Uno de los objetivos que en el análisis de un binario malicioso suele ser más difícil de obtener es el del descubrimiento de todas las funcionalidades que posee. Si además estas funcionalidades sólo se ejecutan a discreción de los atacantes a través de su centro de control, la cosa se complica. Por diversas razones, muchas veces no podemos llevar a cabo un análisis dinámico completo, como por ejemplo por la caída de la infraestructura del malware o el aislamiento de la muestra para evitar el contacto con el C&C. En estos casos suele ser más lento el análisis de la interacción entre el servidor del atacante y la muestra, ya que hay que crear un servidor ficticio o estar continuamente parcheando/engañando a la muestra para llevarla por todos los distintos caminos que queremos investigar. Dependiendo del tamaño y complejidad del código analizado o del objetivo del análisis, esta tarea puede variar su dificultad y extensión en el tiempo.

Voy a proponer un ejemplo de estudio de las funcionalidades de un RAT ficticio que pueden ser ejecutadas según las órdenes que reciba desde su panel de C&C. Nuestro objetivo sería crear un servidor que simule ser el del atacante. Para ello hemos de comprender el protocolo de comunicación entre el servidor y la muestra instalada en el dispositivo de la víctima.

En lugar de analizar todo el funcionamiento interno de la muestra utilizando las típicas herramientas de desensamblado y depuración, voy a delegar la responsabilidad de parte del análisis en una nueva herramienta que llevaba tiempo queriendo probar: ‘[angr](http://angr.io/)’. ‘angr’ es un entorno de trabajo para el análisis de binarios que hace uso tanto de análisis estático como dinámico simbólico del código. Este herramienta puede ser utilizada con distintos fines que se enumeran en su sitio web en el que además se muestran infinidad de ejemplos. Para esta entrada nos vamos a centrar en la ejecución simbólica, que puede definirse como el análisis de un programa para determinar qué condiciones de entrada han de darse para ejecutar diferentes partes de su código.

Debido a la complejidad de la herramienta, con el objetivo de facilitar su entendimiento voy a proponer un análisis sencillo de un código creado para ilustrar su uso:

<https://gist.github.com/halos/15d48a46556645ae7ff2ecb3dfc95d73>

El código simula ser un RAT que recibe instrucciones de un C&C desde la función ‘call\_home’, inicializando el valor de una variable global ‘c2\_resp’ que contendrá una orden del supuesto C&C. Después se ejecuta la función ‘exec\_order’ que, según la información contenida en esta estructura, llevará a cabo unas operaciones u otras.

Los valores que pueden tomar los distintos campos de la estructura están definidos como constantes al principio del código. Contienen valores aleatorios para no dar pistas a los posibles analistas del significado de cada uno.

Sin embargo, a la hora de enfrentarnos a un malware en el laboratorio, carecemos del código fuente que lo generó, y solo disponemos del binario compilado. Esto sería todo lo que vemos de la función ‘exec\_order’ si compilamos el anterior código en C:



Con un poco más de análisis se puede llegar a obtener un código en ensamblador algo más fácil de digerir:



Este código ya muestra una estructura algo más sencilla de comprender, pero aún tratándose de un código medianamente organizado, su longitud puede dificultar su análisis como puede verse en la siguiente imagen cuando profundizamos un poco más en la función:



Sin embargo, en el malware del día a día, la lógica de este tipo de programas no suele ser tan clara y no porque sus creadores no sepan programar, sino porque se busca dificultar el estudio que saben que se va a realizar de su creación.

Para estudiar esta muestra con ‘angr’ voy a necesitar conocer el valor de 4 direcciones de memoria para orientar a la herramienta sobre dónde y cómo trabajar:

1. ¿Dónde comienza el código que quiero estudiar?
2. ¿Dónde termina?
3. ¿A qué partes del código quiero llegar?
4. ¿Con qué variables puedo jugar?

