Organización del Computador 2

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

MASCHE
Memory Analysis Suite for Checking the Harmony of Endpoints

Grupo

Integrante	LU	Correo electrónico
Martínez Suñé, Agustín	630/11	zerolink92@gmail.com
Lascano, Nahuel	476/11	laski.nahuel@gmail.com
Vanotti, Marco	229/10	marcovanotti15@gmail.com
Palladino, Patricio	218/10	email@patriciopalladino.com

Índice

1. Introducción

El siguiente informe se presenta como complemento y documentación del trabajo practico final de la materia Organización del computador II.

El mismo fue realizado dentro del programa Mozilla Winter Of Security 2014[1]. Éste tiene como finalidad introducir a alumnos de carreras afines a la computación al mundo de la seguridad informática y la automatización. Para esto, el equipo de Security Automation[2] de Mozilla propuso una serie de proyectos para que distintos grupos de alumnos pudieran encarar, contando con un tutor provisto por ellos.

En este trabajo se llevó a cabo uno de dichos proyectos, que consistió en proveer de funcionalidades de inspección de memoria a la plataforma de análisis forense en tiempo real y respuesta a incidentes Mozilla Investigator[3], la cual será introducida a continuación.

1.1. Mozilla Investigator

Mozilla Investigator surge en respuesta a los cambios que se ha vivido en la infraestructura con el advenimiento del cloud computing. Mientras que antes los distintos servidores y servicios permanecían estáticos a lo largo del tiempo, sujetos a revisiones lentas y estrictas a la hora de ser alterados, hoy en día la virtualización permite un dinamismo y escala en la infraestructura que solía ser impensable. Este nuevo modelo demostró ser adecuado en un sinnúmero de situaciones, pero no viene libre de nuevos desafíos.

Uno de estos desafíos, y del cual Mozilla Investigator (MIG) se encarga, es manejar la seguridad y respuesta a incidentes en escalas nunca antes vistas de manera rápida y efectiva.

Para llevar a cabo esta tarea, MIG provee un sistema distribuido que facilita realizar distintos tipos de chequeos en los servidores de Mozilla. Esto se logra con una arquitectura distribuida, en la cual los distintos servidores cuentan con un agente corriendo de manera constante el cual lee una cola de queries a medida que esta se va llenando. Una vez que un query es leído, el agente actúa de manera acorde y envía su respuesta. El funcionamiento del sistema es orquestado por MIG Scheduler, uno o más servidores distinguidos, encargados de distribuir los queries a los agentes y comunicar las respuestas al usuario. De este modo un investigador de Mozilla puede conectarse al servidor de MIG y realizar consultas a miles de servidores en simultaneo en cuestión de segundos, permitiendo acelerar la reacción frente a incidentes de seguridad y agilizando las actividades cotidianas de security operations.

Si bien MIG es usado hoy en día en producción, se encuentra aún bajo desarrollo, principalmente con respecto a qué tipo de queries puede responder un agente. En la actualidad esto está limitado a responder consultas sobre la integridad del sistema de archivos (como chequear el nombre, contenido y hash de los mismos) y consultar el estado del stack de red del sistema donde está corriendo.

Existen proyectos para agregar soporte para distintos tipos de queries, como consultar y manejar reglas de los distintos firewalls, crear, bloquear y destruir cuentas de usuarios, analizar el trafico de la red, acceso a bajo nivel de los distintos componentes físicos del servidor, y muchos otros. En particular, uno de estos módulos, que pronto se encontrará en producción, es el desarrollo para este trabajo, el cual permite analizar la memoria principal de los procesos que se encuentran corriendo en el sistema, y que se introducirá a continuación.

1.2. MASCHE

No todo ataque o error de seguridad se manifiesta en el sistema de archivos de un servidor, sino que pueden residir únicamente en la memoria principal del mismo. A su vez, existen escenarios donde estos dos niveles de memoria no se encuentren en sincronía, haciendo que sea necesario poder acceder a ambos

[1] https://wiki.mozilla.org/Security/Automation/WinterOfSecurity2014
[2] https://wiki.mozilla.org/Security/Automation
[3] http://mig.mozilla.org

durante una investigación. MASCHE, por las siglas de Memory Analysis Suite for Checking the Harmony of Endpoints, es una respuesta a esto, desarrollada con el fin de ser integrado a MIG.

El análisis forense de memoria es un área muy amplia y compleja dentro de la respuesta a incidentes de seguridad informática, por lo que para este proyecto nos enfocamos en los siguientes casos de uso, de manera de terminarlo en un lapso de tiempo razonable:

1. Identificar procesos que estén utilizando versiones vulnerables de librerías dinámicas.
2. Encontrar los procesos que tienen secretos filtrados (por ejemplo claves privadas hechas publicas en algún incidente) en su memoria principal.
3. Localizar procesos con patrones de código que se sepan susceptibles a distintas vulnerabilidades. Notar que esto es necesario para detectar una librería vulnerable que fue linkeada estáticamente, y no únicamente para buscar código del programa en sí mismo.
4. Proveer una API unificada de acceso a memoria principal de procesos en Windows NT, Linux y OS X, que permita agregar nuevas funcionalidades a MIG de manera práctica.

2. Introducción a las plataformas soportadas

Como se menciono anteriormente, uno de los objetivos del proyecto MASCHE es que sea multiplataforma, soportando la familia NT de Microsoft Windows, Linux en sus versiones mas recientes, y OS X.

A continuación se presenta una breve introducción a cada una de estas plataformas, enfocándonos principalmente en los aspectos pertinentes al proyecto, pero sin entrar en demasiado detalle, ya que explicar el funcionamiento de tres sistemas operativos escapa a lo que se pretende hacer en este trabajo.

