"Aplicación de la Programación Orientada a Aspectos como Solución a los Problemas de la Seguridad en el Software"

Fernando Asteasuain [fa@cs.uns.edu.ar]
Leandro Ariel Schmidt[leandroschmidt@hotmail.com]

Resumen: Actualmente, la seguridad es un concepto que todo sistema debe incorporar. La Ingeniería del Software todavía no es capaz de brindar un mecanismo para implementar adecuadamente la seguridad: los lenguajes tienen primitivas inseguras, el código relativo a la seguridad es siempre un código confuso y complejo debido a técnicas de abstracción insuficientes, etc. Estos problemas son los objetivos que promete satisfacer el nuevo paradigma de Programación Orientada a Aspectos(POA). Por lo tanto, en este trabajo, analizamos la aplicación de la POA como solución a los problemas de la seguridad en el software.

1. Introducción

El concepto de la seguridad en los sistemas de software es un área de investigación que ha pasado a ser vital dentro de la Ingeniería de Software. Con el crecimiento de Internet, y otras aplicaciones sobre redes, como el comercio electrónico, correo electrónico, etc., la posibilidad de ataques se ha incrementado notablemente, como también lo han hecho las consecuencias negativas de estos ataques.

En la actualidad prácticamente todo sistema debe incorporar cuestiones de seguridad para defenderse de ataques maliciosos. El desarrollador ya no sólo debe concentrarse únicamente en los usuarios y sus requerimientos, sino también en los posibles atacantes. Esto ha motivado cambios importantes en el proceso de diseño y desarrollo de software para incorporar a la seguridad dentro de los requerimientos críticos del sistema.

Estos cambios son los primeros pasos en la concientización de los ingenieros de software acerca de la importancia de obtener un software seguro. Como todo gran cambio, no se logra de un día para otro, sino que es un proceso gradual que requiere tiempo y maduración. Todavía la Ingeniería de Software no ha dado una respuesta eficaz, coherente y aplicable que satisfaga plenamente a la comunidad informática, sino que las aproximaciones actuales están plagadas de fallas y debilidades fruto de que todavía es un proceso que se encuentra en su infancia.

En este sentido, existe un nuevo paradigma de programación, la programación orientada a aspectos (POA) [7,9], el cual al permitir encapsular requerimientos o conceptos típicamente no funcionales, como la seguridad, se convierte en una herramienta atractiva para el desarrollo de software seguro. La POA nos permitiría encapsular las políticas de seguridad de forma separada e independiente del resto del sistema, con las consecuentes ventajas que esto implica:

- Mayor reusabilidad.

- Mejoras sustanciales respecto al mantenimiento, modificaciones, etc.
- Mayor grado de especialización.
- Menor complejidad del sistema.

Vemos que si bien la POA no agrega por si misma seguridad a sus aplicaciones, nos brinda un entorno ideal para incorporar el concepto de seguridad de forma coherente y natural en el desarrollo de software. Una de las reglas generales que gobiernan la correcta implementación de la seguridad es que la misma debe aplicarse correcta y consistentemente a través de todo el sistema [6]. Esto se logra naturalmente con este nuevo paradigma.

Nuestro trabajo apunta entonces a investigar la aplicabilidad de la POA a la seguridad en software, al considerarlo como la alternativa más viable y prometedora en pos de lograr software seguro.

2. Problemática actual de la seguridad en el software

Los puntos débiles más importantes de la Ingeniería de Software con respecto a la seguridad pueden ser clasificados en dos grandes categorías:

i) Fallas para implementar software seguro.
ii) Fallas para implementar seguridad en el software.

2.1 Fallas para implementar software seguro

Lamentablemente, la mayoría de las herramientas que tiene disponible un desarrollador de software sufren de fallas propias de seguridad.

Una de las debilidades más trascendentes al momento de implementar software seguro surge del estado de los lenguajes de programación desde el punto de vista de la seguridad. Son escasos los lenguajes que proveen primitivas “seguras” que ayuden al programador a escribir un mejor código.

Dos de los lenguajes de programación más usados en la actualidad, *C* y *C++*, presentan graves problemas de seguridad. Esto se debe a que al utilizar muchos de sus servicios provistos por sus librerías estándar se introducen fallas de seguridad que pasarán inadvertidas al programador debido a que éste las considera libre de errores. Como ejemplo, podemos citar el problema de “buffer overflow”, fallas en la rutina *malloc(a,b)* y en la rutina *rand()* [8]. También, en un trabajo de John Viega y su equipo [10], se identifican numerosas fallas en el lenguaje C: en la rutina *gets*, donde también se explota el “overflow” de buffers, en las rutinas para manejar de strings *strcat*, *strcpy*, *sprintf*, en las rutinas *system* y *popen*, para correr programas desde la línea de comandos, que son generalmente usadas de manera incorrecta, y otras fallas más sutiles, que se pueden introducir al considerar cuestiones de sincronización.

