CVE-2023-40031 분석 보고서

투모로우(to-more-row) / 공개된 1-day 취약점 상세 분석팀

01. 요약

1) CVE-2023-40031 개요

CVE-2023-40031 취약점은 Notepad++ 프로그램에서 UTF-16 파일을 UTF-8로 변환하는 과정에서 발생하는 힙 버퍼 오버플로우 취약점으로, 영향받는 notepad++의 버전은 8.5.6 이하 버전이다.

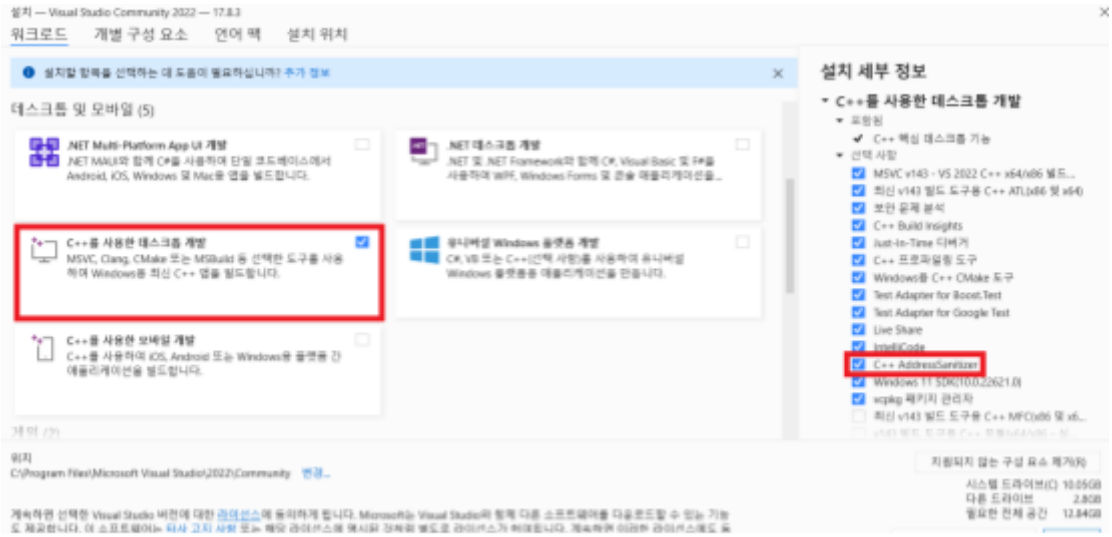
02. 사전지식

1) 재현 환경 구성

1. Notepad++ 8.5.2 소스코드 다운로드

Notepad++는 무료 오픈소스 응용 프로그램으로, 소스코드를 다운로드하여 사용할 수 있다.

2. Visual Studio 설치

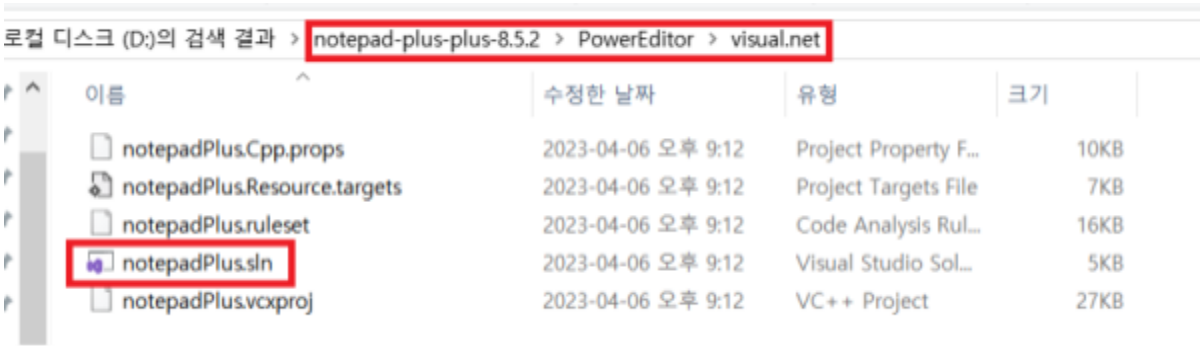


[그림 1] 취약점 환경 구성 - 1

Address Sanitizer(ASAN) 사용을 위해 다음 항목에 체크하여 Visual Studio를 설치한다.