2.1. Linux

No haremos aquí una descripción detallada de la arquitectura de Linux ya que, si bien abunda el material al respecto, no fue necesario tener un conocimiento profundo de la misma para poder llevar adelante este proyecto, gracias a las distintas abstracciones que el sistema provee.

Entre las distintas formas de comunicarse con el kernel Linux que tiene un proceso de userland, una que nos fue especialmente útil para la implementación de MASCHE fue *procfs*. Este es un sistema de archivos especial implementado originalmente en UNIX V8, luego portado a SVR4 y Plan 9, para ser esta ultima implementación clonada luego por casi toda la familia UNIX. La finalidad del mismo es presentar cierta información del sistema y los procesos que están corriendo en forma de archivos y carpetas.

Hace falta remarcar que */proc* no es un directorio con archivos reales que están en disco sino que es un sistema de archivos especial que funciona como vía de acceso a información de las estructuras del kernel. Sin él, el acceso a esta información debería realizarse a través de syscalls y perdería el estándar de interfaz de acceso a archivo.

La implementación de Linux de procfs contiene un sinfín de archivos con información no solo de los procesos sino también del sistema en sí, pero para este proyecto bastó con utilizar los siguientes:

- /proc/[pid]/maps contiene las regiones de memoria mapeadas por el proceso y sus correspondientes permisos.
- /proc/[pid]/mem se usa para acceder a las páginas de la memoria del proceso. Es un archivo con el contenido de la memoria del proceso, donde un offset dentro del mismo corresponde a una posición de memoria en el address space el proceso.
- /proc/[pid]/status brinda información del proceso, por ejemplo, el pathname del ejecutable o el PID del proceso padre.

2.2. Windows NT

Windows NT es la familia actual de sistemas operativos de Microsoft, lanzada por primera vez en 1993, y reemplazando definitivamente al Windows anterior, basado en MS-DOS, con el lanzamiento de XP.

Windows NT fue el primer sistema operativo de 32 bits de Microsoft soportando distintas plataformas de hardware, pero abstrayéndose de ellas en una capa llamada HAL (por Hardware Abstraction Layer).

Sobre el HAL corren tanto los drivers como un kernel híbrido, ambos asentando las bases del sistema exponiendo los Executive Services (responsable de la mayoría del trabajo del sistema) al usuario, accesible a través de distintas APIs.

Si bien la idea original de NT era desarrollar una continuación de OS/2 que pueda correr aplicaciones POSIX, por motivos comerciales se decidió no abandonar la plataforma Windows, sino que se adaptó la API legacy de ésta para NT en WIN32. Es ésta ultima la que utilizamos para la realización del proyecto, ya que si bien no es open source cuenta con una excelente documentación online en la MSDN[4].

[4] https://msdn.microsoft.com/en-us/default.aspx

2.3. OS X

OS X es sin dudas la plataforma más particular de las tres soportadas. Para entender el por qué de su arquitectura y comprender su funcionamiento es necesario hacer una breve reseña histórica.

El kernel de Mac OS, XNU fue desarrollado originalmente para el sistema operativo NeXTSTEP de la empresa NeXT, luego adquirida por Apple, lo que llevó a su inclusión en la versión 10 de Mac OS, llamada actualmente OS X.

A su vez, XNU está basado en el kernel Mach. Éste surgió como un proyecto de investigación en Carnegie Mellon University sobre micro kernels, programación distribuida y computación en paralela. Como tal, presenta una arquitectura muy diferente respecto a los demás sistemas comerciales, basándose fuertemente en la orientación a objetos y pasajes de mensajes.

Con la integración de este kernel a Mac OS surgen distintos cambios. Los más importantes son que se abandona la arquitectura interna de microkernel, pero se mantiene expuesto el sistema de Inter Process Comunication en el cual se basaba, y se agrega una capa de abstracción POSIX basada en código de FreeBSD. De este modo, el sistema expone distintas APIs muy diferentes dependiendo a que parte del mismo se quiera acceder.

Durante el desarrollo del proyecto intentamos utilizar la API POSIX en cuanto era posible, pero lamentablemente es muy incompleta y no fue suficiente en la mayoría de los casos. Por esto mismo hubo que recurrir al sistema de IPC de Mach, el cual se encuentra notablemente indocumentado. Pero esto no nos impidió seguir adelante ya que, afortunadamente, XNU es open source en casi su totalidad.

Es destacable mencionar que, a diferencia de en UNIX, la comunicación con el kernel usando la API de Mach se hace a través de Remote Procedure Calls. Esto se logra utilizando un lenguaje especial, Mach Interface Language, para declarar los mensajes aceptados por cada modulo, y transformando los llamados a funciones (que serian syscalls en otro sistema) a envío de mensajes mediante un puerto. Debido a la falta de documentación al respecto, y a la antigüedad de la poca que se encuentra disponible, no explicaremos en este informe más sobre el tema, pero se puede investigar al respecto buscando las referencias presentes en la bibliografía.

3. Implementación de MASCHE

En esta sección se explica cómo esta implementado MASCHE, los módulos que lo conforman y su interacción con los distintos sistemas operativos.

Siendo MASCHE parte del proyecto Mozilla Investigator, distintas decisiones de diseño se vieron atadas a las de éste último. Por empezar, el mismo se encuentra desarrollado en Go[5], por ser el lenguaje en el que MIG está implementado. A su vez, MASCHE no pretende ser una aplicación en sí misma, sino que fue desarrollado como un conjunto de módulos de Go para ser utilizados en MIG con el fin de agregar soporte para nuevas funcionalidades en el mismo. Por ultimo, al momento de tomar decisiones que implicaron algún tipo de *trade-off* se priorizaron los objetivos de MIG aun cuando esto no fuese lo ideal para MASCHE como proyecto independiente.

