**CVE-2019-8044 – Adobe Reader DC에서 double-free 취약점**

**고려대학교 박성하**

1. **서론 : 취약점 설명**

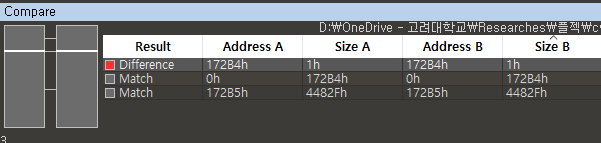
GOOGLE SECURITY RESEARCH TEAM이 2019년 8월 공개한 취약점으로 ,이 취약점을 통해, 최신 버전의 Acrobat Reader DC에서 Double free 취약점으로 Crash를 유도할 수 있다. Exploit-DB에 올라온 설명을 발췌한 내용을 보면, 아래와 같다.

|  |
| --- |
| VERIFIER STOP 00000007: pid 0x2C1C: Heap block already freed.  0C441000 : Heap handle for the heap owning the block.  147E6638 : Heap block being freed again.  00000010 : Size of the heap block.  00000000 : Not used |

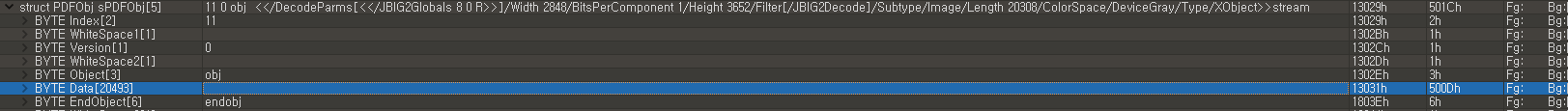
즉, 이미 free된 heap을 다시 호출하여 취약점이 발생함을 알 수 있으며, 0x147E6638에서 다시 heap을 하여 취약점이 발생함을 볼 수 있다. 여기서는 공개된 정보와 PoC를 기반으로 취약점을 분석한 후, 재구성하는 과정을 담는다.

1. **취약점 기초 분석**

먼저, GOOGLE SECURITY RESEARCH TEAM에서 공개한 PoC를 보면, 아래와 같이 원본과 PoC 파일을 얻을 수 있다. 두 파일을 diffing하여 최초 분석을 시작한다.



해당 PoC 파일에서 0x172b부분의 1바이트를 통해 취약점이 발생하였음을 알 수 있다.

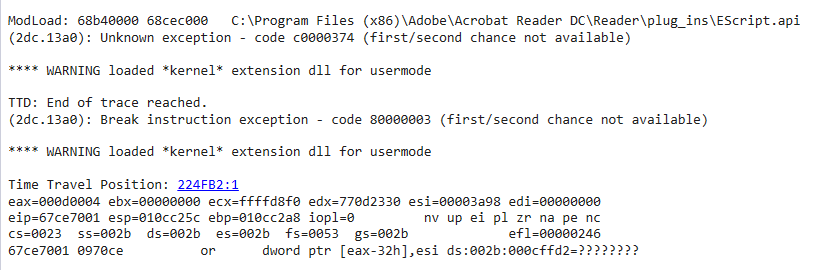


Pdf 구조를 반영하여 보면, 해당 부분은 특정한 이미지 오브젝트를 디코딩하는 과정에서 취약점이 발생하였다. 이제, Root Cause Analysis를 통해 이 바이트가 취약점을 어떻게 trigger하는지 파악하도록 한다. 이후, 새로운 PDF파일로 이 취약점을 다시 trigger해본다.