También lenguajes con arquitecturas de seguridad mucho más complejas, como Java, dejan mucho que desear. En [8], se establece que un alto porcentaje de aplicaciones tiene problemas de seguridad que están presenten desde la fase de diseño y que permanecen hasta la implementación, independientemente del lenguaje de programación usado.

Las razones por las cuales estas fallas “internas” permanecen en la actualidad son varias: mal entendimiento de los protocolos de seguridad, una visión ingenua respecto a lo que un sistema debiera considerar como seguro, aproximaciones no serias a la seguridad como “corregir luego” o “no se van a dar cuenta”, o directamente desconocimiento, ya que lamentablemente estas fallas no son conocidas universalmente, y existen pocas fuentes de información para escribir código seguro.

Otro tipo falla en esta categoría, que se establece en [4], nace del hecho de que la seguridad es un tema complejo y requiere un entendimiento completo sobre lo que puede ir mal y qué es lo que puede ser explotado por un posible atacante. Un programador promedio no cuenta con la experiencia suficiente como para poder determinar los requerimientos de seguridad que necesite su aplicación. Esto resulta en la subestimación de pequeños detalles que luego pueden llegar a introducir grandes fallas de seguridad.

2.2 Fallas para implementar seguridad en el software

En la actualidad, el desarrollador de software que quiera incorporar seguridad a su sistema se enfrentará con la difícil realidad de las técnicas de programación tradicionales, y también, con una Ingeniería de Software que recién está aprendiendo sobre la seguridad.

Típicamente la seguridad es considerada como un requerimiento no funcional. Luego, debido a los problemas de planificación y presupuesto, la seguridad sólo es tenida en cuenta una vez que los requerimientos funcionales son obtenidos. Esto conduce a que la seguridad sea considerada como un concepto “afterthougth”[1], y que se incorpore tardíamente al sistema. Esto lleva a una implementación pobre, ineficiente, e inadecuada de la seguridad. También, la mayoría de las metodologías de diseño y herramientas dedicadas a la seguridad trabajan de esta forma, como herramientas “afterthougth” (ver sección de Trabajo Relacionado). Por lo tanto, es mandatorio que los conceptos de seguridad formen parte integral en todo el ciclo de vida de desarrollo de software, tal como se lo demanda en [1,3].

Una de las aproximaciones más ampliamente utilizada para la seguridad es la aproximación “ataque-parche” (en inglés “penetrate and patch”), en donde la seguridad es tratada de una manera ad-hoc, a medida que las fallas se van revelando. Esto es, se desarrolla el sistema con mínimas consideraciones con respecto a la seguridad. Luego, una vez que el sistema está funcionando se detectarán los ataques al mismo, y se buscará recién ahí la manera de corregirlos. Bajo esta aproximación, claramente, es imposible implementar la seguridad de una manera adecuada:

1) Se deben detectar los ataques, que no es una tarea menor.

2) Una vez identificado el ataque, se deben tomar las acciones necesarias como para que este tipo de ataque no vuelva a ocurrir. Esto implica revisar todo el código del sistema y ubicar el(los) lugar (lugares) donde se deban introducir las modificaciones. Como sabemos, esta tarea no es para nada sencilla.
3) Volver a revisar el código y analizar la posible existencia de efectos colaterales introducidos por las modificaciones del paso 2).

Esta aproximación a la seguridad es similar la gestión de riesgo "Indiana Jones"[11]: "*no ocuparse de la seguridad hasta que se de un ataque, entonces buscar una solución heroica*". Como establece Pressman en [11], ni el ingeniero de software es Indiana Jones ni el resto del equipo sus valientes colaboradores. Es imperioso por lo tanto un enfoque más formal al tema de la seguridad en la Ingeniería de software.

Otra notable debilidad proviene del hecho de que las técnicas tradicionales de descomposición no logran encapsular correctamente el concepto de seguridad. Ya sea con la Programación Orientada a Objetos (POO), con la programación estructurada por bloques, o con la programación declarativa, no se consigue separar adecuadamente el concepto de seguridad del resto del sistema.

En [4], Bart de Win y sus colaboradores, sugieren que esta incorrecta modularización se produce por una diferencia de estructuras. Se mencionan dos estructuras diferentes: por un lado, las estructuras de abstracción y modularización propuestas por las técnicas actuales de descomposición, y por el otro, la estructura de la seguridad como requerimiento. Esta incompatibilidad estructural, lleva a que en el momento de introducir la seguridad en el sistema, el código correspondiente a la seguridad quede desparramado, disperso, por todo el sistema. Los problemas que trae consigo este código "mezclado"[7] son numerosos:

- Código duplicado.
- Aumento en la complejidad del mantenimiento: introducir un cambio implica revisar todo el código, ya que el código de seguridad resulta disperso, y tan solo olvidarse de una modificación llevará a una falla de seguridad.
- Incremento en la complejidad global del sistema: el código de la funcionalidad básica se ve "ensuciado" por el código relativo a la seguridad.
- Menor flexibilidad para el manejo de políticas de seguridad: al ser compleja la realización de cambios, también será complejo implementar cambios en la política de seguridad.