- C++ 개발
- C++ AddressSanitizer

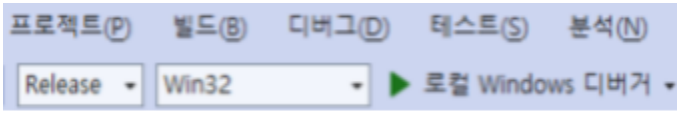
3. 솔루션 파일 열기



[그림 2] 취약점 환경 구성 - 2

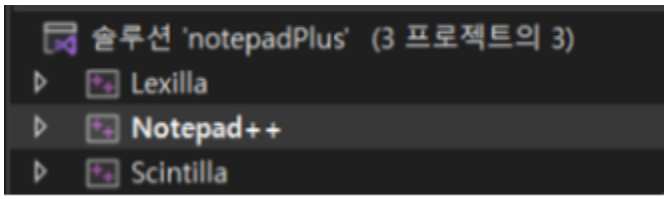
소스코드 압축을 해제하고 Visual Studio로 솔루션 파일 notepadPlus.sln을 열어준다.

4. 빌드

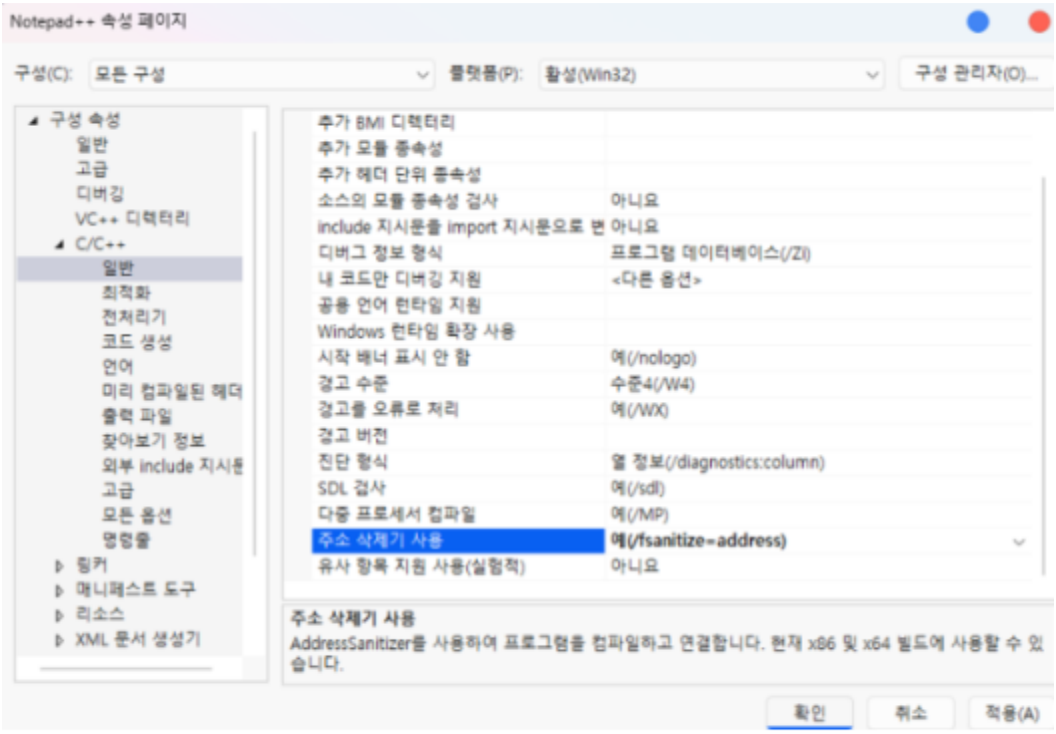


[그림 3] 취약점 환경 구성 - 3

빌드 방식은 다음과 같이 Release, Win32로 설정한다.

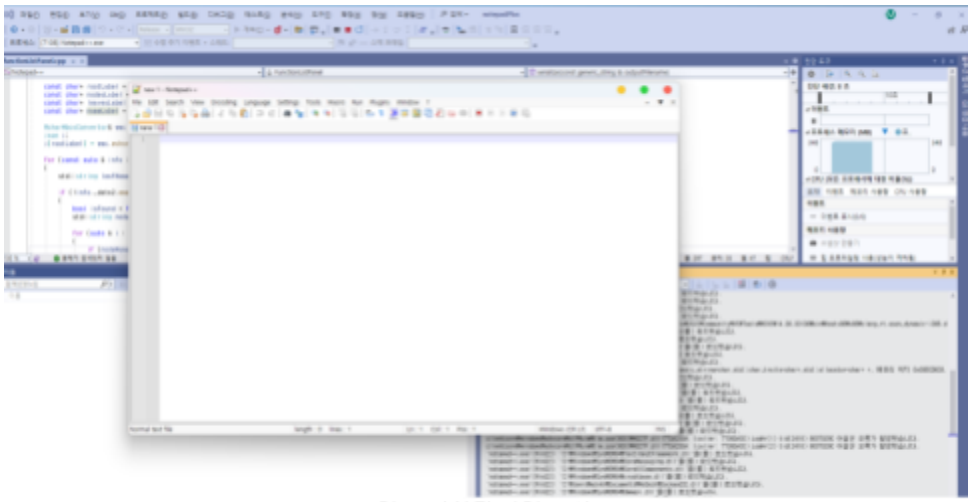


[그림 4] 취약점 환경 구성 - 4



[그림 5] 취약점 환경 구성 - 5

프로젝트 3개(Lexilla, Notepad++, Scintilla)의 [주소 삭제기 사용]을 [예(/fsintize=address)]로 변경하여 ASAN을 적용한다.



