Quitando el Velo a la Memoria: Estructuras Ocultas y Malware

BIP-M, un Framework de Extracción de Información de Memoria

A. H. Di Iorio, G. M. Ruiz De Angeli, J. I. Alberdi, B. Constanzo, A. Podestá, M. Castellote

***Abstract*—Forensic analysis of a computer's volatile memory has become a major area of interest, with relatively few solutions outside closed commercial packages. This branch of digital forensics is full of challenges: the memory has a different layout depending on the Operating System, hardware architecture and even on features of the microprocessor. After extensive research, the BIP-M project is close to presenting a framework for the forensic analysis of RAM memory, and a tool built on top of it.**

***Keywords*— Análisis forense en memoria, estructuras ocultas, *malware, hooks*.**

I. INTRODUCCIÓN AL ANÁLISIS FORENSE EN MEMORIA

EN los últimos años el análisis forense de la memoria volátil se ha convertido en una herramienta fundamental de la informática forense por distintas razones. En primer lugar, el incremento en la capacidad de almacenamiento de los discos rígidos, combinado con la lentitud inherente a este tipo de tecnologías, impone tiempos de análisis cada vez mayores para estas fuentes de evidencia. Si a esto se suma la creciente utilización de tecnologías de cifrado para los medios de almacenamiento, la situación presenta serios obstáculos para superar.

El análisis de memoria en cambio es mucho más rápido para realizar, ya que los *dumps* de memoria son mucho más pequeños que una imagen de disco, al punto que en ocasiones es posible cargarlos en su totalidad dentro de la memoria de la computadora del investigador. En el caso que el sistema analizado utilizara tecnologías de cifrado, estas claves por lo general se encuentran almacenadas en la memoria, o es posible deducirlas en base a información que se encuentra presente en la misma. También en la memoria se encuentran artefactos propios de ella que es imposible encontrar en otras fuentes de evidencia digital: información de las conexiones del equipo, los procesos que se están ejecutando al momento de realizar la imagen (junto con toda la información que habían cargado), e información administrativa del sistema operativo, entre otras.

Las ventajas que brinda el análisis de memoria se contraponen con el profundo conocimiento que es necesario para poder realizar este tipo de investigaciones. Se debe conocer los esquemas, jerarquías y modos de memoria que brindan las distintas arquitecturas de hardware, cómo construyen sobre ellas sus módulos de administración de memoria los distintos sistemas operativos y cómo son las estructuras propias de cada sistema y aplicaciones que se desee analizar.

Considerando la potencia de estas técnicas, la prevalencia de opciones comerciales de código cerrado, y la necesidad de desarrollar conocimiento en esta área [1], es que se decidió investigarlas en el marco de un proyecto final de graduación, junto con el soporte del Grupo de Investigación en Sistemas Operativos e Informática Forense de la Universidad FASTA.

II. BIP-M: UN FRAMEWORK DE ANÁLISIS DE MEMORIA

El proyecto "BIP-M: Búsqueda de Información de Procesos en Memoria" tuvo por objetivo el estudio del marco teórico y la implementación de un *framework* de análisis de memoria volátil. El proyecto se planteó de forma que sus resultados formen una base sobre la cual nuevos proyectos puedan construir nuevo conocimiento y nuevas funcionalidades. Actualmente se encuentra en la fase de testeo y validación de las funcionalidades implementadas, que se limitaron al sistema operativo Windows 7 en sus ediciones de 32 y 64 bits. Ese Sistema Operativo fue elegido por ser un producto muy presente en el mercado, y a nivel técnico tiene grandes similitudes con las versiones más recientes.

El objetivo de la etapa de desarrollo, es el diseño e implementación de un *framework* extensible y adaptable, apoyándose en patrones de diseño orientados a objetos con el propósito de lograr bajo acoplamiento, alta cohesión y definiendo

claramente las diferentes responsabilidades de cada módulo. El *framework* pretende brindar un marco para analizar la memoria y encontrar cualquier tipo de objeto que se requiera, denominado genéricamente ***entidades***. Un objeto, ya sea un proceso, módulo, conexión, *driver*, archivo o un nuevo artefacto que se desee contemplar, puede considerarse un tipo especial de entidad y esto permite extender funcionalidad para la búsqueda de nuevos objetos.