Se describen a continuación cada uno de los módulos del proyecto, explicando cuando sea pertinente cómo se implemento cada cosa en los distintos sistemas operativos.

3.1. Módulo Process

El primer modulo a analizar es Process. El mismo provee una manera uniforme de describir a un proceso en las distintas plataformas y reservar los recursos necesarios para poder obtener información sobre estos.

La funcionalidad exportada es simplemente una interfaz Process, y funciones para listar, abrir (reservar los recursos mencionados) y cerrar (liberarlos) procesos.

3.1.1. Implementación en Linux

En el caso de Linux basta el *pid* de un proceso para tener una descripción del mismo. Abrir un Process en este caso simplemente revisa que tengamos permisos para leer los archivos correspondientes a este en procfs, y luego solo retorna el *pid*.

Luego, la información restante que puede pedirse sobre el proceso es sacada de *exe* y *status* dentro del directorio de procfs del proceso.

Para listar los procesos disponibles basta con inspeccionar los directorios de */proc*.

3.1.2. Implementación en Windows y OS X

A diferencia de Linux donde todo pudo ser implementado en Go, Windows y OS X requirieron utilizar llamadas a APIs en C. Si bien estas pueden hacerse directamente desde Go, una decisión que mantuvimos a lo largo de todo el proyecto es no abusar de esto, y exportar nuestras propias funciones de C en caso de que el código se vuelva muy engorroso.

En este caso creamos una interfaz común en C que tanto la implementación de Windows como la de OS X implementan. Ésta simplemente define cómo se representa, abre y cierra un proceso en cada sistema.

En el caso de OS X un proceso es representado por un *task_t* que es un renombre de un *mach_port_t*, un puerto a través del cual comunicarse con el proceso y poder inspeccionar distintas características de este. Abrir un proceso luego consiste simplemente llamar a *task_for_pid* y cerrarlo desalocar el task.
Para obtener información de un proceso y listar los mismos se utiliza en OS X la API POSIX del sistema, haciendo que implementar esto en Go no sea demasiado complejo. En particular las funciones *proc_pidpath* y *proc_listpids* de la librería *libproc* hacen justo lo necesario.

La API WIN32 ya provee un tipo *HANDLE* que es utilizado para describir un proceso, y funciones para abrirlos, cerrarlos y enumerarlos. Obtener el path de un proceso abierto requiere mas trabajo, pero puede obtenerse listando los módulos cargados por el mismo e inspeccionando el primero, que siempre

[5]Para una breve introducción a Go ver el primer apéndice.

es el ejecutable.

3.2. Módulo Memaccess

Memaccess, el modulo principal de MASCHE, exporta una misma interfaz para acceder a la memoria de un proceso (representado por un Process del modulo anterior) en cada una de las plataformas soportadas.

Para lograr esto hizo falta abstraerse de la estructuración y el manejo de la memoria en cada sistema. Esto se llevo acabo planteando la memoria de un proceso como secuencia ordenada de bloques contiguos de memoria.

Cada uno de estos bloques es representado por una estructura MemoryRegion, que posee la dirección de inicio del mismo (en el address space del proceso) y su tamaño. Para obtenerlos se debe utilizar la función NextReadableMemoryRegion, que dado un Process y una dirección retornará el próximo bloque de memoria legible a partir de la misma.

Una decisión que debimos tomar a la hora de hacer esto fue qué hacer cuando nos encontramos con un bloque de memoria sin permiso de lectura. Las opciones que contemplábamos eran las siguientes:

1. Ignorar este la memoria y buscar el próximo bloque legible.
   - Ventajas: Es fácil de implementar, y no interrumpe ni compromete de forma alguna al proceso inspeccionado.
   - Desventajas: Un atacante puede aprovechar esto para ocultar información de MASCHE manteniendo la memoria sin permiso de lectura la mayor cantidad de tiempo posible.
2. Cambiar los permisos, copiar la memoria y restaurar los permisos originales.
   - Ventajas: Permite leer toda la memoria.
   - Desventajas: El cambio de permisos se realiza mientras se está ejecutando el proceso, lo que puede llevarse a que se deje al mismo con permisos distintos a los que esperaba debido a esto. Si esto ocurre existe la posibilidad de que el proceso no funcione como es esperado o incluso a que el sistema operativo detenga su ejecución debido a un error de permisos.
3. Pausar el proceso, hacer lo mismo que en la opción anterior, y volver a correr el proceso.
   - Ventajas: Permite leer toda la memoria sin riesgos de comprometer la ejecución del proceso inspeccionado.
   - Desventajas: Daña la performance del sistema, ya que detiene la ejecución de los procesos inspeccionados.

Como se menciono anteriormente, priorizamos los objetivos de MIG a la hora de tomar este tipo de decisiones, y este debe interferir lo mínimo posible con el sistema que está analizando, por lo que se optó por la primer opción. Decidir no frenar los procesos resolvió también qué hacer cuando el layout de la memoria cambia mientras se la está analizando. MASCHE no hace nada en especial al respecto más que asegurarse de poder seguir adelante con el resto de su trabajo.

Una vez conocido el layout de la memoria mediante los MemoryRegion disponibles uno puede leer la misma con la función CopyMemory, que recibe un Process, una dirección de memoria dentro del mismo, y un buffer a llenarse con la memoria del proceso a partir de la dirección dada.

Con esto ya bastaría para poder inspeccionar la memoria de un proceso, pero sabiendo que el principal objetivo es buscar patrones conocidos en la misma se implementaron dos funciones que facilitan dicha tarea. Las mismas están basadas en la abstracción que explicamos anteriormente, por lo que tienen una sola implementación (en Go) para todas las plataformas.