[그림 6] 취약점 환경 구성 - 6

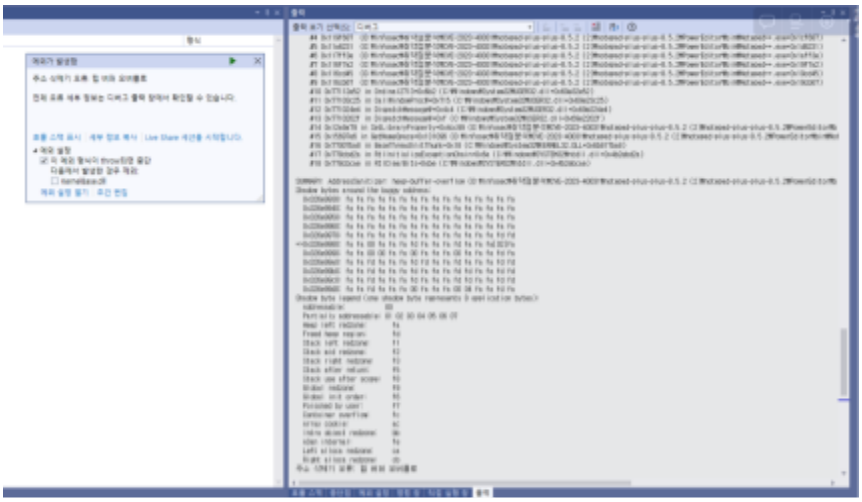
빌드하면 Notepad++를 실행할 수 있다.

5. 취약점 환경 구성 구성

```
1  with open("poc", "wb") as f:
2      f.write(b'\xfe\xff')
3      f.write(b'\xff' * (128 * 1024 + 4 - 2 + 1))
```

[그림 7] 취약점 환경 구성 - 7

파이썬 스크립트로 다음과 같은 poc.py 파일을 작성한다.



[그림 8] 취약점 환경 구성 - 8

poc.py로 생성한 파일을 Notepad++에 넣으면 힙 버퍼 오버플로우가 발생하는 것을 확인할 수 있다.

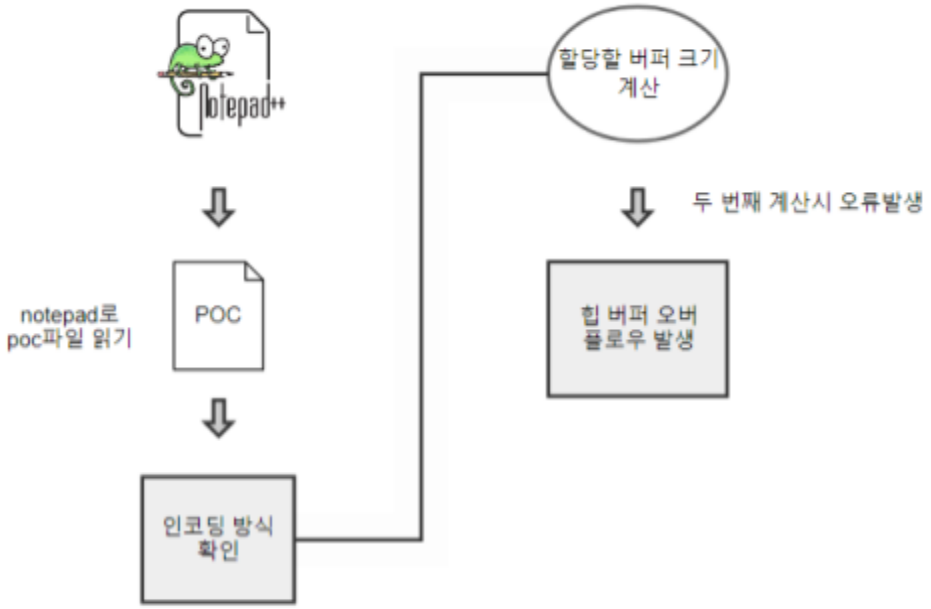
2) 디버깅 도구

Address Sanitizer(ASAN) : visual studio의 C/C++ 컴파일러의 옵션이며, 메모리에 관련하여 찾기 힘든 버그들을 탐지해주는 메모리 관련 버그 탐지 도구로, ASAN이 탐지해주는 항목들은 다음과 같다.