Por otra parte, *BIP-M framework* ofrece desde su diseño la posibilidad de adaptarse a varios formatos de volcado de memoria, así como también a diferentes Sistemas Operativos con sus correspondientes versionados. Además, implementa persistencia en bases de datos MySQL, lo que permite un procesamiento de información más poderoso para lograr un análisis más profundo. Además, también permite desarrollar a futuro nuevas maneras de persistir la información, dado que esta funcionalidad está completamente desacoplada del resto de las clases.

Desde el punto de vista arquitectónico, *BIP-M framework* define *entities*, *dump managers*, *parsers* y *seekers*, cada uno de los cuales representan: objetos a buscar, objetos que tienen la responsabilidad y el conocimiento necesario para extraer información del volcado de memoria y *parsearla*, y objetos que deben obtener los datos que el usuario solicite de la información extraída previamente por los *managers*.

III. ESTRUCTURAS EN MEMORIA

Los artefactos que se pueden encontrar en memoria son variados y, por consecuencia, también las estructuras que los representan. En memoria es posible encontrar, entre otras cosas: procesos, hilos (*threads*), módulos, archivos, conexiones, *sockets*, entradas de registro, *drivers* y *timers*.

Cada uno de estos artefactos está representado por una o varias estructuras que poseen un determinado formato, el cual puede diferir de un sistema operativo a otro. El conocimiento y estudio de estas estructuras es el punto de partida para llevar adelante un análisis forense de cada objeto que presentes en memoria. Antes de avanzar en un análisis de estas características, se debe entender cada uno de los artefactos y cómo se relacionan entre sí para lograr obtener información coherente a partir de los datos identificados. A continuación se presentan ejemplos de las estructuras para Windows 7 de 32 bits.

III.A. PROCESOS

La estructura _EPROCESS[2] representa a los procesos en los sistemas operativos Windows, y contiene, los atributos que se presentan en este esquema, útiles en un análisis forense.

_EPROCESS

Windows 7 x86

```
+0x000 Pcb              : _KPROCESS
 +0x098 ProcessLock      : _EX_PUSH_LOCK
 +0x0a0 CreateTime       : _LARGE_INTEGER
 +0x0a8 ExitTime         : _LARGE_INTEGER
 +0x0b0 RundownProtect   : _EX_RUNDOWN_REF
 +0x0b4 UniqueProcessId  : Ptr32 Void
 +0x0b8 ActiveProcessLinks : _LIST_ENTRY
+0x140 InheritedFromUniqueProcessId : Ptr32
                                       Void
 +0x16c ImageFileName    : [15] UChar
 +0x188 ThreadListHead   : _LIST_ENTRY
 +0x198 ActiveThreads    : Uint4B
 +0x1a8 Peb              : Ptr32 _PEB
```

Entre estos atributos, el PCB (*Process Control Block*), se representa por una estructura _KPROCESS, embebida dentro del _EPROCESS, y almacena valores de vital importancia: el puntero al *Process Environment Block* (PEB) y el puntero a la cabeza de la lista de *threads* del proceso:

_KPROCESS

Windows 7 x86

```
+0x000  struct _DISPATCHER_HEADER Header;
+0x010  struct _LIST_ENTRYProfileListHead;
+0x018  ULONG32          DirectoryTableBase;
+0x02C  struct _LIST_ENTRYThreadListHead;
+0x078  struct_LIST_ENTRYProcessListEntry;
+0x088  ULONG32          KernelTime;
+0x08C  ULONG32          UserTime;
+0x094  UINT8            _PADDING0_[0x4];
```

Los siguientes atributos son algunos de los más importantes dentro de la estructura _KPROCESS:

- *CretateTime*: fecha de creación del proceso.
- *ExitTime*: fecha de terminación del proceso.
- *UniqueProcessID*: indica el id del proceso.
- *ActiveProcessLinks*: puntero a la cabecera de la lista circular doblemente enlazada de procesos activos.

- *InheritedFromUniqueProcessId*: identificador del proceso padre.

Los atributos *CreateTime* y *ExitTime* son estructuras de tipo *WindowsFileTime*[3], un formato de timestamp definido por Microsoft donde se utiliza un entero de 64 bits para contar intervalos de 100 nanosegundos desde el 1ro de Enero del año 1601. Dependiendo las características del sistema, si es de 32 o 64 bits y la *endianness*, la estructura puede ser representada de distinta forma. Por simplicidad se considerará un entero sin signo de 64 bits.