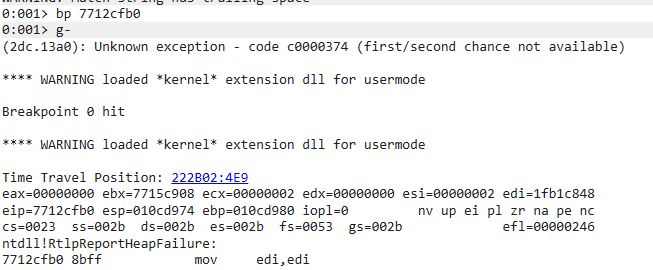
1. **Root Cause Analysis**

취약점에 대한 Root Cause Analysis는 기본에 취약점에 대해 구체적으로 분석해둔 블로그[[1]](#footnote-1)를 참조하여 진행하였다. Windows의 Time-Travel-Debugging을 통해 Acrobat Reader에서 크래시가 터지는 흐름(TTD)을 재현한 후, 재현하여 기록한 흐름을 WinDBG Preview로 분석할 수 있다. (여기서는 블로그에서 공개한 TTD를 사용하였음)

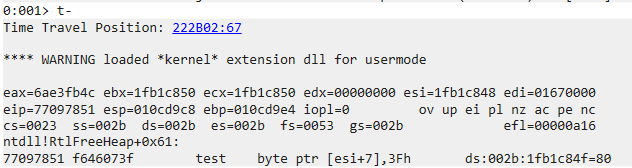
TTD를 열고, 레코드한 포인트의 끝으로 이동하면 아래와 같이 크래시 포인트를 알 수 있다.



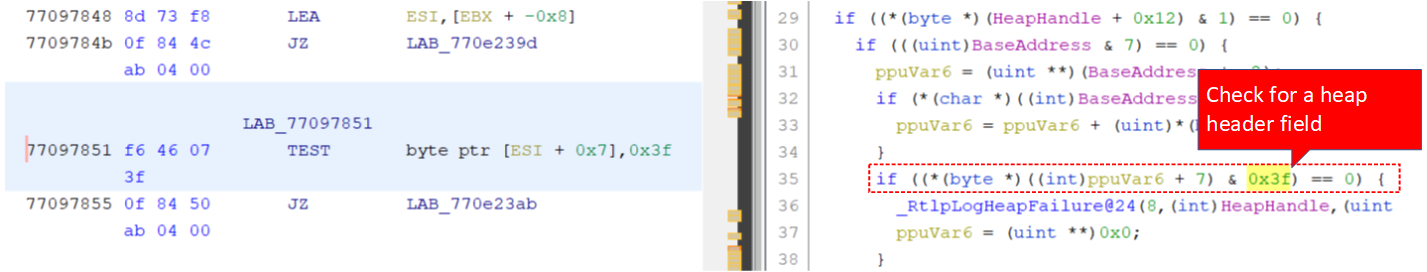
콜스택을 확인하면, MSVCR120!free가 ntdll!RtlpLogHeapFailure 함수를 부르는 것을 확인할 수 있다. 이는 힙 손상이 발생했으며, 메모리 관리자가 메모리 주소를 free하려고 할 때, 힙 관리자가 이를 식별했음을 나타낸다. 아래와 같이 Ntdll!RtlpLogHeapFailure에 breakpoint를 걸고, 뒤로 이동하여 해당 지점에 도달할 수 있다.



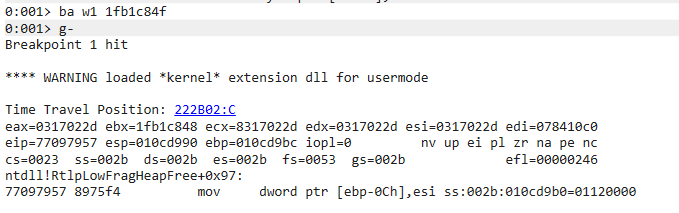
거꾸로 하나씩 실행시키면서 어떤 조건에서 힙 check가 실패했는지를 확인할 수 있다.



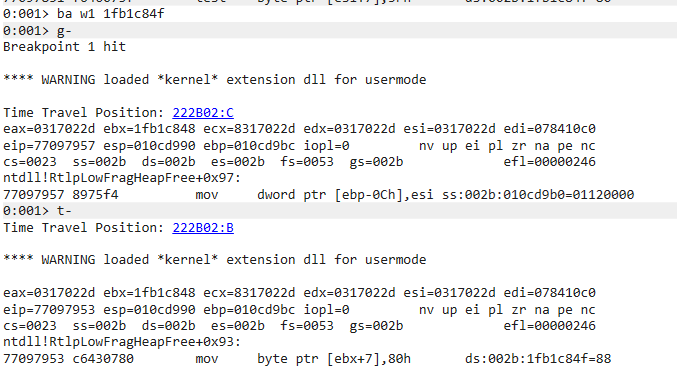
이 부분을 디스어셈블하면, 아래와 같이 확인할 수 있다. (디스어셈블 부분은 블로그에서 발취)



0x1fb1c84f의 바이트가 손상되어있으므로, 여기에 write한 시점의 명령어를 식별한다.