- Alloc/dealloc mismatches and new/delete type mismatches
- Allocations too large for the heap
- calloc overflow and alloca overflow
- Double free and use after free
- Global variable overflow
- Heap buffer overflow
- Invalid alignment of aligned values
- memcpy and strncat parameter overlap
- Stack buffer overflow and underflow
- Stack use after return and use after scope
- Memory use after it’s poisoned

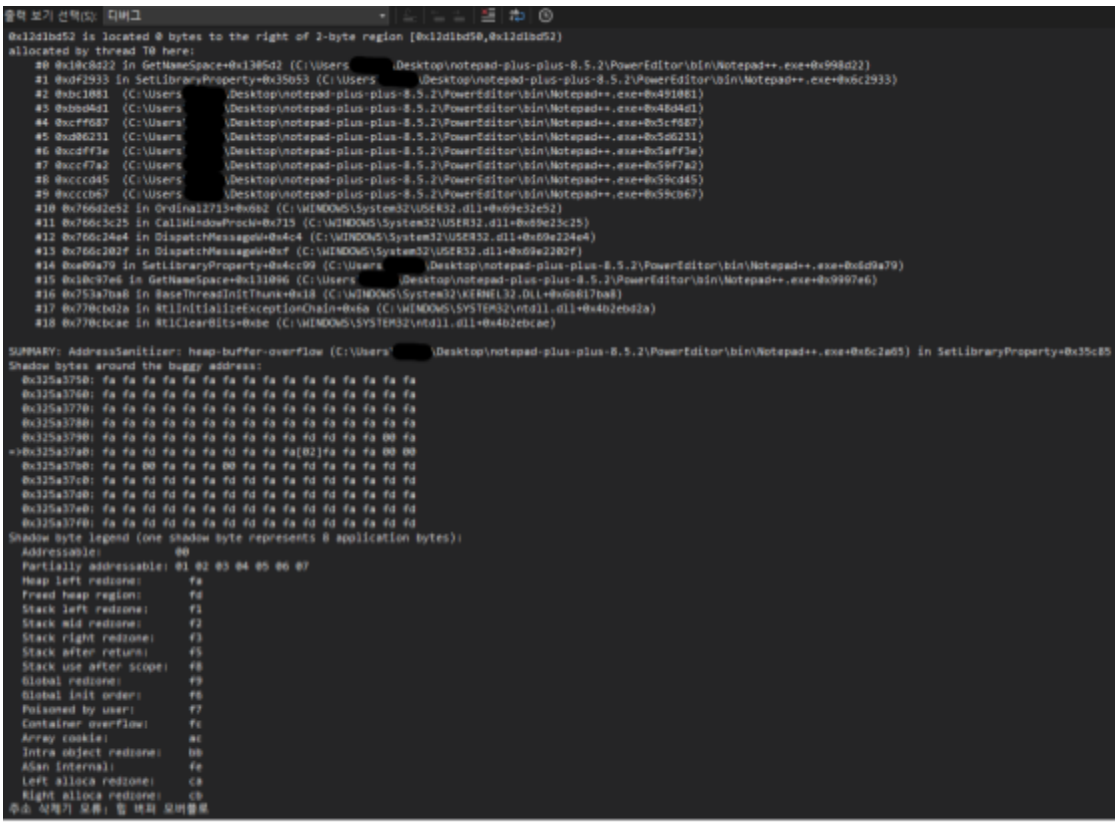
03.. 상세 분석

a) 취약점 발생 원리



[그림 9] CVE-2023-40031 취약점 발생 과정

Notepad++는 문서 편집기이자 소스코드 편집기이다. Notepad++는 처음 파일을 읽을 때, 인코딩 방식을 확인하고 UTF-8이 아니라면 UTF-8로 변환한 후 사용자에게 보여주는데, 변환 과정에서 버퍼의 크기를 잘못 할당하면서 힙 버퍼 오버플로우가 발생할 수 있다. 위의 [그림 9]는 CVE-2023-40031 취약점이 발생하는 과정을 정리한 그림이다.



