Malware Analysis

PETYA 분석

MBR Locker PETYA 에 대하여 알아보자.

4/2/2016

By Kali KM

SAMPLE DETAILES

MBR Ransomware

File Name	Petya.exe
File Size	226 KB
CRC32	488C77E7
MD5	AF2379CC4D607A45AC44D62135FB7015
SHA-1	39B6D40906C7F7F080E6BEFA93324DDDADCBD9FA

Extracted #1

File Type	DLL
File Size	47 KB
CRC32	3D93BCF5
MD5	7899D6090EFAE964024E11F6586A69CE
SHA-1	9078E741D6D66FB6B4920878F0B7CD6A0F8B1CC7

목차

그림

1 개요

PETYA 는 MBR Locker 로 랜섬웨어의 유행과 함께 다시 나타나고 있다. 2016 년 3 월 25 일부터 MBR 영역을 변조하여 정상적인 부팅이 불가능 하도록 만든다. 또한 변조된 MBR 영역에는 기존의 랜섬웨어와 같이 아래의 메시지를 출력한다.

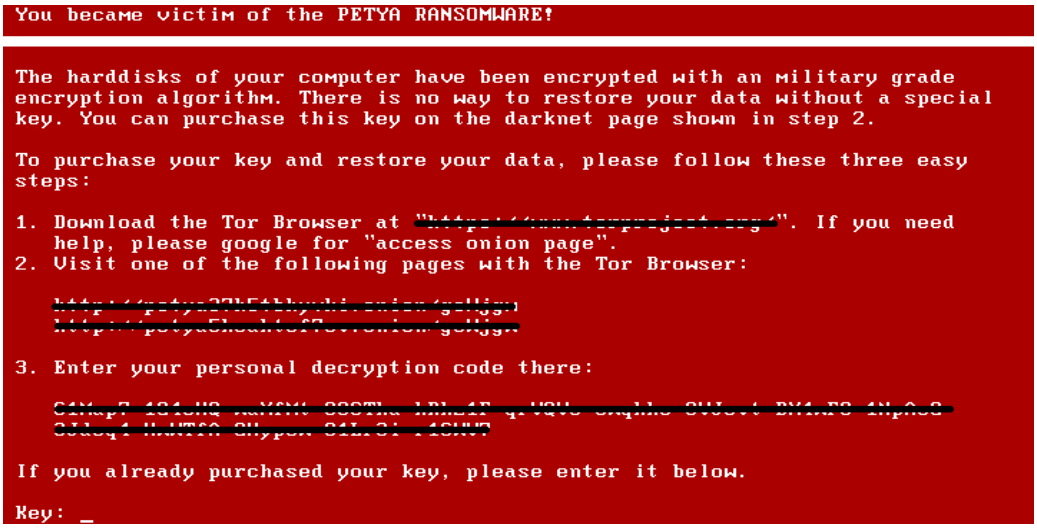

그림 1. 랜섬웨어 메시지

해당 메시지의 내용은 하드 디스크가 암호화되었으니 이를 해제하기 위한 키를 구매하라는 내용으로, 토르를 통해 개인에게 지정된 특정사이트로 접속하도록 한다. 해당 사이트에 접속하여 금전을 요구한다.해당 샘플이 실행된 이후 MBR 은 아래 그림과 같다.

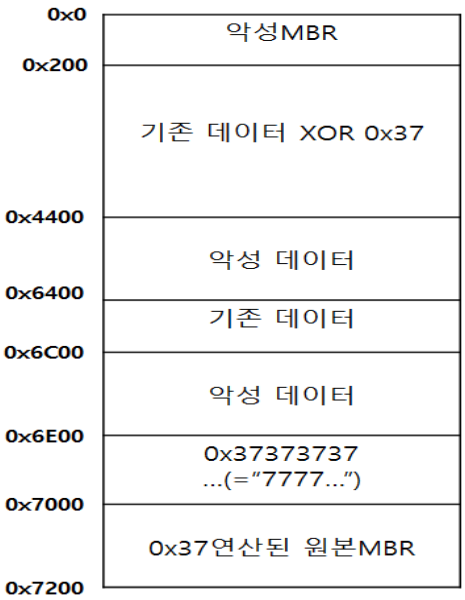

그림 2. 실행 후 MBR 의 모습

원본 MBR 의 내용이 사라지는 것이 아니라 간단하게 인코딩 되어 뒷 부분(0x7200)에 존재하고 있으므로 이는 다시 XOR 0x37 연산을 통해 원본 MBR 로 복구가 가능하다. 그리고 0x200~0x4400 사이의 데이터는 기존의 데이터에 XOR 0x37 연산을 진행한 뒤 다시 해당 위치에 기록이 된다.