III.B. MÓDULOS

Los módulos, por ejemplo las DLLs, se almacenan en memoria con una estructura denominada _LDR_DATA_TABLE_ENTRY, la cual contiene los siguientes atributos:

Windows x86

```
+0x000 InLoadOrderLinks : _LIST_ENTRY
+0x008 InMemoryOrderLinks : _LIST_ENTRY
+0x010 InInitializationOrderLinks:_LIST_ENTRY
+0x018 DllBase          : Ptr32 Void
+0x01c EntryPoint       : Ptr32 Void
+0x020 SizeOfImage      : Uint4B
+0x024 FullDllName      : _UNICODE_STRING
+0x02c BaseDllName      : _UNICODE_STRING
...
+0x070 LoadTime         : _LARGE_INTEGER
```

Los primeros 3 atributos son punteros a diferentes listas enlazadas de módulos cargados en memoria, que se mencionaran más adelante. Además de esos punteros, los atributos de mayor interés son:

- *DllBase*: es el puntero a la dirección de memoria donde está cargado el módulo propiamente dicho.
- *FullDllName*: es la dirección de memoria donde se encuentra almacenado el nombre del módulo completo (*path* + *filename*).
- *BaseDllName*: dirección de memoria donde se encuentra almacenado el nombre del módulo.

Estos últimos dos atributos, apuntan a una estructura _UNICODE_STRING, que lleva el largo de la cadena, el tamaño máximo y un buffer de caracteres.

IV. CÓMO SE ALMACENAN LAS ESTRUCTURAS EN MEMORIA

La memoria se divide en *kernel pools*, que a su vez se dividen en *pools*, donde se almacenan las estructuras vistas anteriormente o distintos tipos de datos. Un *pool* de memoria puede verse como un conjunto de capas que se apilan y se representa con una estructura *Kernel Pool Layout*:

TABLA I. KERNEL POOL LAYOUT

ESTRUCTURA	WIN7 X86	WIN7 X64	Presencia
_POOL_HEADER	8 elementos, 0x8 bytes	9 elementos, 0x10 bytes	Obligatorio
_OBJECT_HEADER_PROCESS_INFO	2 elementos, 0x8 bytes	2 elementos, 0x10 bytes	Opcionales
_OBJECT_HEADER_QUOTA_INFO	4 elementos, 0x10 bytes	5 elementos, 0x20 bytes	
_OBJECT_HEADER_HANDLE_INFO	2 elementos, 0x8 bytes	2 elementos, 0x10 bytes	
_OBJECT_HEADER_NAME_INFO	3 elementos, 0x10 bytes	3 elementos, 0x20 bytes	
_OBJECT_HEADER_CREATOR_INFO	4 elementos, 0x10 bytes	4 elementos, 0x20 bytes	
_OBJECT_HEADER	12 elementos, 0x20 bytes	12 elementos, 0x38 bytes	Obligatorio
OBJECT (_EPROCESS, _FILE_OBJECT...)			Obligatorio

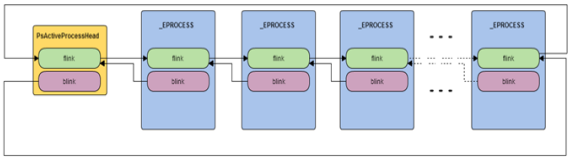
Para estructura el encabezado almacena un *tag* que permite identificar qué tipo de objeto se almacena en ella:

TABLA II. POOL-TAG DE ESTRUCTURAS EN MEMORIA

Estructura	Etiqueta en memoria (tag)
Process	Proã
Thread	Thrä
File	Filå
SymbolicLink	Linö
Driver	Driö
Desktop	Desö
WindowStation	Winä
TCPConnection	TcpE
TCPSocket	TcpL
UDPConnection	UdpA
RegistryEntry	CM10

Más allá que las estructuras en memoria ocupen uno o varios *pools* de memoria existe una relación lógica entre distintos artefactos: los procesos activos y los módulos se vinculan formando parte de una lista circular doblemente enlazada, donde cada uno de ellos es un ítem de la misma.

Figura 1. Lista doblemente enlazada de procesos activos



En la Fig. 1 se ilustra la lista de procesos activos, que permite recorrer los procesos a través de los atributos *FLINK* y *BLINK* de la estructura _EPROCESS. *PsActiveProcessHead* es la cabecera de dicha lista y se la puede encontrar como atributo del KDBG (*KernelDebugger Data Block*), una estructura de Windows que lleva información administrativa del sistema.

PsActiveProcessHead es una estructura del tipo _LIST_ENTRY que contiene dos punteros, FLINK y BLINK. Cada atributo *FLINK* apunta al atributo *FLINK* del ítem que le sigue en la lista, y cada atributo *BLINK* apunta al atributo *FLINK* del ítem que lo precede en la lista.