[그림 10] CVE-2023-40031 ASAN 감지 결과

위의 사진은 CVE-2023-40031을 재현했을 때 ASAN에서 감지하는 내용이다. 0x12d1bd52에서 1바이트 크기의 쓰기 작업이 수행되려고 할 때, 해당 주소에서 힙 버퍼 오버플로우가 발생하고 있음을 확인할 수 있다.

b) 읽은 파일의 인코딩 방식 확인

```
// Buffer.cpp
// FileManager::loadFileData()

if (isFirstTime) // 파일을 처음 읽는 경우라면
{
  NppGUI& nppGui = NppParameters::getInstance().getNppGUI();

  // check if file contain any BOM
  if (Utf8_16_Read::determineEncoding((unsigned char *)data, lenFile) != uni8Bit) // determineEncoding 함수를 호출하여 인코딩 방식 확인
  {
    // if file contains any BOM, then encoding will be erased,
    // and the document will be interpreted as UTF
    fileFormat._encoding = -1; // 확인한 결과 uni8Bit(UTF-8)가 아니라면 fileFormat._encoding에 -1 대입
  }
...
if (fileFormat._encoding != -1)
{
  ...
}
else
{
  lenConvert = unicodeConvertor->convert(data, lenFile); // 읽은 파일 데이터를 UTF-8로 변환
  _pscratchTilla->execute(SCI_APPENDTEXT, lenConvert, reinterpret_cast<LPARAM>(unicodeConvertor->getNewBuf()));
  if (format == EolType::unknown)
    format = getEOLFormatForm(unicodeConvertor->getNewBuf(), unicodeConvertor->getNewSize(), EolType::unknown);
}
```

[그림 11] FileManger::loadFileData - 1

Buffer.cpp의 FileManager::loadFileData 함수에서 Notepad++는 파일을 처음 읽을 때 인코딩 방식을 확인한다. 이때 인코딩 방식이 uni8Bit, 즉, UTF-8이 아니라면 fileFormat._encoding을 -1로 설정한다. 그리고 해당 값이 -1이라면 convert 함수를 호출하여 파일 데이터를 UTF-8로 변환하는 것을 확인할 수 있다. 즉, Notepad++는 읽어 들인 파일 데이터가 UTF-8로 인코딩되어 있지 않다면 무조건 UTF-8로 변환한다.

```
// Utf8_16.cpp

size_t Utf8_16_Read::convert(char* buf, size_t len)
{
  // bugfix by Jens Lorenz
  static  size_t nSkip = 0; // 스킵할 바이트 수

  m_pBuf = (ubyte*)buf;      // 변환할 버퍼
  m_nLen = len;              // 변환할 데이터의 길이
  m_nNewBufSize = 0;         // 변환한 결과 버퍼 크기

  if (m_bFirstRead == true) // 파일을 처음 읽는 경우라면
  {
    determineEncoding();     // 인코딩 방식 결정 -> UTF-16BE
    nSkip = m_nSkip;         // UTF-16BE의 경우 BOM 2바이트를 포함하므로 m_nSkip = 2
    m_bFirstRead = false;
  }
```

[그림 12] Utf8_16_Read::convert - 1

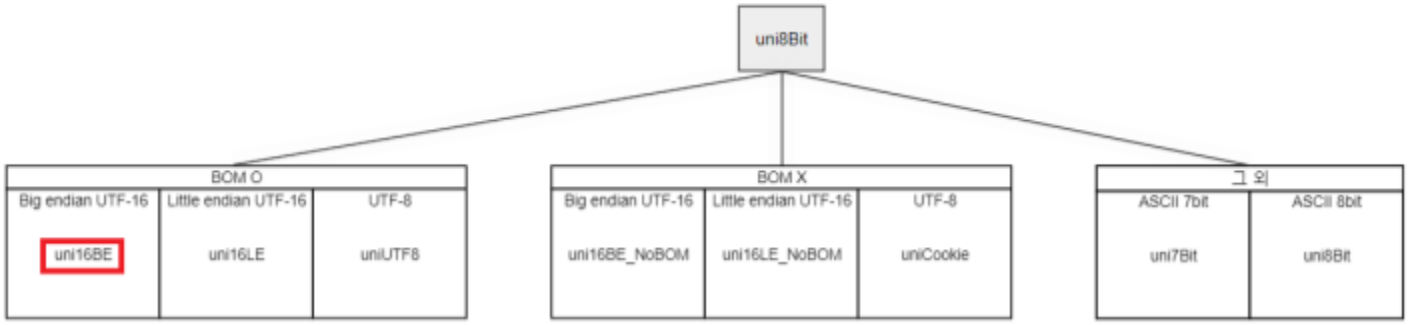
Utf8_16_Read::convert 함수에서 파일 데이터를 UTF-8로 변환하는 과정을 살펴보았다. 해당 함수에서는 파일을 처음 읽는 경우, determineEncoding 함수를 호출한다.

```
void Utf8_16_Read::determineEncoding()
{
    INT uniTest = IS_TEXT_UNICODE_STATISTICS;
    m_eEncoding = uni8Bit;
    m_nSkip = 0;

    // detect UTF-16 big-endian with BOM
    if (m_nLen > 1 && m_pBuf[0] == k_Boms[uni16BE][0] && m_pBuf[1] == k_Boms[uni16BE][1])
    {
        m_eEncoding = uni16BE;
        m_nSkip = 2;
    }
    // detect UTF-16 little-endian with BOM
    else if (m_nLen > 1 && m_pBuf[0] == k_Boms[uni16LE][0] && m_pBuf[1] == k_Boms[uni16LE][1])
    {
        m_eEncoding = uni16LE;
        m_nSkip = 2;
    }
    // detect UTF-8 with BOM
    else if (m_nLen > 2 && m_pBuf[0] == k_Boms[uniUTF8][0] &&
        m_pBuf[1] == k_Boms[uniUTF8][1] && m_pBuf[2] == k_Boms[uniUTF8][2])
    {
        m_eEncoding = uniUTF8;
        m_nSkip = 3;
    }
    // try to detect UTF-16 little-endian without BOM
    else if (m_nLen > 1 && m_nLen % 2 == 0 && m_pBuf[0] != 0 && m_pBuf[1] == 0 && IsTextUnicode(m_pBuf, static_cast<int32_t>(m_nLen), &uniTest))
    {
        m_eEncoding = uni16LE_NoBOM;
        m_nSkip = 0;
    }
    /* UTF-16 big-endian without BOM detection is taken away scince this detection is very week
    // try to detect UTF-16 big-endian without BOM
    else if (m_nLen > 1 && m_pBuf[0] == NULL && m_pBuf[1] != NULL)
    {
        m_eEncoding = uni16BE_NoBOM;
        m_nSkip = 0;
    }
    */
    else
    {
        u78 detectedEncoding = utf8_7bits_8bits();
        if (detectedEncoding == utf8NoBOM)
            m_eEncoding = uniCookie;
        else if (detectedEncoding == ascii7bits)
            m_eEncoding = uni7Bit;
        else //(detectedEncoding == ascii8bits)
            m_eEncoding = uni8Bit;
        m_nSkip = 0;
    }
}
```

