **modos de operación**. Veremos dos de ellos:

- **ECB** Electronic Code Book: Cada bloque se encripta y desencripta con la misma clave, esto lleva a que similitudes en los bloque de entrada se traduzcan en similitudes en los bloques de salida.
- **CBC** Cipher Block Chaining: Se da un **vector de inicialización** el cuál se suma con **XOR** con el resultado de encriptar el primer bloque con la clave provista, el segundo bloque se suma con **XOR** con el primer bloque, y luego se encripta con la clave provista, y así sucesivamente. De esta forma no sucede lo mismo que en ECB.

4.2. Encriptación Asimétrica

El problema por el cuál se desarrolla la **encriptación asimétrica** es el **problema de distribución de claves**. Vimos que en la criptografía simétrica, la clave debe ser compartida por los usuarios que participan en la comunicación, y esto no siempre es seguro ni factible.

Vimos que estos sistemas utilizan dos claves para realizar la (des)encriptación, una clave **pública** y una clave **privada**. El funcionamiento general es el siguiente:

1. Tenemos dos usuarios **A** y **B** con claves (s_A, p_A) y (s_B, p_B)
2. Intercambian sus claves públicas.
3. **A** utiliza la clave pública de **B** para encriptar un mensaje m en m_e, luego **B** utiliza su clave secreta para desencriptarlo.

De esta forma, la clave secreta debe ser conocida por cada usuario y no por todos los involucrados en la comunicación. De esta forma, por medio de la clave pública, el problema de la distribución de claves se reduce considerablemente.

4.2.1. RSA

RSA es un algortimo de encriptación asimétrica, que se usa para proteger datos y comunicaciones. Funciona con un par de claves:

1. **Clave pública:** usada para cifrar mensajes.
2. **Clave privada:** usada para descifrarlos.

Estas claves son generadas a partir de dos números primos, y ambas mantienen una relación matemática definida.

La fortaleza de este sistema yace en elegir dichos dos números lo suficientemente grandes, ya que es difícil factorizar números grandes, lo que hace que, en teoría, solo quien tiene la clave privada pueda descifrar el mensaje.

Pero realizar esta elección no es tan sencillo, debido a que no es fácil demostrar que dos números grandes efectivamente primos. Para realizar esto hay dos formas:

- **Pruebas de primalidad**: Existen algoritmos polinomiales que realizan esto. Generalmente no son utilizados por que existen las Pruebas de primalidad probabilísticas.
- **Pruebas de primalidad probabilísticas**: Estas nos dicen si dos números son primos con una determinada probabilidad, por lo que son más eficientes que las pruebas de primalidad normales.

Además de esto, los sistemas de encriptación asimétrica tienen una lista de primos precalculada para aumentar la eficiencia.

Igualmente, lo que se quiere resaltar, es que la generación de un par de clave pública, clave privada, no puede ser realizado por un usuario convencional, por lo que mencionamos anteriormente.

Este sistema se considera seguro a partir de claves de 2048 bits. Su fortaleza yace en que es computacionalmente muy costoso (NP) obtener s_B a partir de (m_e, p_B), de esta forma esto casi resuelve el problema de la distribución de claves dado que se puede usar un canal inseguro para enviar una clave pública, puesto que la seguridad del sistema no depende de ellas.

4.2.2. Firma Digital o Electrónica

Firmar digitalmente un documento no es más que encriptarlo con la **clave privada**. De esta forma, cualquiera puede corroborar lo que firmó una persona A con solo desencriptar el documento con su **clave pública**. Notar la diferencia sustancial con el procedimiento habitual de una encriptación asimétrica. En este caso, queremos que el documento pueda ser verificado (desencriptado) por cualquiera, es por esto que se encripta con la clave privada, y se desencripta con la clave pública.

La **firma digital garantiza** preservar la integridad del mensaje luego de haber sido firmado. Es imposible modificar un mensaje antes de ser firmado, pues se necesitaría la clave privada de la persona. Se podría modificar el mensaje ya encriptado, pero esto no tiene sentido debido a que a la hora de desencriptarlo, se obtendría basura.

Por razones prácticas, la firma digital no se realiza sobre todo un documento, si no sobre un **resumen criptográfico** el cuál se calcula en base al documento. Es decir, para comprobar la autenticidad, la persona que recibe el mensaje, recalcula el resumen, desencripta el resumen encriptado y los compara.

4.2.3. Certificados Digitales y Autoridades de Certificación

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**.

Igualmente, este problema es menos severo que en el caso de la criptografía simétrica. En esta, el atacante lo único que debe hacer es escuchar el canal inseguro, y así obtener la clave privada.