PC 의 부팅이 불가능해진다는 점에서 요즘 유행하고 있는 랜섬웨어 보다 더 파괴적인 기능을 할 것이라 생각하지만, 현재 발견된 PETYA 는 MBR 에 한해 조작하며 PC 가 가진 데이터는 암호화하지 않기 때문에 MBR 을 복구하거나 하드를 다른 곳에 연결하여 기존 파일을 추출해낼 수가 있다.

다만, 이러한 MBR Locker 가 좀 더 진화하여 변형된 형태로 나타날 경우 PC 부팅부터 불가능해진 상황에 더욱 피해를 커질 수가 있다. 따라서 이러한 이후에 나올 변형을 이해하기 위해서는 원형에 대하여 상세히 이해하고 있는 것이 도움이 될 것이라 생각한다.

2 분석 내용

해당 악성코드 샘플에 대해 상세 분석한 내용들을 정리한 페이지로, 주로 OllyDBG 와 IDA Pro 를 통해 진행하였다. 어떻게 프로그램이 동작하는지 확인할 것이며, 주로 악의적인 부분에 한하여 다룰 것이다. 분석한 내용을 차례대로 살펴보자.

해당 샘플은 메모리에 MZ 헤더로 시작하는 새로운 DLL 파일을 올려 해당 부분의 명령을 수행하도록 한다. 이러한 DLL 파일의 생성과정은 아래의 그림과 같다. 0x41AE72 부터 0x41B358 까지의 반복문을 통해 0x41B35E 부터 새로운 바이너리를 기록한다. 해당 첫 부분에는 JMP 명령어가 생성되며 그 다음부터는 새로운 DLL 을 위해 MZ 헤더부터 48,128 Bytes(47 KB) 기록한다. 그리고 앞의 JMP 명령어를 통해 새로 생성된 DLL(1)의 Export "ZuWQdweafdsg345312" 부분으로 이동하게 된다.

```
.text:0041B343                ror     ecx, 1
.text:0041B346                neg     ebx
.text:0041B348                inc     esi
.text:0041B349                xor     edx, ecx
.text:0041B34B                xor     esi, 38037A4h
.text:0041B351                sub     ebx, ebp
.text:0041B353                neg     eax
.text:0041B355                sub     esi, 4
.text:0041B358                jnz     loc_41AE72
.text:0041B358 ; ---------------------------------------------------------------------------
.text:0041B35E                db 0EDh                 ; JMP 명령어 생성
.text:0041B35F                db 9Dh                  ; MZ 헤더부터 새로운 DLL 생성
.text:0041B360                db 81h
.text:0041B361                db 0C7h
```

그림 3. DLL 생성 과정

메모리에 새로 로드된 DLL 의 Export 부분에 대하여 알아보자. 우선 VirtualAlloc API 를 통해 새로운 공간을 할당하며 여기서 DLL 자신을 다시 복사한다. 그리고 아래 몇 가지 기능을 수행한 후 복사한 자신의 DllMain 함수 부분을 호출하게 된다. 이에 대해선 이후에 더 자세히 알아보자.

```
.text:100013C8                mov     esi, [ebx+3Ch]
.text:100013CB                push    40h
.text:100013CD                add     esi, ebx
.text:100013CF                push    3000h
.text:100013D4                mov     [esp+40h+var_1C], esi
.text:100013D8                push    dword ptr [esi+50h]
.text:100013DB                push    0
.text:100013DD                call    ebp             ; VirtualAlloc()
```

그림 4. 메모리 할당

메인 함수로 가기 전 새로운 함수들을 LoadLibrary 와 GetProcAddress API 를 통해 로드한다. 해당 함수들은 악성코드가 자신의 기능을 하는데 있어 핵심적으로 사용될 함수들이며, 이는 안티 바이러스 제품으로부터 API 기반 탐지에 인식되지 않도록 하기

위함이라 볼 수 있다. 로드되는 DLL 은 KERNEL32.DLL 과 ADVAPI32.DLL 두 가지이며 이에 따른 API 목록은 아래의 그림과 같다.