La primera de estas, WalkMemory, trabaja sobre un Process y un buffer de bytes. Esta va llenando el buffer de manera secuencial con la memoria del proceso (iniciando donde dejó la lectura anterior) y llamando a una función provista por el usuario con el buffer y la dirección inicial de la memoria que contiene luego de cada vez que se llena. La segunda función, SlidingWalkMemory, es similar pero llena el buffer empezando por la posición del medio de la memoria que se había leído en la iteración anterior.

Se explican a continuación las implementaciones de NextReadableMemoryRegion y CopyMemory de cada plataforma.

3.2.1. Implementación en Linux

Es en la implementación de este módulo donde más se puede apreciar el poder de *procfs* y la simpleza que conlleva tener una interfaz de archivos para acceder a la información.

En este caso, NextReadableMemoryRegion simplemente lee una a una las lineas de el archivo *maps* dentro de /proc/[pid], buscando el primer memory map que contenga la dirección provista, uniéndolo a los que le siguen en caso de ser consecutivos y legibles, y retornando la información obtenida.

Sorprendentemente, CopyMemory fue aun más sencillo de implementar, pues solo hace falta abrir el archivo /proc/[pid]/mem y utilizar como offset de lectura la dirección provista.

3.2.2. Implementación en OS X

Nuevamente se creo una interfaz en C común para Windows y OS X, la misma sólo contiene las funciones get_next_readable_region copy_process_memory, que luego son llamadas desde Go en la implementación de las funciones descriptas.

En el caso de get_next_readable_region el sistema operativo cuenta con el llamado RPC mach_vm_region_recurse, que realiza una tarea muy similar. La principal diferencia es que la memoria en OS X se encuentra organizada de manera distinta, de manera recursiva, donde un mapa de memoria puede estar incluido como submapa de otro. Esto es utilizado normalmente para poder compartir en memoria librerías dinámicas entre distintos procesos de manera sencilla, y no es algo que suela utilizarse más allá de esto. Simplemente debe tenerse en cuenta que al pedir una region de memoria se puede obtener en realidad un submapa, y en este caso nos introducimos en el mismo para encontrar la región buscada. A su vez, igual que en el caso de Linux, unimos las distintas regiones consecutivas en una sola.

Copiar memoria resultó más sencillo, ya que se provee la función mach_vm_read_overwrite, que copia el contenido de la memoria del proceso a un buffer local, sin limitaciones impuestas por el alineamiento ni limites entre páginas.

3.2.3. Implementación en Windows

La implementación en Windows resultó asombrosamente similar. La principal diferencia, además de que no se usa un sistema de RPC, es que la memoria no está organizada de manera jerárquica, simplificando la implementación de get_next_readable_region.

Comparándola con la implementación de OS X, VirtualQueryEx cumple el papel de mach_vim_region_recurse y ReadProcessMemory de mach_vm_read_overwrite.

3.3. Módulo Memsearch

Memsearch se basa exclusivamente en la funcionalidad provista por Memaccess, por lo que bastó una única implementación en Go para todas las plataformas. Este modulo permite la búsqueda de distintos patrones en la memoria del proceso.

El modulo exporta únicamente las siguientes dos funciones:

- FindBytesSequence: Busca la primer aparición literal de una cadena de bytes a partir de una dirección dada en la memoria de un proceso. Su implementación consiste en llamar a memaccess.SlidingWalkMemory pasándole una función que busque de manera secuencial en cada buffer leído.

- FindRegexpMatch: Es similar a la anterior, pero en lugar de buscar ocurrencias literales utiliza una *regexp* para encontrar secciones de memoria que matcheen con esta. La implementación también es parecida, reemplazándose la búsqueda secuencial por su equivalente en expresiones regulares.

3.4. Módulo Listlibs

El último modulo de MASCHE, Listlibs, permite listar las librerías dinámicas cargadas por un proceso. Es importante notar que estas no son necesariamente las mismas que se pueden ver en el binario que está corriendo el mismo, sino que nos enfocamos en obtener la información acerca de cuales están efectivamente cargadas.

Listlibs exporta simplemente la función ListLoadedLibraries. Ésta retorna una lista de paths a las distintas librerías cargadas, y su implementación difiere completamente en cada plataforma.

3.4.1. Implementación en Linux

En el caso de Linux las librerías dinámicas son cargadas utilizando mmap, una syscall que mapea el contenido de un archivo en la memoria virtual del proceso. Gracias a esto basta con analizar el contenido de /proc/[pid]/map para obtener la información buscada.

La función ListLoadedLibraries en Linux simplemente parsea linea por linea el archivo mencionado, y en caso de contar con un path a un archivo mapeado, éste se agrega a la lista a devolver.

3.4.2. Implementación en Windows

Si bien no se cuenta con un sistema como procfs simplificando las cosas en Windows, la API WIN32 es sumamente completa y nos permitió obtener la lista de librerías cargadas sin demasiado problema.

La función EnumProcessModulesEx realiza algo muy similar a lo que estábamos buscando implementar, con la diferencia de que en lugar de retornar la ruta a los archivos cargados, se retornan HMODULEs. Éstos últimos son handles a un modulo cargado (representado con la dirección en memoria del mismo) que pueden utilizarse en distintos llamados de WIN32. En particular, los utilizamos con GetModuleFileNameEx para obtener el archivo que cada modulo representaba.

Fuera de esto, solo tuvimos que encargarnos de reservar los recursos necesarios para llamar a estas funciones, lo cual no es trivial y se logró mediante iteraciones de prueba y error, pero no dificultó demasiado la tarea.

3.4.3. Implementación en OS X

Contrario a las otras plataformas, esta tarea resultó sorprendentemente engorrosa en OS X. Como era de esperarse, no se encuentra disponible documentación oficial alguna sobre cómo realizar esto. Pero fue particularmente llamativo no encontrar funcionalidad que permita hacer esto en el código del kernel ni las herramientas open source liberadas por Apple.