3. Objetivos para un software seguro

En pos de conseguir un software seguro, se debe dejar claro qué se entiende por seguridad, para así luego poder establecer requisitos mínimos que debe satisfacer un sistema que pretenda ser considerado seguro.

Como definición del concepto de seguridad en software, se adoptará en este trabajo la definición que propone Doshi Shreyas en [1]: la seguridad de un sistema de

software es un concepto multi-dimensional. Las múltiples dimensiones de la seguridad son:

- Autenticación: el proceso de verificar la identidad de una entidad.
- Control de acceso: el proceso de regular las clases de acceso que una entidad tiene sobre los recursos.
- Auditoria: un registro cronológico de los eventos relevantes a la seguridad de un sistema. Este registro puede luego examinarse para reconstruir un escenario en particular.
- Confidencialidad: la propiedad de que cierta información no esté disponible a ciertas entidades.
- Integridad: la propiedad de que la información no sea modificada en el trayecto fuente-destino.
- Disponibilidad: la propiedad de que el sistema sea accesible a las entidades autorizadas.
- No repudio: la propiedad que ubica la confianza respecto al desenvolvimiento de una entidad en una comunicación.

La seguridad puede tener diferentes significados en distintos escenarios. En general, cuando se habla de seguridad implica referirse a más de una de las dimensiones mencionadas anteriormente. Por ejemplo:

- Seguridad en correo electrónico: involucra confidencialidad, no repudio e integridad.
- Seguridad en compras online: implica autentificación, confidencialidad, integridad y no repudio.

Bajo este punto de vista, se define un ataque a la seguridad como un intento de afectar en forma negativa una o más de las dimensiones del concepto de seguridad.

Una vez definido el concepto de seguridad, se pueden establecer objetivos básicos para un software seguro:

- **Independencia de la seguridad**: la seguridad debe construirse y utilizarse de manera independiente de la aplicación.
- **Independencia de la aplicación**: la aplicación no debe depender del sistema de seguridad usado, debe ser desarrollada y mantenida en forma separada.
- **Uniformidad**: la seguridad debe aplicarse de manera correcta y consistente a través de toda la aplicación y del proceso que desarrolla la misma.
- **Modularidad**: mantener la seguridad separada. Entre otras ventajas, esto nos brindará mayor flexibilidad y menor costo de mantenimiento.
- **Ambiente seguro**: se debe partir de un entorno confiable. Es decir, las herramientas de desarrollo y lenguajes de programación no deben contener agujeros de seguridad.
- **Seguridad desde el comienzo**: la seguridad debe ser considerada como un requerimiento desde el inicio del diseño.

4. Programación Orientada a Aspectos como solución

En esta sección se estudia la aplicabilidad de la Programación Orientada a Aspectos (POA) al problema de la seguridad. Está dividida en tres partes: primero, una introducción al paradigma; luego, se analizan algunos casos de estudio específicos, y por último se valora el impacto de la POA en la seguridad.

4.1 Introducción a la Programación Orientada a Aspectos[1]

Existen conceptos que no pueden ser encapsulados con las metodologías de programación actuales dentro de una unidad funcional, debido a que atraviesan todo el sistema, o varias partes de él. Algunos de estos conceptos son: sincronización, manejo de memoria, distribución, chequeo de errores, profiling, seguridad o redes, entre otros. Aquí se muestran algunos ejemplos:

1) Consideremos una aplicación que incluya conceptos de seguridad y sincronización, como por ejemplo, asegurarnos que dos usuarios no intenten acceder al mismo dato al mismo tiempo. Ambos conceptos requieren que los programadores escriban la misma funcionalidad en varias partes de la aplicación. Los programadores se verán forzados a recordar todas estas partes, para que a la hora de efectuar un cambio y / o una actualización puedan hacerlo de manera uniforme a través de todo el sistema. Tan solo olvidarse de actualizar algunas de estas repeticiones lleva al código a acumular errores.
2) Manejo de errores y de fallas: agregar a un sistema simple un buen manejo de errores y de fallas requiere muchos y pequeños cambios y adiciones por todo el sistema debido a los diferentes contextos dinámicos que pueden llevar a una falla, y las diferentes políticas relacionadas con el manejo de una falla [9].
3) En general, los aspectos en un sistema que tengan que ver con el atributo performance, resultan diseminados por todo el sistema [9].