DLL Name	API Name	
KERNEL32.DLL	SetFilePointer	WriteFile
	ReadFile	GetSystemDirectory
	DeviceIoControl	CloseHandle
	CreateFile	GetLastError
	GetProcAddress	GetModuleHandle
	HeapAlloc	HeapFree
	GetProcessHeap	GetTickCount
	GetCurrentProcess	
ADVAPI32.DLL	LookupPrivilegeValue	OpenProcessToken
	CryptGenRandom	CryptAcquireContext
	CryptReleaseContext	AdjustTokenPrivileges

그림 5. 로드한 API 목록

이렇게 DLL 과 API 를 메모리에 새로 올린 뒤, 위에서 VirtualAlloc 를 통해 할당된 공간에 다시 기록된 DLL 의 Main() 함수를 호출하게 된다. 해당 함수에는 핵심적인 기능들이 모두 포함되어있다. 우선 아래의 그림과 같이 논리 드라이브와 물리 드라이브를 CreateFile API 를 사용하여 "₩₩.₩C:", "₩₩.₩PhysicalDrive0"를 통해 존재하고 있는지, 그리고 핸들을 가질 수 있는지 확인한다.

```
.text:10008B5D                mov     [esp+0C48h+NumberOfBytesWritten], ecx
.text:10008B61                lea     ecx, [esp+0C48h+FileName]
.text:10008B65                mov     [esp+0C48h+var_C1C], edi
.text:10008B69                call    LogicalDriveCheck ; GetSystemDirectory()
.text:10008B69                                          ; CreaeteFile("₩₩.₩C:")
.text:10008B69                                          ; DeviceIoControl
.text:10008B6E                test    eax, eax
.text:10008B70                jz      loc_10008E3A
.text:10008B76                lea     ecx, [esp+0C48h+FileName] ; lpFileName
.text:10008B7A                call    PhysicalDriveCheck ; CreateFile("₩₩.₩PhysicalDrive0")
.text:10008B7A                                          ; DeviceIoControl
```

그림 6. Drive 확인

만약 위 그림에서 실패할 경우 함수는 바로 종료 루틴으로 넘어가게 된다. 만약 함수가 종료 루틴으로 가지 않을 경우 본격적으로 MBR 과 관련된 작업을 수행한다. 우선 물리 드라이브에 대한 핸들을 얻은 다음, 0x200 Bytes 만큼 MBR 의 내용을 읽는다.

```
.text:10008CC6                push    ebx             ; int
.text:10008CC7                push    ebx             ; int
.text:10008CC8                lea     edx, [esp+0C50h+var_600] ; lpBuffer
.text:10008CCF                lea     ecx, [esp+0C50h+FileName] ; lpFileName
.text:10008CD3                call    C_R_File        ; CreateFile("₩₩.₩PhysicalDrive0")
.text:10008CD3                                        ; SetFilePointer(0)
.text:10008CD3                                        ; ReadFile(size=0x200)
.text:10008CD3                                        ; MBR의 내용을 읽음
```

그림 7. MBR 의 내용을 읽음

이렇게 읽은 MBR 의 내용은 아래 첫 번째 그림과 같이 XOR 0x37 연산을 거치게 되며, 연산된 데이터는 이후에 물리 드라이브의 시작점에서 0x7000 떨어진 지점에 기록하게 된다. MBR 을 제거하는 것이 아니라 단순한 XOR 연산을 진행한 뒤 이를 보존한다는 점에서 실제 CTBLocker 나 테슬라 랜섬웨어보다는 위험하지 않다고 볼 수 있다.

```
.text:10008D30                  xor     [esp+eax+0C48h+var_800], 37h ; 읽은 MBR XOR 0x37
.text:10008D38                  inc     eax
.text:10008D39                  cmp     eax, ebp
.text:10008D3B                  jb      short loc_10008D30 ; 읽은 MBR XOR 0x37


.text:10008EB1                  push    edi             ; int
.text:10008EB2                  push    38h             ; int
.text:10008EB4                  lea     edx, [esp+0C50h+var_800] ; lpBuffer
.text:10008EBB                  lea     ecx, [esp+0C50h+FileName] ; lpFileName
.text:10008EBF                  call    C_W_File        ; PhysicalDrive0
.text:10008EBF                                          ; SetFilePointer(0x7000)
.text:10008EBF                                          ; WriteFile(size=0x200,
.text:10008EBF                                          ; data=XOR연산된 원본MBR)
```