Los módulos activos también se vinculan entre sí, pero a través de 3 listas circulares doblemente enlazadas, que parten también del KDBG. Estas listas difieren una de otra en el orden de los elementos: la lista *InLoadMemoryOrder* se organiza en base al orden en que se cargan los módulos, *InMemoryOrderLinks* en base al orden que se encuentran en memoria, e *InInitializationOrder* según el orden de inicialización.

V. ESTRUCTURAS OCULTAS Y MALWARE

Los procesos ocultos, a diferencia de los procesos activos mencionados anteriormente, son aquellos procesos que no mantienen sus referencias en la lista de procesos, por lo que al recorrer la misma, no pueden ser identificados. Esta situación usualmente se asocia con *malware* o *rootkits*, y se ilustra en la siguiente figura:

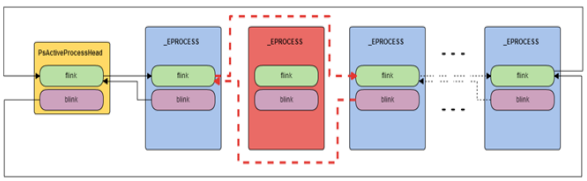


Figura 2. Proceso oculto, fuera de la lista de procesos activos

De igual manera sucede con los módulos, se puede ocultar uno de ellos quitándole las referencias a la lista de módulos activos, encabezada por la estructura *PsActiveModuleList*.

Es en este punto es donde resulta de gran utilidad lo visto en el capítulo anterior sobre los *pools* de memoria y sus encabezados para determinar qué tipo de estructura se encuentra almacenado en éste. Este conocimiento permite llevar adelante una técnica de búsqueda de estructuras basada en los *tags* de los diferentes artefactos en sus respectivos encabezados de los *pools* que ocupan[4, 5]. Por ejemplo para encontrar procesos se debe buscar el *tag* “Proã” dentro del volcado de memoria a analizar. Una vez encontrado, evaluar si se trata de un pool que contiene una estructura _EPROCESS y descartar falsos positivos. Para ello, es necesario tener en cuenta los siguientes detalles:

a) La pila de estructuras que puede haber dentro del pool (ver TABLA I), para evaluar en qué offset se puede encontrar el objeto _EPROCESS.

b) Qué longitud tiene la estructura OBJECT.

c) Qué datos esperamos según el offset en la estructura OBJECT.

Respecto al punto c), en el caso de los procesos, un criterio a tener en cuenta para descartar falsos positivos es el valor DTB (*DirectoryTable Base*) que posee cada proceso dentro de su estructura _KPROCESS. Dicho valor debe coincidir con el valor que posee un proceso conocido, como es el caso del proceso *System*. Sin embargo, si el volcado de memoria a analizar es del tipo *CrashDump*, en el encabezado del archivo se cuenta con dicho valor, con lo cual la validación se puede hacer directamente con ese valor[6].

Una vez recorrido el *dump* de memoria e identificados todos los objetos de tipo _EPROCESS, se puede construir una lista de los mismos que se compara con la lista de procesos activos para encontrar los procesos ocultos.

Si bien es una técnica costosa, resulta sumamente útil para detectar estructuras ocultas, ya que recorre todo el contenido disponible de la memoria y se independiza de las estructuras lógicas como las listas de procesos o módulos.

Si el *malware* simplemente se extrajera de la lista de procesos activos, no podría ejecutarse más. Previo a ocultarse, debe corromper alguna llamada del sistema a través de la cual pueda ganar control de ejecución en el sistema, y privilegios elevados. Esta técnica se denomina *hooking* y existen diferentes técnicas para hacerlo[7]:

a. IDT *Hooking* (sobre todo en versiones más antiguas de Microsoft Windows)
b. SYSENTER *Hooking*
c. SSDT *Hooking*

Actualmente, el *hooking* a la SSDT (*SystemServiceDispatchTable)* es ampliamente utilizado por los *rootkits*. Podríamos nombrar tres formas de realizar *hooking* a la SSDT:

1. Reemplazar punteros de la SSDT: se reemplazan punteros en la SSDT para realizar *hooks* a funciones específicas.
2. Duplicar la tabla SSDT: esto permite que el *malware* pase desapercibido más fácilmente, dado que crea una copia de la SSDT original, realiza *hook* sobre las funciones que desea y actualiza la referencia a la *ServiceTable* del/los *thread/s* que desea. Dado que la mayoría de las herramientas analiza la existencia de *hooks* verificando las ServiceTable originales y no las copias, la detección de este tipo de *hooks* en muchos casos falla.
3. Realizar *hook* dentro de una entrada válida de la SSDT (*Inline Hook*): en este caso, el *hook* se realiza modificando una instrucción dentro de una función existente y válida de la SSDT.