아래 코드는 0x1fb1c84f의 바이트가 수정되는 위치를 보여준다.

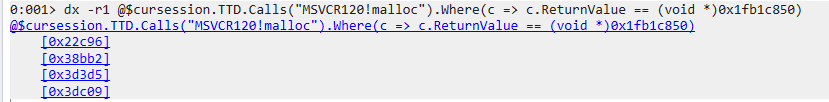


이 당시의 콜스택을 보면, 아래와 같다. 콜스택의 02번을 보면, 0x1fb1c850에서 msvcr120!free 호출로 인해 발생한다. 이로 인해 0xfb1c850의 더블 프리 취약점임을 알 수 있다.

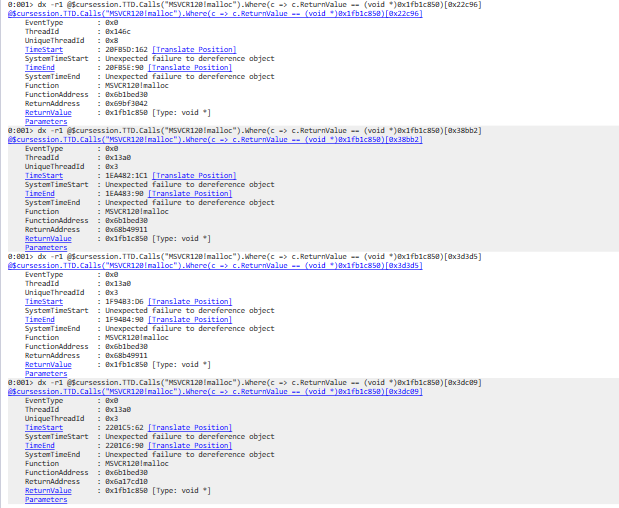


0x1fb1c850에서의 free는 두번 수행되었다. 이제, 첫 번째 free 이후 이 메모리에서 새로운 할당이 있었는지 확인하여야 한다. 한 가지 방법은 TTD object에서 LINQ 쿼리를 이용하는 것이다.

아래의 결과는 0x1fb1c850에 의해 리턴된 MSVCR120!malloc에 있는 모든 인스턴스들을 나타낸다. 네 개의 인스턴스를 확인할 수 있다.

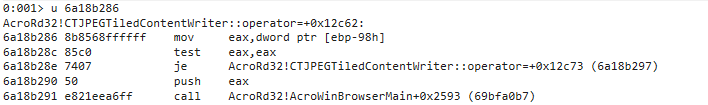


각각의 call에 대해 조사하면, 0x14e29가 마지막 메모리 할당이며, 다른 두 free보다 이전에 실행됨을 알 수 있다. 따라서 이 취약점은 더블 프리에 의해 야기되는 것이다.

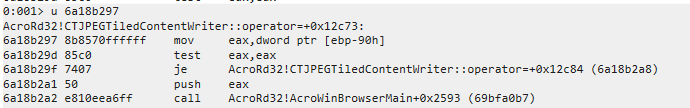


다음으로, free된 메모리를 추적한다. Double-free가 발생하는 코드는 다음과 같다.

* 첫 free



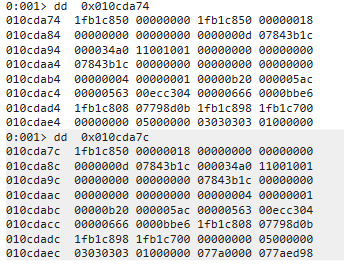
* 두번째 free



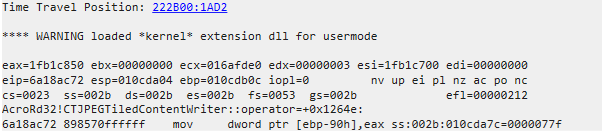
위 코드를 디스어셈블하면, 아래와 같다.