Podemos afirmar entonces que las técnicas tradicionales no soportan de una manera adecuada la separación de las propiedades de aspectos distintos a la funcionalidad básica, y que esta situación tiene un impacto negativo en la calidad del software.

Como respuesta a este problema nace la Programación Orientada a Aspectos. La POA permite a los programadores escribir, ver y editar un aspecto diseminado por todo el sistema como una entidad por separado, de una manera inteligente, eficiente e intuitiva. La POA es una nueva metodología de programación que aspira a soportar la separación de las propiedades para los aspectos antes mencionados. Esto implica separar la funcionalidad básica y los aspectos, y los aspectos entre sí, a través de mecanismos que permitan abstraerlos y componerlos para formar todo el sistema.

La idea central que persigue la POA es permitir que un programa sea construido describiendo cada concepto separadamente. El soporte para este nuevo paradigma se

[1] Esta introducción se obtuvo de [7], para los interesados en profundizar sobre el tema.

logra a través de una clase especial de lenguajes, llamados lenguajes orientados a aspectos (LOA), los cuales brindan mecanismos y constructores para capturar aquellos elementos que se diseminan por todo el sistema. A estos elementos se les da el nombre de aspectos. Se define entonces a un aspecto como un concepto que no es posible encapsularlo claramente, y que resulta diseminado por todo el código. Los aspectos son la unidad básica de la programación orientada a aspectos.

Los tres principales requerimientos de la POA son: [7]

- Un lenguaje para definir la funcionalidad básica, conocido como lenguaje base o componente. Podría ser un lenguaje imperativo, o un lenguaje no imperativo (C++, Java, Lisp, ML).
- Uno o varios lenguajes de aspectos, para especificar el comportamiento de los aspectos. (COOL, para sincronización, RIDL, para distribución, AspectJ, de propósito general.)
- Un tejedor de aspectos (del inglés weaver), que se encargará de combinar los lenguajes. Tal proceso se puede retrasar para hacerse en tiempo de ejecución o en tiempo de compilación.

Los lenguajes orientados a aspectos definen una nueva unidad de programación de software para encapsular aquellos conceptos que cruzan todo el código. A la hora de "tejer" los componentes y los aspectos para formar el sistema final, es claro que se necesita una interacción entre el código base y el código de los aspectos. También es claro que esta interacción no es la misma que ocurre entre los módulos del lenguaje base, donde la comunicación está basada en declaraciones de tipo y llamadas a procedimientos y funciones. La POA define entonces una nueva forma de interacción, provista a través de los puntos de enlace.

Los puntos de enlace brindan la interfaz entre aspectos y componentes; son lugares dentro del código donde es posible agregar el comportamiento adicional que destaca a la POA. Dicho comportamiento adicional es especificado en los aspectos. Dado un punto de enlace *pde*, este comportamiento adicional puede agregarse, en general, en tres momentos particulares:

- "antes de *pde*".
- "después de *pde*"
- "en lugar de *pde*"

Por ejemplo, dentro de la POO, algunos puntos de enlace serían: llamadas a métodos, creación de objetos, acceso a atributos, etc.

Aún nos falta introducir el encargado principal en el proceso de la POA. Este encargado principal conocido como tejedor debe realizar la parte final y más importante: "tejer" los diferentes mecanismos de abstracción y composición que aparecen tanto en los lenguajes de aspectos como en los lenguajes de componentes, guiado por los puntos de enlace.

La estructura general de una implementación basada en aspectos es análoga a la estructura de una implementación tradicional. Una implementación tradicional consiste de:

- Un lenguaje.
- Un compilador o intérprete para ese lenguaje.
- Un programa escrito en ese lenguaje que implemente la aplicación.

Una implementación basada en la Programación Orientada a Aspectos consiste en [9]:

- El lenguaje base o componente para programar la funcionalidad básica.
- Uno o más lenguajes de aspectos para especificar los aspectos.
- Un tejedor de aspectos para la combinación de los lenguajes.
- El programa escrito en el lenguaje componente que implementa los componentes.
- Uno o más programas de aspectos que implementan los aspectos.

Gráficamente, se puede comparar ambas estructuras, como queda reflejado en la figura 1.