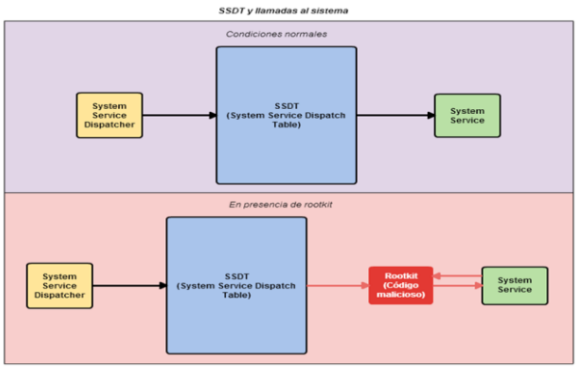


Figura 3. *Hook* a la SSDT

La Figura 3 muestra cómo el *rootkit* reemplaza la entrada en la SSDT una llamada al sistema por un puntero a su propio código.

Un ejemplo práctico de cómo lograr un *hook* a la SSDT, consiste en deshabilitar el WP bit (*WriteProtection bit*) del registro CR0 del procesador. Cuando este bit se encuentra en 0, cualquier código en el *kernel* tiene acceso de lectura/escritura en todas las páginas de memoria. Esto se puede lograr simplemente con instrucciones de *assembler*. De esta forma, se puede modificar la SSDT obteniendo permisos de escritura.

Otra alternativa, que no requiere la alteración del WP bit, consiste en modificar la SSDT sin necesidad de habilitar los permisos de escritura sobre la memoria.

Se accede a la SSDT desde la estructura *KeServiceDescriptorTable*, un *array* que posee 4 valores DWORD. El primer valor es la dirección virtual de memoria que apunta directamente a la SSDT y el tercer valor indica la cantidad de funciones que contiene la SSDT.

Si bien la zona de la SSDT está protegida contra escritura, la zona de memoria que contiene al *KeServiceDescriptorTable* que apunta a la SSDT no lo está. Se puede acceder a la SSDT y hacer una copia exacta de todas las direcciones de las *APIs* contenidas en la SSDT original sobre una nueva zona de memoria. Luego se realiza el *hook* ala *API* elegida y luego se modifica el primer DWORD de la estructura *KeServiceDescriptorTable*con con la dirección de la nueva SSDT[8][9].

En el caso de Windows 7, para analizar si existen *hooks* a la SSDT, es necesario adoptar dos alternativas diferentes según la arquitectura. En Windows 7 x86 cada *thread* posee un puntero a una determinada *ServiceTable*, por lo que es posible obtener todos los punteros únicos a las *ServiceTable* listando previamente todos los *threads* y analizando su estructura. En particular, el TCB (*ThreadEnvironment Block*) es el que posee el puntero a una *ServiceTable*. Veamos su ubicación dentro de la estructura _ETHREAD y, a continuación, el detalle de la estructura que representa al bloque, llamada _KTHREAD:

```
typedefstruct _KTHREAD
{
/*0x000*/     struct DISPATCHER_HEADER Header;
/*0x090*/     struct KTIMER Timer;
/*0x0BC*/     VOID*        ServiceTable;
/*0x18C*/     VOID*        Win32Thread;
/*0x1E0*/     struct LIST_ENTRYThreadListEntry;
/*0x1E8*/     struct LIST_ENTRYMutantListHead;
/*0x1F4*/     struct KTHREAD_COUNTERS*
ThreadCounters;
/*0x1F8*/     struct XSTATE_SAVE* XStateSave;
          }KTHREAD, *PKTHREAD;
```

De esta manera, analizando todos los *threads*, podemos ubicar los punteros a las SSDT y, desde estos, analizar la existencia de algún tipo de *hook*.

VI. CONCLUSIÓN

Además de los *rootkits*, existen otros tipos de *malware*, como *backdoors*, troyanos y *botnets*. Cualquiera sea su tipo, debe considerarse que el *malware* moderno siempre busca ocultarse, sin embargo, deja indefectiblemente rastros en la memoria. Es así que se torna fundamental hacer un análisis forense de la memoria volátil para poder detectar la presencia de *malware*.