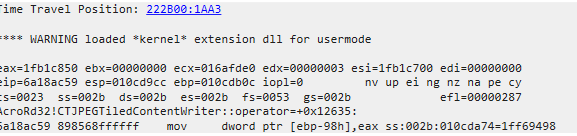
|  |
| --- |
| void FreeJP2Resources(void)  {  ...  if (\*(int \*)(unaff\_EBP + -0x98) != 0) {  free(\*(int \*)(unaff\_EBP + -0x98));  }  if (\*(int \*)(unaff\_EBP + -0x90) != 0) {  free(\*(int \*)(unaff\_EBP + -0x90));  } |

위의 ebp-98h(0x010cda74)와 ebp-90h(0x010cda7c)는 같은 포인터인 0x1fb1c850d을 나타내고 있다.



두 개의 메모리 주소가 할당된 주소로 다시 추적하면, 아래와 같은 두 위치를 얻을 수 있다.

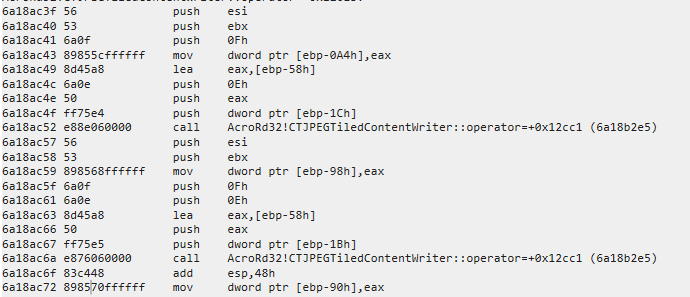
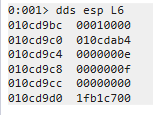




위 두 개의 메모리 주소는 0x6a18b2e5(GetMemoryBlock)의 함수에서 찾을 수 있다.

|  |
| --- |
| undefined4  GetMemoryBlock(undefined4 memory\_block\_type,int \*param\_2,byte offsetVal1,byte offsetVal2,  undefined4 offsetVal3,int param\_6)    {  undefined4 retVal;  int memory\_block\_base;  byte offset;  code \*pcVar1;    if ((char)memory\_block\_type == '\x03') {  if ((param\_2 == (int \*)0x0) || (param\_6 == 0)) {  exception:  CallThrowException(0x40000003,0);  memory\_block\_type = 0;  \_CxxThrowException(&memory\_block\_type,0x7472e75c);  pcVar1 = (code \*)swi(3);  retVal = (\*pcVar1)();  return retVal;  }  \*param\_2 = \*param\_2 + 1;  retVal = RetrieveMemoryBlock(param\_6,\*param\_2 + -1);  }  else {  offset = offsetVal1;  if (((char)memory\_block\_type != '\0') &&  (offset = offsetVal2, (char)memory\_block\_type != '\x01')) {  if ((char)memory\_block\_type != '\x02') goto exception;  offset = (byte)offsetVal3;  }  if (offset == 0) goto exception;  memory\_block\_base = (\*\_TlsGetValueStub)(\_dwTlsIndexForMemoryBlockBase);  retVal = \*(undefined4 \*)(memory\_block\_base + 0x20 + (uint)offset \* 4);  }  return retVal;  } |

위 함수를 살펴보면, 전반적으로 memory\_block\_type 변수에 따라 서로 다른 오프셋과 memory\_block\_base 주소를 사용하게 된다.



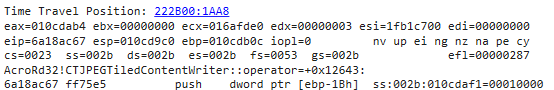


첫 번째 호출 시 memory\_block\_type으로 0x01000000을 전다하고, 두 번째 호출은 0x00010000을 전달한다. 그러나 GetMemoryBlock을 호출할 때 memory\_block\_type은 char로 불러들이기에 두 변수는 모두 0x00으로 해석된다.