[그림 13] determineEncoding

determineEncoding 함수는 인코딩을 감지하고, BOM을 기준으로 인코딩 형식인 m_eEncoding 변수 값을 결정한다.



[그림 14] m_eEncoding 값 결정

해당 함수로 인해 설정되는 m_eEncoding의 값은 다음과 같다.

```
switch (m_eEncoding)    // 인코딩 방식에 따라 버퍼를 변환할 지 변환하지 않을 지 결정
{
  case uni7Bit:
  case uni8Bit:
  case uniCookie: {
  // Do nothing, pass through
    m_nAllocatedBufSize = 0;
    m_pNewBuf = m_pBuf;
    m_nNewBufSize = len;
    break;
  }
  case uniUTF8: {
    // Pass through after BOM
    m_nAllocatedBufSize = 0;
    m_pNewBuf = m_pBuf + nSkip;
    m_nNewBufSize = len - nSkip;
    break;
  }
```

[그림 15] Utf8_16_Read::convert - 2

Utf8_16_Read::convert 함수에서는 determineEncoding 함수로 인해 결정된 m_eEncoding 값에 따라 다른 case를 실행한다.

```
case uni16BE_NoBOM:    // BOM이 없는 빅 엔디안 UTF-16
case uni16LE_NoBOM:    // BOM이 없는 리틀 엔디안 UTF-16
case uni16BE:          // BOM이 있는 빅 엔디안 UTF-16
case uni16LE: {        // BOM이 있는 리틀 엔디안 UTF-16
      size_t newSize = len + len / 2 + 1;

  if (m_nAllocatedBufSize != newSize)
      {
    if (m_pNewBuf)
      delete [] m_pNewBuf;
          m_pNewBuf  = NULL;
          m_pNewBuf  = new ubyte[newSize];
    m_nAllocatedBufSize = newSize;
      }

      ubyte* pCur = m_pNewBuf;

      m_Iter16.set(m_pBuf + nSkip, len - nSkip, m_eEncoding);
```

[그림 16] Utf8_16_Read::convert - 3

그 중 UTF-16으로 인코딩되었을 경우, 새로운 버퍼의 크기를 계산하는 것을 확인할 수 있다.