BIP-M *framework* permite realizar búsqueda de estructuras ocultas y análisis de *hooks*, y brinda la posibilidad de extender su funcionalidad, agregando *parsers* y *seekers* que puedan colaborar para realizar búsqueda de nuevos objetos y/o distintos tipos de *malware*. En su implementación actual, el *framework* demuestra que es apto para este tipo de análisis y extensibilidad, lo que deja abierta la posibilidad de incorporar nuevas clases para soportar tanto nuevas versiones como otros sistemas operativos.

El conocimiento adquirido en esta temática ha fortalecido las bases del Grupo de Investigación en Sistemas Operativos e Informática Forense de la Universidad FASTA, permitiendo definir futuros proyectos que continuarán esta línea.

Se espera que la nueva herramienta desarrollada, colabore con la tarea de los analistas forenses facilitando la tarea de análisis de memoria, y con el mundo académico, desde el aporte de un framework factible de ser utilizado y extendido.

REFERENCIAS

[1] A. H. Di Iorio, R. E. Sansevero, M. Castellote, A. Podestá, F. Greco, B. Constanzo, J. Waimann., «"Determinación de aspectos carentes en un Proceso Unificado de Recuperación de Información digital."» p. 12, 2012.

[2] M. Russinovich, D. A. Solomon, A. Ionescu, «Windows Internals Part 1 6th Edition,» 2012.

[3] Microsoft, «Windows Dev Center,» [En línea]. Available: https://msdn.microsoft.com/en-us/library/windows/desktop/ms724284%28v=vs.85%29.aspx. [Último acceso: 20 Abril 2015].

[4] A. Schuster, *Searching for processes and threads in Microsoft*, Deutsche Telekom AG, Friedrich-Ebert-Allee 140, D-53113 Bonn, Germany, 2006.

[5] M. H. Ligh, A. Case, J. Levy, A. Walters, «The Art of Memory Forensics: Detecting Malware and Threats in Windows, Linux, and Mac Memory,» 2014, pp. 129-142.

[6] Q. Zhao, T. Cao, *Collecting Sensitive Information from Windows*, 2009.

[7] M. H. Ligh, A. Case, J. Levy, A. Walters, «The Art of Memory Forensics: Detecting Malware and Threats in Windows, Linux, and Mac Memory,» 2014, pp. 390-395.

[8] D. P. Castro, «Un informático en el lado del mal,» [En línea]. Available: http://www.elladodelmal.com/2014/03/ssdt-hooking-v2.html. [Último acceso: 24 Abril 2015].

[9] G. Hoglund, J. Butler, Rootkits: Subverting the Windows Kernel, 2015.

Ana Haydee Di Iorio es Ingeniera en Informática de la Universidad FASTA y Directora del InFo-Lab: Laboratorio de Investigación y Desarrollo de Tecnología en Informática Forense. Es miembro de la Comisión Asesora de la Red de Laboratorios Forenses de Ciencia y Tecnología del Ministerio de Ciencia Tecnología e Innovación Productiva de la República Argentina. Ha participado, presidido y dictado conferencias en numerosos congresos nacionales e internacionales.

Gonzalo Ruiz de Angeli es Técnico en Informática de la Universidad FASTA, estudiante avanzado de Ingeniería en Informática de la Universidad FASTA y participa como investigador alumno en el Grupo de Investigación en Informática Forense. Proyecto final en curso: BIP-M.

Juan Ignacio Alberdi es Técnico en Informática de la Universidad FASTA y estudiante avanzado de Ingeniería en Informática de la Universidad FASTA. Proyecto final en curso: BIP-M.

Bruno Constanzo es Ingeniero en Informática, Investigador del InFo-Lab y docente e Investigador del Grupo de Investigación en Sistemas Operativos e Informática Forense de la Facultad de Ingeniería de la Universidad FASTA.

Ariel Podestá es Ingeniero en Informática de la Universidad FASTA, profesor Adjunto de la cátedra de Informática Aplicada de la Licenciatura en Criminalística, en la Facultad de Ciencias Jurídicas y Sociales, y participa como investigador en el Grupo de Investigación en Informática Forense.

Martín Catellote es Ingeniero en Informática de la Universidad FASTA, jefe de Trabajos Prácticos en la cátedra de Sistemas Operativos, en la Facultad de Ingeniería de la Universidad FASTA, y participa como investigador en el Grupo de Investigación en Informática Forense.