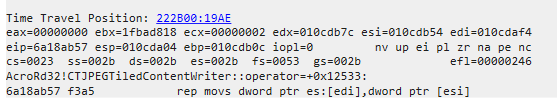
|  |
| --- |
| memory\_block\_base = (\*\_TlsGetValueStub)(\_dwTlsIndexForMemoryBlockBase);  retVal = \*(undefined4 \*)(memory\_block\_base + 0x20 + (uint)offset \* 4); |

두 호출 모두 위 코드에서 메모리 블록을 검색하고, 다른 매개변수는 동일하기 때문에, 같은 memory\_block\_type을 사용하면 동일한 메모리 주소를 다른 필드에 할당하게 되어 이중 프리가 발생하게 된다.

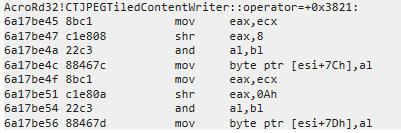
위에서 두 번째 호출인 0x6a18ac6a의 memory\_block\_type은 0x6A18AC67에서 푸쉬되었다.



아래와 같이, 이 메모리값의 출처를 확인할 수 있다. 여기서 0x010cdaf1은 [ebp-1Bh] 메모리의 위치이다.



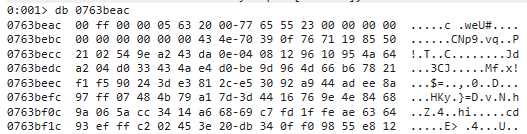
esi레지스터의 0x010cdb54의 4바이트는 여기서 edi레지스터인 0x010cdaf4로 복사된다. ba명령을 반복하면, CalcMemoryBlockType 내에서 memory\_block\_type값이 계산되는 코드 블록을 식별할 수 있다. 아래 0x6a17be56에서 al 레지스터는 memory\_block\_type값을 보유한다.



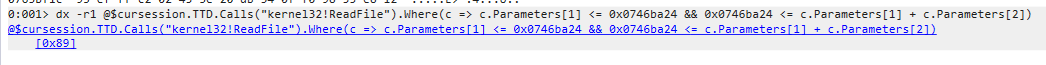
0x6a17be1d의 레지스터 ax의 값은 자체 수식을 사용하여 바이트를 정수로 변환하는 ReadInt 함수 내의 다음 코드에서 읽어진다.

|  |
| --- |
| ret\_val = (uint)\*\*buffer;  currente\_byte = \*buffer + 1;  \*buffer = currente\_byte;  if (1 < size) {  iVar2 = size - 1;  do {  ret\_val = ret\_val \* 0x100 + (uint)\*currente\_byte;  currente\_byte = currente\_byte + 1;  \*buffer = currente\_byte;  iVar2 = iVar2 + -1;  } while (iVar2 != 0); |

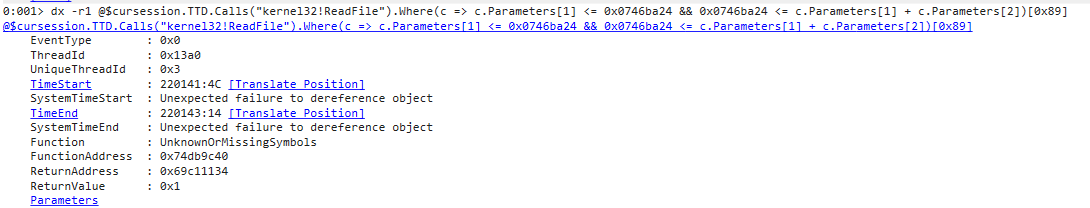
정수로 변환된 바이트는 0x0763beac에서 2바이트이다.



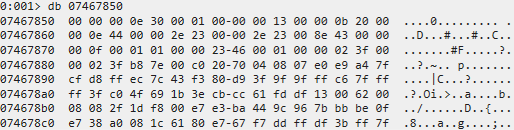
ba명령을 통해 0x0746ba24에서 복사되었음을 확인할 수 있지만, 이는 여기서 기록된 TTD에서 추적할 수 없으므로 0x0746ba24의 메모리 내용이 커널 함수에 복사되어있다고 가정한다. 이를 테스트하기 위해 아래 TTD 쿼리를 싱행하여 대상 메모리인 0x0746ba24에서 ReadFile호출을 찾는다. C.Parameter[1]은 버퍼 주소를 보유하고, c.Parameters[2]는 버퍼 크기값을 갖는다.