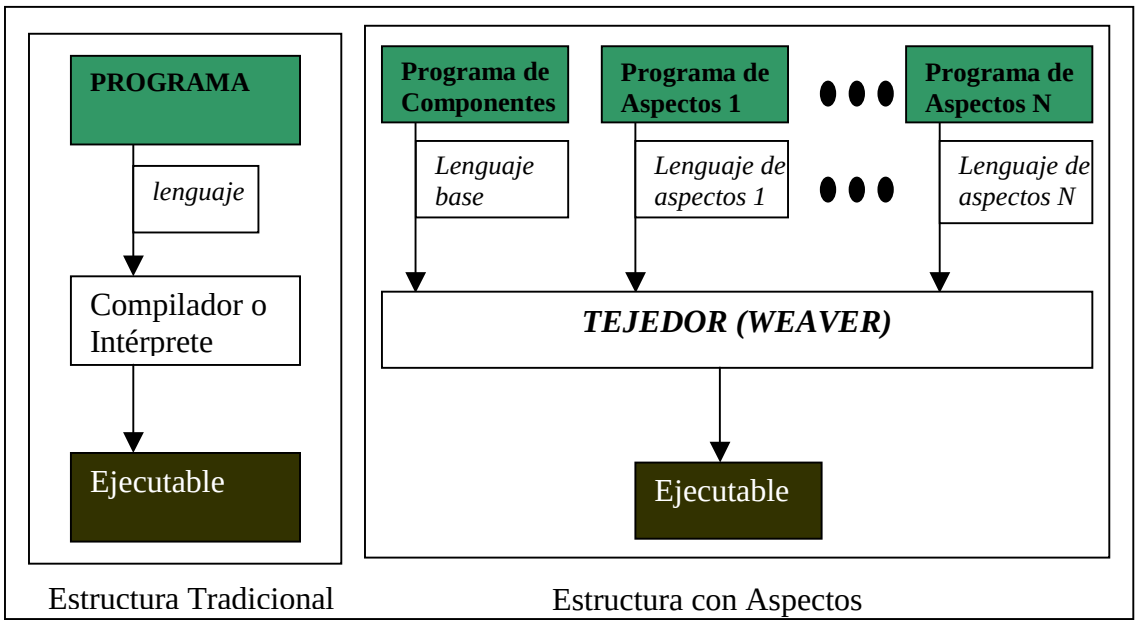

Figura 1: Comparación entre las estructuras tradicionales y las estructuras con aspectos.

4.2 Casos de estudio

A continuación se analizan diferentes contextos en los cuales la implementación de aspectos muestra una clara ventaja respecto al resto de los paradigmas en lo que hace a principios fundamentales de la Ingeniería de Software como la encapsulación, modularidad, legibilidad, facilidad de mantenimiento, etc.

4.2.1 Ejemplo biblioteca

Para analizar el primer ejemplo considerar la siguiente definición de una clase para el manejo de una biblioteca, donde se necesita validar el acceso a algunos métodos de la clase. Dicha clase se muestra en la figura 2.

En los dos métodos detallados, se marcó con azul el código que se corresponde con la funcionalidad básica, y con rojo, el código correspondiente al control de acceso. Cómo vemos, ambos códigos se mezclan, con las consecuentes desventajas que esto acarrea. Por un lado el código base, que sería el manejo de la biblioteca en sí, se ve "ensuciado" por el código de control de acceso, y por el otro, el código para el control de acceso queda disperso por toda la clase. Ahora, ¿ que ocurriría si necesitamos modificar de alguna forma las políticas sobre el control de acceso? Se debe recorrer exhaustivamente toda la clase en busca de sentencias que involucren controles de este tipo, y realizar los cambios correspondientes. Esto proceso no es sólo tedioso sino también es una fuente de generación de errores, que se corresponden con fallas de seguridad.

```
Class Biblioteca {
  private libro [] libros ;
  private socio [] socios;

  public Biblioteca() {
  …
  }
  public void prestamo( socio S, libro L)
{
   if controlDeAccesoValido() then{
   // código del método
   }
   else{
    informarError();
   }
  }
  public void ingresarSocio(socio S){
   if controlDeAccesoValido() then{
   // código del método
   }
   else{
    informarError();
  }
 }
// demás métodos…
}
```

Figura 2: Clase Biblioteca

La misma clase implementada con aspectos sería similar a la anterior, salvo que desaparece todo el código marcado en rojo, y queda únicamente el código en azul, que representa la funcionalidad básica del sistema. Se tiene entonces una funcionalidad básica mucho más limpia y pura, debido a que el código tiene ahora que ver únicamente con el manejo de la biblioteca. Para el control, se define el siguiente aspecto:

```
Aspecto Control {

punto de enlace operacionesSeguras = llamadas a Biblioteca.prestamo &
                                     llamadas a Biblioteca.ingresarSocio&
                                     ...
antes de operacionesSeguras:  {
if  !=(controlDeAccesoValido()) then{
      informarError();
}
```

De esta forma se tiene en un solo lugar todo el código referente al control de acceso, facilitando de esta manera la legibilidad y el mantenimiento. Además, ahora el control de acceso es una entidad independiente, con lo cual un cambio en la política de control de acceso es casi trivial de implementar.