그림 8. MBR 데이터 XOR 연산 후 기록

0x0~0x200 의 내용은 위에서 작업이 이루어졌다면, 0x200~0x4400 의 데이터는 아래의 과정을 통해 수정된다. 우선 0x200 부터 0x200 Bytes 씩 읽어 이를 XOR 0x37 연산을 진행한 뒤, 바로 해당 자리에 연산된 값을 기록한다. 0x200 부터 0x200 Bytes 씩 읽는 작업이 0x22 번 반복되기에 0x4400 까지 이러한 연산이 진행된다.

```
.text:10008DAE                  push    eax             ; int
.text:10008DAF                  push    edi             ; int
.text:10008DB0                  lea     edx, [esp+0C50h+Buffer] ; lpBuffer
.text:10008DB7                  mov     [esp+0C50h+NumberOfBytesWritten], eax
.text:10008DBB                  call    C_R_File        ; ₩₩.₩PhysicalDrive0
.text:10008DBB                                          ; SetFilePointer(0x200~0x4400)
.text:10008DBB                                          ; Read(size=0x200)
.text:10008DC0                  pop     ecx
.text:10008DC1                  pop     ecx
.text:10008DC2                  xor     ecx, ecx
.text:10008DC4
.text:10008DC4 loc_10008DC4:                            ; CODE XREF: Mal_Func+288↓j
.text:10008DC4                  xor     [esp+ecx+0C48h+Buffer], 37h ; 읽은 0x200 Bytes를 XOR 0x37연산
.text:10008DCC                  inc     ecx
.text:10008DCD                  cmp     ecx, 200h
.text:10008DD3                  jb      short loc_10008DC4 ; 읽은 0x200 Bytes를 XOR 0x37연산
.text:10008DD5                  push    [esp+0C48h+NumberOfBytesWritten] ; int
.text:10008DD9                  lea     edx, [esp+0C4Ch+Buffer] ; lpBuffer
.text:10008DE0                  push    edi             ; int
.text:10008DE1                  lea     ecx, [esp+0C50h+FileName] ; lpFileName
.text:10008DE5                  call    C_W_File        ; CreateFile("₩₩.₩PhysicalDrive0")
.text:10008DE5                                          ; SetFilePointer("0x200~")
.text:10008DE5                                          ; WriteFile(size=0x200,
.text:10008DE5                                          ; 위에서 암호화된 내용을 해당 자리에 기록
```

그림 9. 0x200~0x4400 XOR 0x37 연산

위 두 작업을 통해 0x0~0x200 의 내용은 XOR 0x37 되어 0x7000 에 기록된 것을 확인할 수 있으며, 0x200~0x4400 까지의 내용은 XOR 연산된 후 다시 원래 자리에 기록되었다는 것을 알 수 있다. 이제 해당 샘플은 아래의 그림과 같이 자신이 가지고 있던 내용을 MBR 에 기록하므로 MBR 을 변조시킨다.

```
.text:10008DFC                  push    edi             ; int
.text:10008DFD                  push    edi             ; int
.text:10008DFE                  mov     edx, ebp        ; lpBuffer
.text:10008E00                  call    C_W_File        ; CreateFile("₩₩.₩PhysicalDrive0")
.text:10008E00                                          ; SetFilePointer("0x0")
.text:10008E00                                          ; WriteFile(size=0x200,
.text:10008E00                                          ; data = NEW악성MBR)
```