Viendo el problema en el caso asimétrico, 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**.

A fin de resolver el problema de autenticidad de claves, se crea una estructura jerárquica llamada **infraestructura de clave pública** (PKI), formada por dos elementos claves **certificados digitales** (CDs) y **autoridades de certifiación** (ADs)

1. Un **CD** es un documento electrónico el cuál contiene información como: ID del propietario, clave pública del propietario, ID del AC que lo emite, etc.

2. La función del **AC** es verificar la autenticidad de las claves públicas contenidas en estos documentos. Por lo tanto, una persona le solicita a una **AC** que le certifique su clave pública para que pueda distribuirla de forma segura.

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**.

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**.

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**.

Las funciones de este tipo se utilizan para verificar la integridad y autenticidad de mensajes. Una función hash criptográfica es una función hash que además, idealmente, debe tener las siguientes propiedades:

1. Eficientes para calcular el resumen.
2. Mensajes muy similares devuelven respuestas muy diferentes.
3. Es computacionalmente complejo obtener el mensaje a partir del resumen.
4. Es computacionalmente muy complejo encontrar dos mensajes con un mismo resumen.

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.

Estas 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.

4.4. Aplicaciones

1. **Autenticación UNIX**: Históricamente, la autenticación funciona de la siguiente forma:

   a) Cuando se ejecuta el programa **passwd** con una cadena que provee un usuario (contraseña), este encripta una cadena de 8 ceros utilizando como clave la cadena recibida, luego el valor es almacenado.

   b) Al momento de realizar el **login**, cuando el usuario provee una contraseña, de nuevo se encripta una cadena de 8 ceros utilizando como clave la contraseña recibida y se compara la salida con el valor almacenado previamente, si coinciden entonces la contraseña es la correcta.

   De esta manera, se evita almacenar la contraseña.

   Hoy en día, el procedimiento es el mismo pero con un par de modificaciones, por ejemplo, se utiliza una función hash criptográfica en lugar de un cifrador de bloques. También se utiliza el concepto de *salt* para realizar el cifrado. Este valor es elegido por **passwd** y es almacenado como texto legible junto al hash. Este último se calcula *appendeando salt* antes de la contraseña. Esto es utilizado para evitar ataques de precomputación (es decir usar un diccionario de posibles contraseñas y probar una tras otra).

2. **Integridad de datos**: Se utiliza el concepto de **MAC** *código de autenticación del mensaje* que es calculado a partir de un código legible. Este es enviado junto al MAC al destinatario, y este cuando lo recibe recalcula el MAC y lo compara con el recibido. Si no coincide quiere decir que la integridad fue violada.

3. **Negociación de clave simétrica**: Como vimos, en el cifrado simétrico, uno de los problemas principales era la distribución de las claves, debido a que estas deben ser solo conocidas por los participantes de la comunicación. Una forma de resolver esto, es utilizar encriptación asimétrica para negociar la clave privada de una sesión que use criptografía asimétrica. Una sesión es básicamente una comunicación continua entre dos o más personas. Esto se realiza en algunas situaciones debido a que generalmente los algoritmos de cifrado simétrico son más eficientes que los de cifrado asimétrico.

4. **El protocolo de Needham-Schroeder (versión simétrica)**: Es un protocolo de autenticación basado en criptografía simétrica, que permite a dos entidades A y B establecer una clave de sesión compartida mediante un servidor de autenticación de confianza S.

   Los pasos del protocolo son los siguientes:

   a) A envía a S una solicitud para comunicarse con B, incluyendo su identidad y la de B.

   b) S genera una clave de sesión K_{AB} y la envía a A, cifrada con la clave K_A compartida entre A y S, junto con un ticket T_B cifrado con la clave K_B compartida entre B y S.

   c) A reenvía el ticket cifrado T_B a B.

   d) B genera un número aleatorio N_B y lo envía a A cifrado con K_{AB}.

   e) A responde con el número aleatorio modificado (por ejemplo, restándole 1) cifrado con K_{AB}, confirmando así que conoce la clave.

   Este protocolo garantiza que A y B compartan una clave secreta segura sin necesidad de una clave previa entre ellos. Sin embargo, es vulnerable a ataques de repetición si no se implementa correctamente.

5. **El protocolo Secure Sockets Layer**: El protocolo Secure Sockets Layer (SSL) es un protocolo de seguridad diseñado para proporcionar comunicaciones seguras sobre una red mediante el uso de cifrado, autenticación e integridad. SSL fue sucedido por Transport Layer Security (TLS), pero el funcionamiento básico sigue siendo similar.