Las dos primeras servirán a ‘angr’ para delimitar la zona de trabajo sobre la que poder evaluar todos sus posibles caminos y evitar así la pérdida de tiempo analizando caminos que no nos sirven en este estudio. En el tercer punto tendremos varias direcciones que marcarán las zonas de interés que quiero alcanzar y sobre las que se centrará el estudio. La 4ª apuntará al ‘buffer’ proveniente del servidor que, según contenga unos valores u otros, hará al programa alcanzar las distintas zonas de interés marcadas en el punto 3.



Puede verse en el código del primer grafo mostrado, que la función a analizar comienza en la dirección 0x0804844d, por lo que se le ha asignado a la constante START\_ADDR. En la lista FIND\_ADDRS se almacenan las direcciones a encontrar. Por ejemplo, se ha añadido la dirección 0x8048483 de la instrucción “push str.Creando\_fichero:\_\_s”, de la rama de ejecución para crear un fichero (bloque con la dirección 0x804847b).

Sin entrar en detalles profundos sobre el resto del código, solamente indicar que inicializo el analizador con el ejecutable compilado ‘a.out’ del [código en C](https://gist.github.com/halos/15d48a46556645ae7ff2ecb3dfc95d73) pasándole la dirección de inicio. Es necesario también señalar el buffer de respuesta del servidor (BUFF\_ADDR) que será la variable sobre la que trabajar y le indico las direcciones donde parar el análisis. Finalmente le indicaré que comience a explorar el código con todas estas indicaciones e imprimiré los resultados de una forma que más o menos  pueda ser entendida por un humano:



La primera entrada que aparece viene a decir que si el C&C quiere ordenar al bot que borre un fichero (acción que se ejecuta en la dirección 0x80485050), mandará un ‘buffer’ con los bytes “00 E9 BB 64 D4 D4 01 1F  AF 78 09 6E 00 00 00 00 00 00 00 00”.

A continuación aparece, a modo de explicación/curiosidad, cómo se ha llegado a esa dirección jugando con el contenido de la dirección de memoria BUFF\_ADDR. Cabe indicar que ‘angr’ trabaja a nivel de bits, de ahí los índices tan altos que aparecen para un buffer de solo 20 bytes (‘c2\_resp\_0\_160[159:128]’).

La primera restricción indica que el 4º DWORD ha de contener el valor 0x64bbe900 ([T\_FILESYSTEM](https://gist.github.com/halos/15d48a46556645ae7ff2ecb3dfc95d73#file-angr_example_bot-c-L8) como puede verse en el código fuente en C). Sin embargo, sobre el contenido del tercer byte aparecen tres restricciones: que sea distinto de 0xaff80c17 y de 0xc6e6ef6b, y además debe valer 0x1f01d4d4. Aunque con la tercera restricción sería suficiente, esto ocurre por una razón. Al simular la ejecución, ‘angr’ ha de evitar entrar en ciertas ramas y la condición es que el tercer byte no tenga un valor que le haga entrar en distintos ‘if’. Si miramos el código en C creado, el 3er byte, que se corresponde con el campo ‘fs\_order’, se compara primero con las constantes FSO\_CREATE ([línea 55](https://gist.github.com/halos/15d48a46556645ae7ff2ecb3dfc95d73#file-angr_example_bot-c-L55)) y FSO\_WRITE ([línea 67](https://gist.github.com/halos/15d48a46556645ae7ff2ecb3dfc95d73#file-angr_example_bot-c-L67)), y finalmente entra en la rama de borrado cuando es igual a la constante FSO\_DELETE ([línea 71](https://gist.github.com/halos/15d48a46556645ae7ff2ecb3dfc95d73#file-angr_example_bot-c-L71)).

Esta información nos ayudará a comprender el protocolo de comunicación para obtener nuevos IOC, montar un servidor que simule ser el atacante, modificar el buffer de la muestra en vivo y de esta forma ir a tiro hecho centrándonos en ciertas funcionalidades, etc.