그림 10. 새로운 MBR 기록

이제 0x0~0x200 의 내용에 새로운 내용이 생성이 되었으며, 0x200~0x4400 에는 XOR 되어 있는 데이터가 존재하고 있게 된다. 그 다음 아래의 그림과 같이 물리 드라이브에서 포인터를 0x4400 으로 지정한 후 0x200 Bytes 의 데이터를 기록하는 것을 확인할 수 있다.

```
.text:10008E44                  push    edi             ; dwMoveMethod
.text:10008E45                  push    edi             ; lpDistanceToMoveHigh
.text:10008E46                  push    4400h           ; lDistanceToMove
.text:10008E4B                  push    esi             ; hFile
.text:10008E4C                  call    ds:SetFilePointer ; seek(0x4400)
.text:10008E52                  push    edi             ; lpOverlapped
.text:10008E53                  lea     eax, [esp+0C4Ch+NumberOfBytesWritten]
.text:10008E57                  push    eax             ; lpNumberOfBytesWritten
.text:10008E58                  push    ebx             ; nNumberOfBytesToWrite
.text:10008E59                  lea     eax, [ebp+200h]
.text:10008E5F                  push    eax             ; lpBuffer
.text:10008E60                  push    esi             ; hFile
.text:10008E61                  call    ds:WriteFile    ; 0x4400~0x6400
.text:10008E61                                          ; size=0x2000
.text:10008E61                                          ; data=악성데이터
```

그림 11. 0x4400~0x6400 새로운 데이터 기록

이후 아래 첫 번째 그림과 0x6C00 으로 포인터를 설정하고 다시 0x200 만큼 새로운 데이터를 기록한다. 이렇게 기록된 내용 뒤에는 두 번째 그림과 같이 "₩x37"로 0x200 Bytes 만큼 기록이 되며, 이는 무의미한 값을 가진다.

```
.text:10008E7F                  push    edi             ; int
.text:10008E80                  push    36h             ; int
.text:10008E82                  lea     edx, [esp+0C50h+var_400] ; lpBuffer
.text:10008E89                  lea     ecx, [esp+0C50h+FileName] ; lpFileName
.text:10008E8D                  call    C_W_File        ; PhysicalDrive0
.text:10008E8D                                          ; SetFilePointer(0x6C00)
.text:10008E8D                                          ; WriteFile(size=0x200,
.text:10008E8D                                          ; data=악성데이터)
```

```
.text:10008E98                push    edi             ; int
.text:10008E99                push    37h             ; int
.text:10008E9B                lea     edx, [esp+0C50h+var_200] ; lpBuffer
.text:10008EA2                lea     ecx, [esp+0C50h+FileName] ; lpFileName
.text:10008EA6                call    C_W_File        ; PhysicalDrive0
.text:10008EA6                                        ; SetFilePointer(0x6E00)
.text:10008EA6                                        ; WriteFile(size=0x200,
.text:10008EA6                                        ; data="0x37373737...")
```

그림 12. 0x6C00~0x7000 새로운 데이터 기록

이렇게 0x0000 부터 0x7200 까지 데이터들의 변화가 일어났다. 이렇게 변조된 MBR 을 위해 PC 가 강제로 재부팅되어야 하는데, 필자가 이전에 분석했던 .micro 확장자는 "shutdown" 명령어를 통해 재부팅시킨 것에 반해 해당 샘플은 권한을 상승시킨 뒤 NTDLL.DLL 의 "RaiseHardError"의 주소를 얻고 이를 호출하므로 PC 가 재부팅되도록 한다.

```
.text:10008ECE                call    Reboot_Function ; OpenProcessToken
.text:10008ECE                                        ; LookupPrivilegeValue("SeShutdownPrivilege")
.text:10008ECE                                        ; AdjustTokenPrivileges
.text:10008ECE                                        ; GetModuleHandleA("NTDLL.DLL")
.text:10008ECE                                        ; GetProcessAddress("NtRaiseHardError")
.text:10008ECE                                        ; CALL NtRaiseHardError
```

그림 13. PC 강제 재부팅

이후 시스템은 자동으로 재부팅을 진행하게 되며 이러한 과정에서 MBR 이 손상되었기 때문에 CHKDSK 검사 단계를 진행하게 된다. 그리고 CHKDSK 검사 단계가 100%로 모두 완료된 후에는 PETYA 에 의해 새로 작성된 MBR 이 실행되어 랜섬웨어 메시지를 출력하게 된다.