4.2.2 Ejemplo Trivial File Transport Protocol (TFTP)

En [7] se implementa con AspectJ[15], uno de los lenguajes orientados a aspectos más populares, el protocolo TFTP. El aspecto principal que se consideró fue el aspecto de logging, Se concluyó que la utilización de aspectos fue más que exitosa, ya que el código relativo al logging representaba un 30% del total del código. Se obtuvo una funcionalidad base limpia, con código referente únicamente al protocolo. Además, se definió un aspecto de logging, donde se detallan todas las acciones del logging en forma natural, modular, e independiente del resto del código.

4.2.3 Ejemplo de un lenguaje de programación C seguro

J. Viega, J.T. Bloch y P. Chandra, describen en [8], la implementación de un lenguaje C seguro. Se trata de una extensión orientada a aspectos del lenguaje C convencional, que intenta solucionar los problemas de seguridad clásicos de este lenguaje.

Esta aproximación utiliza aspectos para:

- reemplazar funciones inseguras por versiones seguras de las mismas. Por ejemplo, reemplaza las llamadas a *rand()* por *secure_rand(), malloc() por secure_malloc(), etc.*
- Realizar automáticamente chequeos de error en llamadas críticas de seguridad.

- Implementar técnicas de protección de “buffer overflow”, insertando código especial antes y después de las llamadas a las funciones que sufran este problema.
- Logear automáticamente datos relevantes a la seguridad.
- Reemplazar código genérico de socket por código de socket SSL.

El funcionamiento de este nuevo lenguaje se ve reflejado en la figura 3. En tiempos de compilación el lenguaje toma el programa fuente escrito en C, el aspecto de seguridad, y los “teje” en un único programa C, que luego es compilado (por un compilador tradicional de C), para así obtener el ejecutable.

Esta aproximación intenta solucionar algunas de las fallas para implementar software seguro, mencionadas en la sección 2.1. En este caso se trata de solucionar las fallas propias del lenguaje C. Sin embargo, el enfoque es lo suficientemente general como para ser aplicado a otros lenguajes.

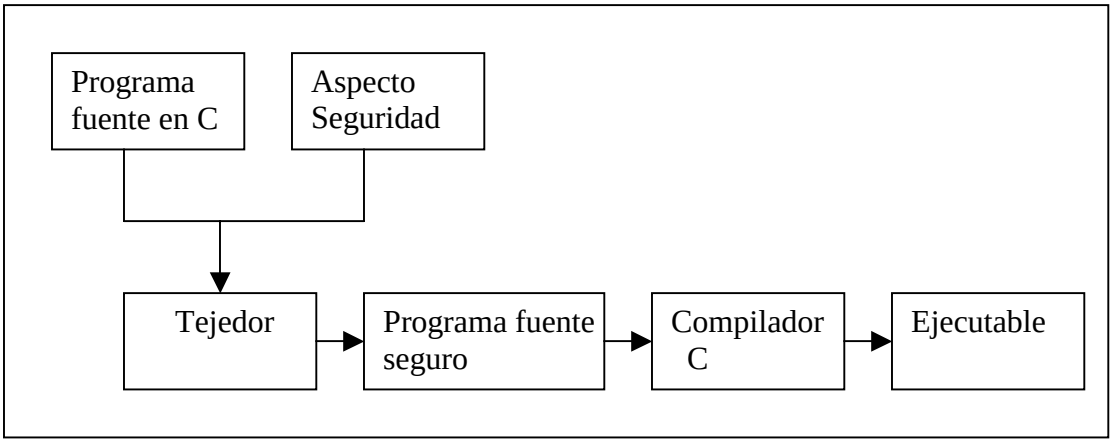

Figura 3. Esquema del tejedor C seguro.

4.2.4 Ejemplo de implementación de sistemas operativos con aspectos

La implementación de sistemas operativos ha sido desde siempre una tarea compleja. Sin embargo, parte de esta complejidad se debe a problemas de modularidad, dado que los sistemas sufren interacciones no intencionales entre los módulos, y requieren por lo tanto que los desarrolladores estén íntimamente familiarizados con los patrones de interacción implícitos entre subsistemas.

Teniendo esto en cuenta, en [12,13] se analiza si la POA, al capturar conceptos entrecruzados, mejora o no la modularidad en la implementación de sistemas operativos.

En estos trabajos se concluye que gran parte de la complejidad en el código de un sistema operativo es innecesaria, y proviene de técnicas no apropiadas de abstracción más que de la complejidad inherente al dominio. En la implementación sin aspectos, los conceptos entrecruzados formaron un código totalmente oscuro y enredado. Como contrapartida, en la implementación con POA la estructura de conceptos entrecruzados resultó clara y manejable.