[그림 17] Utf8_16_Read::convert - 4

그리고 반복문을 통해 UTF-16 문자를 2바이트씩 읽어 UTF-8 문자로 변환하는 것을 확인할 수 있다. 여기서 현재 포인터와 다음 포인터가 데이터 끝을 넘어서지 않는지 확인하는 과정이 없어 버퍼 오버플로우가 일어날 가능성이 존재한다.

c) UTF-8과 UTF-16 비교

Code point ↔ UTF-8 conversion

First code point	Last code point	Byte 1	Byte 2	Byte 3	Byte 4
U+0000	U+007F	0xxxxxxx			
U+0080	U+07FF	110xxxxx	10xxxxxx		
U+0800	U+FFFF	1110xxxx	10xxxxxx	10xxxxxx	
U+10000	[b]U+10FFFF	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx

[그림 18] Code point <-> UTF-8 conversion

UTF-16는 2바이트 단위로 문자를 표현하지만, UTF-8은 가변 길이 인코딩 방식을 사용한다. UTF-8의 경우, 유니코드의 범위에 따라 바이트 크기가 정해진다.

- 1바이트 : 아스키 문자
- 2바이트 : 그리스 문자, 히브리어 문자, 아랍어 문자, 라틴 문자 등
- 3바이트 : 중국어, 일본어, 한국어 등
- 4바이트 : 일부 특수 문자, 이모티콘 (일반적으로 사용하지 않는다.)

```
case uni16BE_NoBOM:    // BOM이 없는 빅 엔디안 UTF-16
case uni16LE_NoBOM:    // BOM이 없는 리틀 엔디안 UTF-16
case uni16BE:          // BOM이 있는 빅 엔디안 UTF-16
case uni16LE: {        // BOM이 있는 리틀 엔디안 UTF-16
        size_t newSize = len + len / 2 + 1;

  if (m_nAllocatedBufSize != newSize)
        {
    if (m_pNewBuf)
      delete [] m_pNewBuf;
          m_pNewBuf  = NULL;
          m_pNewBuf  = new ubyte[newSize];
    m_nAllocatedBufSize = newSize;
        }

        ubyte* pCur = m_pNewBuf;

        m_Iter16.set(m_pBuf + nSkip, len - nSkip, m_eEncoding);
```

[그림 19] Utf8_16_Read::convert - 3

이 때문에 UTF-16에서 UTF-8로 변환하는 데 필요한 정확한 바이트 수를 계산하기 어렵다. 따라서 2바이트 UTF-16 문자를 3바이트 UTF-8 문자로 변환해야 하는 최악의 경우를 기반으로 한 길이를 계산하게 된다.

d) poc.py 실행

```python
with open("poc", "wb") as f:
    f.write(b'\xfe\xff')
    f.write(b'\xff' * (128 * 1024 + 4 - 2 + 1))
```

[그림 20] poc.py

사용한 poc.py는 다음과 같다.

```
const Utf8_16::utf8 Utf8_16::k_Boms[][3] = {
    {0x00, 0x00, 0x00},   // Unknown
    {0xEF, 0xBB, 0xBF},   // UTF8
    {0xFE, 0xFF, 0x00},   // Big endian
    {0xFF, 0xFE, 0x00},   // Little endian
};
```