Para empeorar aún más la situación, toda la documentación extraoficial encontrada en la web al respecto se encontraba ampliamente desactualizada y recurría a buscar patrones reconocibles dentro de toda la memoria del proceso para obtener algún indicio, no del todo fiable, de qué librerías habían sido cargadas. Si bien esto ultimo es aceptable en muchas situaciones, se encuentra en conflicto directo con

el objetivo de MIG de interferir lo menos posible la performance del sistema analizado.

Sin embargo, no encontrar información al respecto no puede ser un sinónimo de que esta funcionalidad no pueda implementarse de una forma eficiente, ya que es necesario para el funcionamiento normal del sistema operativo, en particular para el linker dinámico.

Afortunadamente, fuimos a parar con el llamado RPC *task_info*, que provee distinta información acerca de un proceso a quien lo llame. En este caso, nos interesó obtener información acerca del linkeo dinámico del proceso, llamada TASK_DYLD_INFO, el cual retorna la siguiente estructura:

```
struct task_dyld_info {
        mach_vm_address_t       all_image_info_addr;
        mach_vm_size_t          all_image_info_size;
        integer_t               all_image_info_format;
};
```

El primer campo de la misma es la dirección dentro de la memoria del proceso de la estructura *all_image_info*, que contiene información sobre todos los binarios cargados por el linker dinámico. Para obtener lo buscado nos queda entonces leer esta estructura desde la memoria del proceso.

El problema al hacer esto, es que el proceso inspeccionado puede ser de 32 o 64 bits, independientemente de cómo este compilado MASCHE, por lo que tenemos que soportar ambos. Por suerte, Apple agrego en versiones recientes del struct *task_dyld_info* su tercer parámetro, que nos provee esta información. Una vez obtenido esto, podemos leer una de las siguientes estructuras, dependiendo cual corresponda:

```
struct user32_dyld_all_image_infos {
        uint32_t        version;
        uint32_t        infoArrayCount;
        user32_addr_t   infoArray;
        user32_addr_t   notification;
        bool            processDetachedFromSharedRegion;
        bool            libSystemInitialized;
        user32_addr_t   dyldImageLoadAddress;
        user32_addr_t   jitInfo;
        user32_addr_t   dyldVersion;
        user32_addr_t   errorMessage;
        user32_addr_t   terminationFlags;
        user32_addr_t   coreSymbolicationShmPage;
        user32_addr_t   systemOrderFlag;
        user32_size_t   uuidArrayCount;
        user32_addr_t   uuidArray;
        user32_addr_t   dyldAllImageInfosAddress;
};

struct user64_dyld_all_image_infos {
        uint32_t        version;
        uint32_t        infoArrayCount;
        user64_addr_t   infoArray;
        user64_addr_t   notification;
        bool            processDetachedFromSharedRegion;
        bool            libSystemInitialized;
        user64_addr_t   dyldImageLoadAddress;
        user64_addr_t   jitInfo;
        user64_addr_t   dyldVersion;
        user64_addr_t   errorMessage;
        user64_addr_t   terminationFlags;
        user64_addr_t   coreSymbolicationShmPage;
        user64_addr_t   systemOrderFlag;
        user64_size_t   uuidArrayCount;
        user64_addr_t   uuidArray;
```

```
        user64_addr_t    dyldAllImageInfosAddress;
};
```

Es importante notar que las mismas no están exportadas en ningún header accesible al usuario de OS X, sino que pudimos conocer su definición inspeccionando el código de XNU. Por este motivo no utilizamos las mismas en el código de MASCHE, ya que podría incurrir en una violación del copyright de Apple. En lugar de esto, se utilizaron offsets y aritmética de punteros manual para leer lo restante.

Aclarado esto último, veamos qué nos interesa de estos structs. infoArray es la posición inicial de un arreglo con información de cada una de las librerías, e infoArrayCount el tamaño del mismo, por lo que solo necesitamos de estos campos.

El arreglo que queremos leer esta compuesto por elementos de la siguiente estructura según el modo en el qué este corriendo el proceso:

```
struct user32_dyld_image_info {
        user32_addr_t    imageLoadAddress;
        user32_addr_t    imageFilePath;
        user32_ulong_t   imageFileModDate;
};

struct user64_dyld_image_info {
        user64_addr_t    imageLoadAddress;
        user64_addr_t    imageFilePath;
        user64_ulong_t   imageFileModDate;
};
```

Finalmente imageFilePath es la dirección inicial del path de la librería representado como un string de C. Por lo que sólo queda copiar esta cadena de texto.

Vale aclarar que copiar la misma no es algo trivial, ya que se desconoce su tamaño, y se encuentra en la memoria del otro proceso. Pero luego de haber implementado Memaccess esto nos resultó familiar. Incluso podrían haberse utilizado las primitivas exportadas por Memaccess, pero al haber sido desarrolladas con el fin de exportarse a Go utilizarlas desde C resultaba muy engorroso.

Una vez copiados los strings, estamos en condiciones de devolver un arreglo con todos los paths, y la tarea queda completa.

4. Conclusiones

El desarrollo de este trabajo sirvió para adentrarse en nuevas plataformas, distintas a las vistas en el curso y a los demás del área de sistemas, y poder encontrar similitudes y diferencias entre estas.

Una de las cosas que vale la pena destacar, es el trabajo que esto implica, ya que alejarse del espacio de confort a la hora de trabajar conlleva no solo tiempo dedicado a investigación, sino a comprender y/o aceptar diferencias y limitaciones que en algunos casos parecen arbitrarias.

También es notable cómo los tres sistemas luego de una larga evolución en el tiempo convergieron a una arquitectura similar. Si bien utilizan distintas abstracciones para las mismas cosas, y manejan algunos asuntos de manera particular, el manejo de memoria de los mismos, lo que a MASCHE más concierne, se lleva a cabo de una manera similar. Creemos que esto se debe a que el diseño de los sistemas operativos se ve fuertemente influenciado por el hardware disponible en cada época, y por distintos requerimientos de performance y de compatibilidad por motivos comerciales.