Para la implementación con aspectos se desarrollo el lenguaje AspectC[14], una extensión orientada a aspectos para el lenguaje C. AspectC cuenta con mecanismos que permiten abstraer y modularizar conceptos entrecruzados.

4.2.5 Ejemplo generalización de políticas en AspectJ

En [5], se estudia la posibilidad de obtener un contexto adecuado que posibilite definir políticas generales de seguridad, que luego puedan ser instanciadas en cada caso específico. De esta forma, se logran importantes ventajas al poder reutilizar políticas de seguridad.

AspectJ[15], al brindar la posibilidad de definir puntos de enlace abstractos y permitir herencia entre los aspectos, se convierte en la herramienta ideal para este tipo de aproximación.

4.3 Impacto de la POA en la seguridad

La POA nos permite encapsular y abstraer el concepto de seguridad. Con esto se logra:

- Bajo costo de mantenimiento: al tener todo el código relativo a la seguridad en una única unidad se simplifica la introducción de cambios.
- Mayor flexibilidad: al estar la seguridad implementada de forma independiente, es mucho más sencillo realizar cambios en las políticas de seguridad
- Mayor especialización: los desarrolladores pueden implementar la funcionalidad básica, mientras que un experto en seguridad puede especificar las propiedades de seguridad.
- Independencia de tres niveles:
    i. la seguridad es independiente de la funcionalidad básica
    ii. la seguridad es independiente de los implementadores
    iii. la seguridad es independiente del momento en que se la genera.

También, se debe reconocer que existen ciertos puntos débiles. Por un lado, la POA es un paradigma que recién está naciendo, y por lo tanto continuamente surgen nuevos problemas. Los lenguajes orientados a aspectos no son todavía herramientas maduras ni libres de errores. Los investigadores afirman que el estado actual de la POA es similar al estado en que se encontraba la POO hace 20 años. Además, tampoco tiene la POA un respaldo teórico importante que lo sustente, sino que padece un estado de informalidad acentuado.

5. Trabajo relacionado

Existen otras áreas de investigación, que como la POA, buscan solucionar el problema de la seguridad en el software:

- **Agentes inteligentes**: agentes de software llamados SafeBoots[16] que pueden ser implementados para monitorear y realizar control de acceso, autentificación, etc, capturando componentes inseguras. Poseen cierta inteligencia para adaptar sus acciones a un entorno global y local. La programación de este tipo de agentes aparece como prometedor para el monitoreo de seguridad en sistemas distribuidos. Sin embargo, esta programación es una tarea difícil.
- **Herramientas para la seguridad**: existen diferentes tipos de herramientas que contribuyen a solucionar problemas de seguridad conocidos:
    i. *"Buffer overflow"*: herramientas como StackGuard[17] y FIST [18], diseñadas exclusivamente a evitar este problema.
    ii. *"Flujo de dato seguro"*: existen extensiones a Java y Perl [19] que permiten programación de este tipo. En estas herramientas cada dato es etiquetado como "confiable" o "no confiable". Luego dato "no confiable" no puede ser pasado a ítems "confiables" y viceversa, a menos que el programador explícitamente lo autorice.
    iii. "After-the-fact": herramientas que soportan el modelo "ataque-parche". Estas herramientas generalmente tomen código fuente preexistente e identifican constructores potencialmente dañinos basadas en una base de datos y en análisis estático. Una de ellas es ITS4[10], la cual analiza código de C y C++ en busca de casi cien problemas de seguridad conocidos.
- **Arquitecturas de seguridad**: la mayoría de estos sistemas permiten al programador especificar sus propias políticas, estableciendo qué es lo que un programa puede hacer y qué no. Ejemplos de tales sistemas son: Naccio[2], Ariel[20] y PolicyMaker[21].
- **Arquitecturas Meta-nivel:** permiten también separar la implementación de la seguridad de la implementación del resto de la aplicación. Existen meta-programas que tiene control sobre las entidades y pueden intervenir en la ejecución de la aplicación. Comparado con la POA, es un mecanismo más poderoso, pero también más complejo, dado que el programador está forzado a pensar en meta-elementos, que tienen solo una relación indirecta con la aplicación[4]. Ejemplos de estos arquitecturas son: [22] y [23].

6. Conclusiones

Nuestro análisis nos permite concluir que la POA es una prometedora alternativa para solucionar los problemas actuales de la seguridad en el software. Como se ve en los casos de estudio analizados en este trabajo, la POA es útil tanto para especificar cuestiones de seguridad de bajo nivel, solucionando las fallas para implementar software seguro, como también para definir cuestiones de seguridad de alto nivel, solucionando fallas para implementar seguridad en el software, detalladas en la sección 2.

Además, por las características propias de la programación orientada a aspectos, las aplicaciones desarrolladas bajo este paradigma satisfacen naturalmente los objetivos propuestos en la sección 3 para obtener software seguro.