   Consta de tres fases principales:

   Negociación de la clave

   a) El cliente envía un mensaje *ClientHello* al servidor, incluyendo una lista de cifrados soportados y un número aleatorio.

   b) El servidor responde con un *ServerHello*, eligiendo un cifrado y enviando su propio número aleatorio.

   c) El servidor envía su certificado digital para autenticarse ante el cliente.

   d) Si se usa autenticación de cliente, este envía su certificado al servidor.

   e) Se intercambia una clave pre-maestra cifrada con la clave pública del servidor (en caso de RSA) o se genera mediante un intercambio de claves (como Diffie-Hellman o ECDH).

   f) Se calcula una clave de sesión compartida a partir de la clave pre-maestra y los números aleatorios intercambiados.

   Cifrado y autenticación

   a) Se generan claves simétricas para cifrar la comunicación utilizando el cifrado acordado.

b) Se envían mensajes de finalización (*Finished*) para confirmar que el handshake fue exitoso.

c) A partir de este punto, la comunicación es segura y está cifrada con la clave de sesión.

Intercambio de datos

a) Los datos de la sesión se transmiten de manera cifrada utilizando las claves derivadas en el handshake.

b) Se utilizan códigos de autenticación de mensajes (MAC) para garantizar la integridad de los datos.

c) Si se desea finalizar la sesión, cualquiera de las partes puede enviar un mensaje de cierre (*CloseNotify*).

SSL proporciona autenticación, confidencialidad e integridad mediante una combinación de criptografía asimétrica (para la negociación de claves) y criptografía simétrica (para la comunicación segura). Ha sido reemplazado por TLS en versiones más modernas debido a vulnerabilidades en versiones antiguas de SSL.

5. 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?

La respuesta es *análisis de protocolos criptográficos.*

Esto consiste en formalizar un determinado protocolo en alguna lógica, para poder formalizar y estudiar sus propiedades de seguridad.

La lógica que veremos es la *lógica BAN* (Por los apellidos de sus Autores).

Esta lógica se basa en dos nociones centrales **creencias** y **mensajes vistos**.

- **Creencia**: Se relaciona con que un sujeto cree que cierto dato es verdadero.
- **Mensajes Vistos**: En el sentido anterior, los mensajes vistos corresponden a lo que un sujeto conoce.

Los sujetos van aumentando sus creencias y conocimientos a medida que se ejecuta el protocolo.

5.1. 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.

El lenguaje consta de construcciones de la siguiente forma:

1. $P \overset{\text{cree}}{\Longrightarrow} X$. El sujeto P puede actuar como si X fuera verdadera.
2. $P \overset{\text{ve}}{\Longrightarrow} .X$ Alguien le ha mandado a P un mensaje conteniendo X, quien puede leerlo y reenviarlo.
3. $P \overset{\text{dijo}}{\Longrightarrow} .X$ 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.
4. $P \overset{\text{tjs}}{\Longrightarrow} .X$ El sujeto P **tiene jurisdicción sobre** X. P tiene autoridad sobre X y los otros sujetos deben tenerle confianza sobre X.
5. $\#(X)$. La sentencia X es **fresca**; es decir, nunca fue incluida en un mensaje antes de la ejecución actual del protocolo.

6. $P \overset{\mathrm{K}}{\leftrightarrow} Q$. K es una clave compartida entre P y Q. Esta clave nunca será conocida por otros sujetos salvo aquel que la haya generado.

7. $P \overset{\mathrm{K}}{\rightarrow} Q$. 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.

8. $\{X\}_K \ from \ P$. Significa que X fue generado con la clave K a partir de un sujeto P. El $from$ en algunos contexto se puede evitar.

5.2. Reglas de Inferencia

A partir del lenguaje definido en la sección anterior se definen reglas de inferencia, como por ejemplo:

1. 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.

2. Similar a la anterior pero con clave pública.

3. Verificación de sentencias frescas. 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.

4. Si P le adjudica a Q jurisdicción sobre X entonces P cree en la veracidad de X.

5. etc.

5.3. Metodología para analizar protocolos

1. Se especifica el protocolo partiendo de una descripción. 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. La traducción de un mensaje a **BAN** solo se puede hacer si se comprende el protocolo como un todo. 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. 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.

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.

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

5.4. 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, ¿cómo sabemos que un protocolo criptográfico es seguro? ¿Qué propiedades deben verificar estos protocolos?

Aunque estas propiedades son materia de debate existen algunas casi obvias. En general el objetivo más buscado es compartir una clave simétrica.