Otra cuestión destacable que podemos concluir es que el desarrollo de una plataforma y las funcionalidades que la misma ofrece se encuentran atados a la cultura de la comunidad que lo lleva acabo y utiliza. Es notable como en Linux, una comunidad donde se valora ser abierto y la transparencia para con el usuario, se exporta la información que necesitábamos de forma tal que su utilización sea trivial. Windows, a pesar de que Microsoft lo mantenga como un proyecto de código cerrado, siempre ha sido fuertemente enfocado en proveer una plataforma de desarrollo que permita a otras empresas ofrecer sus productos que corran en este. Lo cual requiere proveer de soporte y funcionalidades para los desarrolladores de software de primer nivel, que se puede ver en la amplia API de WIN32 y en la excelente documentación disponible en la MSDN. Por último, Apple, más enfocado en controlar la experiencia del usuario de modo de asegurar la calidad de la misma, provee sólo herramientas de desarrollo de más alto nivel (exceptuando el desarrollo de drivers) que permiten llevar a cabo la mayoría de los proyectos sin problemas, pero en cuanto uno se aleja de este modelo la situación se complica debido a la falta de soporte y documentación oficial.

En definitiva, a pesar de estas diferencias muy interesantes didácticamente, se lograron cumplir los objetivos planteados para MASCHE dentro del tiempo estipulado, y el éste se encuentra pronto a ser utilizado en producción en la flota de servidores de Mozilla.

5. Bibliografía

- Russinovich, Mark et al. Windows Internals. Microsoft Press. 2012.
- Levin, Jonathan. Mac OS X and iOS Internals. John Wiley & Sons. 2013.
- Singh, Amit. Mac OS X Internals: A system approach. Addison Wesley Professional. 2006.
- Hale Ligh, Michael et al. The art of memory forensics. John Wiley & Sons. 2014.
- https://msdn.microsoft.com
- Código de fuente de LLDB: http://lldb.llvm.org/source.html
- Código de fuente de GDB: http://www.gnu.org/software/gdb/current/
- Código de fuente de XNU: http://www.opensource.apple.com/source/xnu/
- Código de fuente de procps: git://gitorious.org/procps/procps.git
- Código de fuente de Volatility: https://github.com/volatilityfoundation

6. Apéndices

6.1. Introducción a Go

Go es un lenguaje de programación de codigo abierto diseñado y desarrollado por Robert Griesemer, Rob Pike, y Ken Thompson en Google en el año 2007. Según su sitio web oficial[6] es un lenguaje de propósito general diseñado "pensando en programación de sistemas". Tiene una sintáxis similar a C, es estáticamente tipado, posee garbage collection y soporta concurrencia de manera primitiva.

Estos son algunos aspectos importantes para entender el lenguaje:

- **Paquetes:** los programas se organizan en paquetes, todo programa empieza su ejecución en el paquete *main*.
- **Punteros:** Go no tiene aritmética de punteros y la indirección de un puntero es transparente al programador.
- **Slices:** Una abstracción de secuencias de datos del mismo tipo. Es deseable usar este tipo de construcciones en lugar de directamente arreglos.
- **error:** Tipo de datos utilizado para el manejo de errores. Go está pensado para manejar errores explícitamente donde ocurren, por lo que es deseable que cada función devuelva, además de su resultado, un valor de *error*.
- **Go-rutina:** Una función ejecutándose concurrentemente con otras go-rutinas.
- **Channels:** Mecanismo de comunicación que permite el envío y recepción de datos entre dos go-rutinas.

Así se escribe un programa en Go:

```
package main

import (
    "fmt"
    "time"
)

func say(s string) {
    for i := 0; i < 5; i++ {
        time.Sleep(100 * time.Millisecond)
        fmt.Println(s)
    }
}

func main() {
    go say("mundo")
    say("hola")
}
```

El programa es un "Hola mundo" concurrente: imprime cinco veces "hola" mientras imprime cinco veces "mundo" desde otro hilo de ejecución.

6.2. CGO: invocando codigo C desde GO

Go provee una forma nativa de llamar a codigo C desde un programa escrito en Go[7]. Esto fue necesario en nuestro proyecto para las implentaciones en Mac OSX y en Windows ya que una buena parte de las funcionalidades sólo podía ser escrita en un lenguaje de bajo nivel como C. El pseudo-paquete *C* es el encargado de la comunicacion con codigo C, al importar este paquete se puede especificar el header para la compilación en C:

[6] https://golang.org/ref/spec#Introduction
[7] http://golang.org/cmd/cgo/

```
// #include <stdio.h>
// #include <errno.h>
import "C"
```

Así, a lo largo del codigo en Go podemos, a través de este paquete, invocar funciones en C:

```
package main

// typedef int (*intFunc) ();
//
// int
// bridge_int_func(intFunc f)
// {
// return f();
// }
//
// int fortytwo()
// {
//     return 42;
// }
import "C"
import "fmt"

func main() {
f := C.intFunc(C.fortytwo)
fmt.Println(int(C.bridge_int_func(f)))
// Output: 42
}
```

En contrapartida funciones de Go pueden ser exportadas para utilizarlas en codigo C de la siguiente manera:

```
//export MyFunction
func MyFunction(arg1, arg2 int, arg3 string) int64 {...}

//export MyFunction2
func MyFunction2(arg1, arg2 int, arg3 string) (int64, *C.char) {...}
```

6.3. Ejemplo de uso: Detectando Heartbleed

El 2014 fue un año sumamente activo en materia de seguridad informática, con vulnerabilidades e incidentes de escala nunca antes vistos. Algunos ejemplos de estos son la filtración de contenido adulto perteneciente a distintas celebridades proveniente de iCloud, que llevo a un cambio radical en la postura de Apple respecto a la privacidad de sus usuarios; el hackeo a Sony, que publico numerosas películas antes de su estreno y libero una gran cantidad de información sensible; POODLE, que puso fin a la historia de SSLv3; goto fail, poniendo en riesgo a todos los usuarios de productos de Apple; Shellshock, que permitía la ejecución remota de código en infinidad de servidores Linux. Pero sin dudas el ejemplo más anecdótico y dañino de estos es la vulnerabilidad conocida como Heartbleed (CVE-2014-0160 acorde a la base de datos MITRE).