Por lo tanto, estamos ante la presencia de un paradigma, que si bien todavía sufre de cierto grado de inmadurez, muestra bases lo suficientemente sólidas como para pensar que se convertirá en la solución definitiva al problema de la seguridad en el software.

7. Referencias

[1] Doshi Shreyas, *"Software Engineering for Security: Towards Architecting Secure Software"*, Information and Computer Science Dept., University of California, Irvine, CA 92697,USA., abril de 2001.

[2] David Evans, Andrew Twyman, *"Flexible Policy-Directed Code Safety"*, in Proceedings of the 1999 IEEE Symposium on Security and Privacy. IEEE 1999.

[3] Premkumar T. Devanbu, Stuart Stubblebine, "*Software Engineering* for *Security: a Roadmap"*, in Proceedings of the conference on The Future of Software Engineering. 2000.

[4] Bart De Win, Bart Vanhaute, Bart De Decker, *"Security through Aspect Oriented Programming"*, in Proceedings of the 1st working conference on Network Security, noviembre de 2001.

[5] Bart De Win, Bart Vanhaute, Bart De Decker, *"Building frameworks in AspectJ"*, in Proceedings of the ECOOP 2001 Workshop on Advanced Separation of Concerns.

[6] Bart De Win, Bart Vanhaute, *"AOP, Security and Genericity"*, Distrinet, Dept. Computer Wetenschappen, K.U. Leuven, Bélgica, julio de 2001.

[7] Asteasuain Fernando, Bernardo Ezequiel Contreras, *"Programación Orientada a Aspectos: Análisis del Paradigma"*, Tesis de Licenciatura en Ciencias de la Computación. Universidad Nacional del Sur, noviembre de 2002.

[8] John Viega, J.T. Bloch, Pravir Chandra, *"Applying Aspect-Oriented Programming to Security"*, Cutter IT Journal, febrero de 2001.

[9] Gregor Kickzales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, John Irwin, *"Aspect-Oriented Programming"*, in Proceedings of the European Conference on Object-Oriented Programming (ECOOP), Finland. Springer-Verlag LNCS 1241. Junio 1997.

[10] John Viega, J.T. Bloch, Yoshi Kohno, Gary McGraw, *"ITS4: A Static Vulnerability Scanner for C and C++ Code"*, in Proceedings of the 16th Annual Computer Security Applications Conference (ACSAC 2000). IEEE, 2000.

[11] Roger Pressman, *"Ingeniería del Software: Un enfoque práctico"*, Mcgraw Hill, Madrid, 1993.

[12] Yvonne Coady, Gregor Kiczales, Michael Feeley, Norman Hutchinson, Joon Suan Ong, Stephan Gudmundson, "Exploring an Aspect-Oriented Approach to OS Code", University of British Columbia,2000.

[13] Yvonne Coady, *"Crosscutting the Great Divide: Exploring an Aspect-Oriented Approach to OS Code"*, propuesta de Tesis Doctoral, enero de 2001.

[14] Yvonne Coady, Gregor Kickzales, Mike Feeley, Greg Smolyn, *"Using AspectC to improve the modularity of path-specific customization in operating system code"*, in Proceedings of Joint ESEC and FSE-9, 2001.

[15] El sitio de AspectJ: www.aspectj.org . Palo Alto Research Center.

[16] R. Filman, T.Linden, *"SafeBoots: a Paradigm for Software Security Controls"*, in Proceedings of New Security Paradigms Workshop, Lake Arrowkead, CA., USA., 1996.

[17] C. Cowan et.al. *"StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks"*, in Proceedings of the 7th USENIX Security Symposium. ESENIX Association,1998.

[18] A. Ghosh, T. O'Connor, G.McGraw *"An Automated Approach for Identifying Potential Vulnerabilities in software"*, in Proceedings of the 1998 IEEE Symposium on Security and Privacy. IEEE 1998.

[19] A. Myers, *"Practical Mostly-Static Information Flow Control"*, in Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ACM, 1999.

[20] R. Pandey, B. Hashii, *"Providing Fine-Grained Access Control for Mobile Programs Through Binary Editing"*, Technical Report CSE-98-8. University of California, Davis, 1998.

[21] M. Blaze, J.Feigenbaurn, J. Lacy , *"Decentralized Trust Management"*, in Proceedings of the 17th IEEE Symposium on Security and Privacy. IEEE, 1996.

[22] S. Chiba, *"A MetaObject Protocol for C+-"*, in Proceedings of the 1995 Conference on Object-Oriented Programming.

[23] R. Stroud, Z. Wue, *"Using MetaObject Protocols to satifay Non-Functional Requirements"*, in Advances in Object-Oriented Metalevel Arquitectures and Reflection.