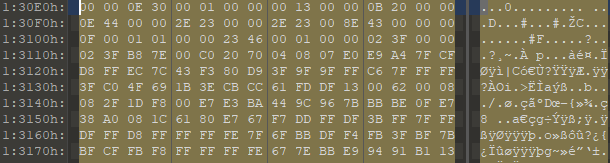
이는 위와 같이 인스턴스를 하나만 반환했으며, 추가 조사를 통해 PDF파일에서 내용을 읽는 ReadFile 함수임을 확인할 수 있다.



이 ReadFile 호출 후 대상 메모리 버퍼는 아래와 같다.



파일 오프셋 0x130DF에서 내용을 읽었음을 확인할 수 있고, 파일에서 확인하면, 다음과 같이 동일한 내용을 볼 수 있다.



최종적으로, 0x0746ba24의 메모리 바이트는 아래와 같이 0x172b3의 위치에서 복사되었으며, 이는 메모리 할당에 직접적인 영향을 줌을 확인할 수 있다.

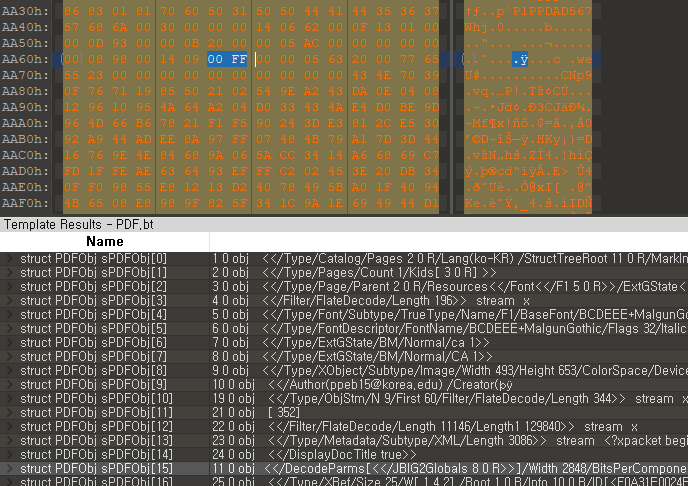


원래 바이트는 아래와 같이 00 1C이며, ReadInt 호출에 의해 정수 1D로 변환된다 1D값은 CalcMemoryBlockType 함수에서 3으로 마스킹되어 memory\_block\_type은 00 ff일 때 0으로 세팅되는 것과 달리, 1로 세팅되게 된다. 즉, GetMemoryBlock의 중복된 메모리 블록은 double free 문제의 root cause가 된다.



1. **취약점 Re-trigger**

사실, 이 취약점만으로 크래시 이외에 다른 행위를 유발할 수는 없기 때문에, 바로 활용하기에는 어려울 수 있지만, 이 취약점을 활용하여 다른 내용의 PDF를 채울 수 있는 PoC를 작성해볼 수 있다. 가장 간단한 방법은 여기서 취약점 trigger에 사용된 이미지 Object를 그대로 활용한 후, pdf에 다른 Object를 넣거나, 임의의 pdf를 생성한 후, 여기서 활용한 이미지 Object를 삽입하는 것이다. 아래와 같이, 새로운 pdf를 만든 후, 취약점을 trigger하는 PoC의 Object를 삽입하여 활용 가능하다.



1. **Reference**

- <https://www.exploit-db.com/exploits/47279>

- <https://github.com/offensive-security/exploitdb-bin-sploits/raw/master/bin-sploits/47279.zip>

- https://darungrim.com/research/2019-10-10-vulnerability-root-cause-analysis-with-time-travel-debugging.html

1. https://darungrim.com/research/2019-10-10-vulnerability-root-cause-analysis-with-time-travel-debugging.html [↑](#footnote-ref-1)