Ésta se debió un error de manejo de memoria en la implementación de la extensión de heartbeat de TLS provista por la librería OpenSSL. Éste error causaba que un atacante pudiera leer toda la memoria de un sistema vulnerable sin dejar log alguno al respecto.

Este problema se encontraba en las versiones 1.0.1 a 1.0.1f, y 1.0.2-beta, por lo que el bug llevaba dos años en sistemas en producción cuando fue descubierto. Debido a esto, las versiones vulnerables de OpenSSL se encontraban en millones de servidores de todo el mundo. Como suele ser usual, un parche solucionando el problema salió casi en simultaneo con el anuncio del mismo, pero esto implicó recompilar software y/o actualizar OpenSSL en infinidad de lugares, haciendo difícil coordinar la tarea de modo de

asegurarse que se haga eficientemente y no se dejen servicios vulnerables por error.

Esto es un escenario donde contar con la ayuda de MIG, y en particular de MASCHE, podría haber simplificado mucho la mitigación del problema, y aumentado la confianza que el equipo de seguridad tiene en la ejecución de la misma.

Afortunadamente OpenSSL es utilizado normalmente cómo una librería dinámica, por lo que utilizando los módulos Process y Listlibs se podrá encontrar qué procesos deben reiniciarse luego de actualizar esta librería.

Pero esto no alcanza para estar seguros de que ningún proceso vulnerable está corriendo en el sistema, ya que la librería con el error puede haber sido linkeada estáticamente. Para encontrar este tipo de procesos se utiliza Memaccess, creando una firma de la vulnerabilidad y utilizando MASCHE para chequear si esta presente o no en cada uno de los procesos.

En el caso de Heartbleed, podemos crear una firma viendo cómo se soluciono el bug, y detectando si esta presente este arreglo o no. Para hacerlo, inspeccionamos el commit que lo arreglo, en particular nos enfocamos en la siguiente parte:

```
diff --git a/ssl/t1_lib.c b/ssl/t1_lib.c
index a2e2475..bcb99b8 100644
--- a/ssl/t1_lib.c
+++ b/ssl/t1_lib.c
@@ -3969,16 +3969,20 @@ tls1_process_heartbeat(SSL *s)
        unsigned int payload;
        unsigned int padding = 16; /* Use minimum padding */

-       /* Read type and payload length first */
-       hbtype = *p++;
-       n2s(p, payload);
-       pl = p;
-
        if (s->msg_callback)
                s->msg_callback(0, s->version, TLS1_RT_HEARTBEAT,
                        &s->s3->rrec.data[0], s->s3->rrec.length,
                        s, s->msg_callback_arg);

+       /* Read type and payload length first */
+       if (1 + 2 + 16 > s->s3->rrec.length)
+               return 0; /* silently discard */
+       hbtype = *p++;
+       n2s(p, payload);
+       if (1 + 2 + payload + 16 > s->s3->rrec.length)
+               return 0; /* silently discard per RFC 6520 sec. 4 */
+       pl = p;
+
        if (hbtype == TLS1_HB_REQUEST)
                {
                unsigned char *buffer, *bp;
```

Luego es cuestión de crear una firma de la función *tls1_process_heartbeat* en su versión vulnerable. Si bien esto no es algo trivial, es un problema cotidiano en la práctica, y no nos embarcamos en éste durante el proyecto. Pero a modo de ejemplo, generamos la firma para una version con símbolos de OpenSSL compilada con GCC en su version más reciente de Ubuntu 14.04. Para esto, luego de compilar la librería y linkearla estáticamente a nginx (a modo de ejemplo), utilizamos gdb para encontrar el código de la función:

```
(gdb) disassemble /r tls1_process_heartbeat
41 56                          push   %r14
41 55                          push   %r13
```

```
41 54                                           push   %r12
55                                              push   %rbp
48 89 fd                                        mov    %rdi,%rbp
53                                              push   %rbx
48 83 ec 10                                     sub    $0x10,%rsp
48 8b 97 80 00 00 00                            mov    0x80(%rdi),%rdx
4c 8b a2 30 01 00 00                            mov    0x130(%rdx),%r12
41 0f b6 5c 24 01                               movzbl 0x1(%r12),%ebx
41 0f b6 44 24 02                               movzbl 0x2(%r12),%eax
45 0f b6 2c 24                                  movzbl (%r12),%r13d
c1 e3 08                                        shl    $0x8,%ebx
09 c3                                           or     %eax,%ebx
48 8b 87 98 00 00 00                            mov    0x98(%rdi),%rax
48 85 c0                                        test   %rax,%rax
74 23                                           je     0x4b6252 <tls1_process_heartbeat+98>
44 8b 82 24 01 00 00                            mov    0x124(%rdx),%r8d
48 8b 97 a0 00 00 00                            mov    0xa0(%rdi),%rdx
49 89 f9                                        mov    %rdi,%r9
4c 89 e1                                        mov    %r12,%rcx
48 89 14 24                                     mov    %rdx,(%rsp)
ba 18 00 00 00                                  mov    $0x18,%edx
8b 37                                           mov    (%rdi),%esi
31 ff                                           xor    %edi,%edi
ff d0                                           callq  *%rax
66 41 83 fd 01                                  cmp    $0x1,%r13w
74 4f                                           je     0x4b62a8 <tls1_process_heartbeat+184>
31 c0                                           xor    %eax,%eax
66 41 83 fd 02                                  cmp    $0x2,%r13w
74 0e                                           je     0x4b6270 <tls1_process_heartbeat+128>
48 83 c4 10                                     add    $0x10,%rsp
5b                                              pop    %rbx
5d                                              pop    %rbp
41 5c                                           pop    %r12
41 5d                                           pop    %r13
41 5e                                           pop    %r14
c3                                              retq
90                                              nop
83 fb 12                                        cmp    $0x12,%ebx
41 0f b6 54 24 03                               movzbl 0x3(%r12),%edx
41 0f b6 4c 24 04                               movzbl 0x4(%r12),%ecx
75 e1                                           jne    0x4b6262 <tls1_process_heartbeat+114>
8b b5 a0 02 00 00                               mov    0x2a0(%rbp),%esi
c1 e2 08                                        shl    $0x8,%edx
09 ca                                           or     %ecx,%edx
39 d6                                           cmp    %edx,%esi
75 d2                                           jne    0x4b6262 <tls1_process_heartbeat+114>
83 c6 01                                        add    $0x1,%esi
c7 85 9c 02 00 00 00 00 00 00                   movl   $0x0,0x29c(%rbp)
89 b5 a0 02 00 00                               mov    %esi,0x2a0(%rbp)
eb bd                                           jmp    0x4b6262 <tls1_process_heartbeat+114>
0f 1f 00                                        nopl   (%rax)
44 8d 73 13                                     lea    0x13(%rbx),%r14d
ba 14 0a 00 00                                  mov    $0xa14,%edx
be 48 1a 6c 00                                  mov    $0x6c1a48,%esi
44 89 f7                                        mov    %r14d,%edi
e8 72 45 03 00                                  callq  0x4ea830 <CRYPTO_malloc>
49 89 c5                                        mov    %rax,%r13
c6 00 02                                        movb   $0x2,(%rax)
```

```
89 d8                                   mov    %ebx,%eax
49 8d 4d 03                             lea    0x3(%r13),%rcx
c1 e8 08                                shr    $0x8,%eax
49 8d 74 24 03                          lea    0x3(%r12),%rsi
48 89 da                                mov    %rbx,%rdx
41 88 45 01                             mov    %al,0x1(%r13)
41 88 5d 02                             mov    %bl,0x2(%r13)
48 89 cf                                mov    %rcx,%rdi
e8 eb 53 17 00                          callq  0x62b6d0 <memcpy>
48 8d 3c 18                             lea    (%rax,%rbx,1),%rdi
be 10 00 00 00                          mov    $0x10,%esi
e8 ed a6 05 00                          callq  0x5109e0 <RAND_pseudo_bytes>
44 89 f1                                mov    %r14d,%ecx
4c 89 ea                                mov    %r13,%rdx
be 18 00 00 00                          mov    $0x18,%esi
48 89 ef                                mov    %rbp,%rdi
e8 4a 7b 02 00                          callq  0x4dde50 <ssl3_write_bytes>
85 c0                                   test   %eax,%eax
78 56                                   js     0x4b6360 <tls1_process_heartbeat+368>
48 8b 85 98 00 00 00                    mov    0x98(%rbp),%rax
48 85 c0                                test   %rax,%rax
74 2a                                   je     0x4b6340 <tls1_process_heartbeat+336>
48 8b 95 a0 00 00 00                    mov    0xa0(%rbp),%rdx
45 89 f0                                mov    %r14d,%r8d
49 89 e9                                mov    %rbp,%r9
41 81 e0 ff ff 01 00                    and    $0x1ffff,%r8d
4c 89 e9                                mov    %r13,%rcx
bf 01 00 00 00                          mov    $0x1,%edi
48 89 14 24                             mov    %rdx,(%rsp)
ba 18 00 00 00                          mov    $0x18,%edx
8b 75 00                                mov    0x0(%rbp),%esi
ff d0                                   callq  *%rax
4c 89 ef                                mov    %r13,%rdi
e8 48 48 03 00                          callq  0x4eab90 <CRYPTO_free>
48 83 c4 10                             add    $0x10,%rsp
31 c0                                   xor    %eax,%eax
5b                                      pop    %rbx
5d                                      pop    %rbp
41 5c                                   pop    %r12
41 5d                                   pop    %r13
41 5e                                   pop    %r14
c3                                      retq
66 0f 1f 84 00 00 00 00 00              nopw   0x0(%rax,%rax,1)
4c 89 ef                                mov    %r13,%rdi
89 44 24 0c                             mov    %eax,0xc(%rsp)
e8 24 48 03 00                          callq  0x4eab90 <CRYPTO_free>
8b 44 24 0c                             mov    0xc(%rsp),%eax
e9 ed fe ff ff                          jmpq   0x4b6262 <tls1_process_heartbeat+114>
End of assembler dump.
```

Una vez obtenido esto, podemos utilizar memsearch.FindBytesSequence para encontrar el código maquina arriba listado en hexadecimal dentro de cada proceso. Es importante aclarar que esta firma dista de ser óptima, y está atada a una sola version de OpenSSL y una sola configuración de compilación. Usar memsearch.FindRegexpMatch puede ayudar a mejorar esto, pero la necesidad o no de hacerlo dependerá de las políticas de configuration management de la organización que esté utilizando MASCHE.

Se provee justo a este informe una maquina virtual que cuenta con versiones de OpenSSL y nginx vulnerables a Heartbleed para recrear todo lo acá explicado.