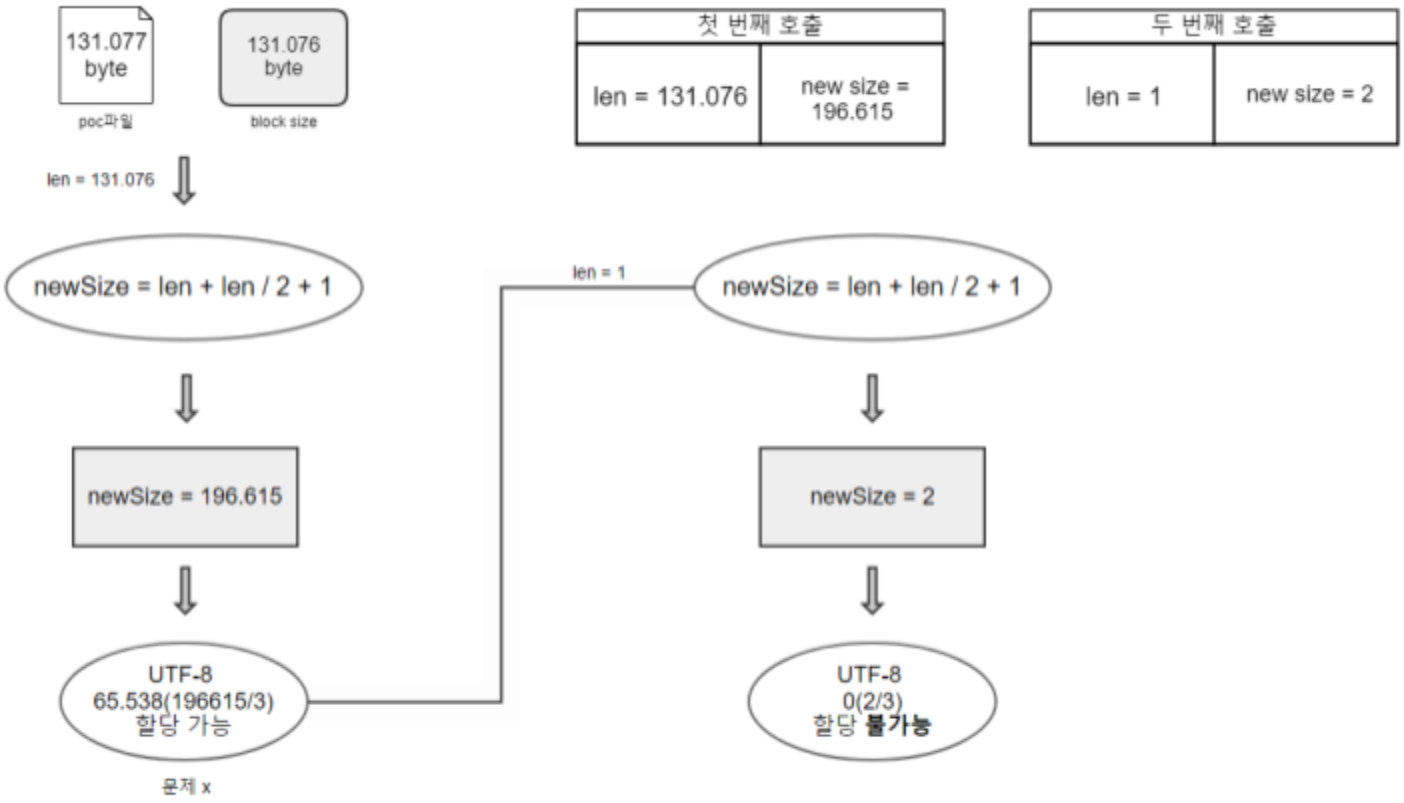
[그림 21] BOM

앞의 '\xfe\xff'는 BOM(Byte Order Mark)으로, 이로 인해 해당 파일을 Notepad++에서 읽을 때 UTF-16BE로 인식하게 된다. 그리고 '\xff'를 128 * 1024 + 4 - 2 + 1, 즉, 131,075바이트 작성하므로 BOM 크기와 합하면 총 131,077바이트가 된다.

```
static const int blockSize = 128 * 1024 * 4; // 131,076
...
do
{
  lenFile = fread(data + incompleteMultibyteChar, 1, blockSize - incompleteMultibyteChar, fp) + incompleteMultibyteChar; // incompleMultibyteChar = 0
  if (ferror(fp) != 0)
  {
    success = false;
    break;
  }
  if (lenFile == 0) break; // 다 읽어들일 부분이 없을 경우 종단
  ...
}
while (lenFile > 0);
```

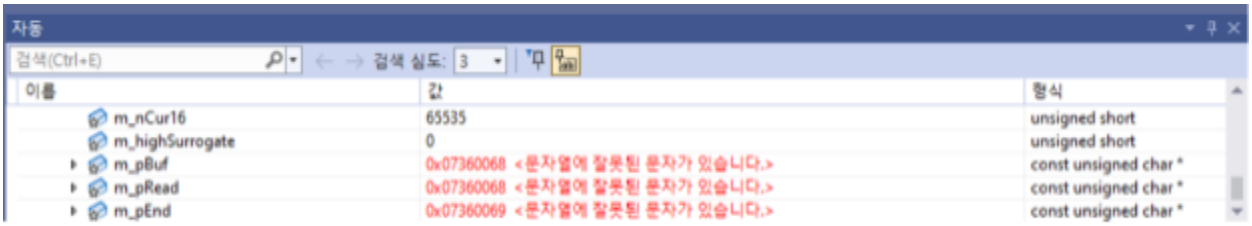
[그림 22] FileManager::loadFileData - 2

Buffer.cpp의 FileManager::loadFileData 함수에서는 파일을 읽어들일 때에는 항상 한 번에 blockSize인 131,076바이트만큼만 읽어올 수 있음을 확인할 수 있다.이는 poc.py에서 작성하는 바이트 크기인 131,077바이트보다 1바이트 작은 크기이다.



[그림 23] poc 실행 과정

때문에 fread가 처음 호출될 때는 1바이트를 남기고 131,076바이트를 읽게 된다. 이때, newSize의 값은 len + len / 2 + 1의 계산식으로 결정되므로, 131,076 + 131,076 / 2 + 1의 계산 과정을 거쳐 196,615가 된다. 이는 65,538개의 UTF-16 문자를 UTF-8 문자로 변환하여 저장할 수 있어, 문제가 발생하지 않는다. fread가 두 번째로 호출될 때는 poc.py에서 1바이트가 남은 상태이므로, len이 1로 설정된다. 그러나 UTF-16 문자의 크기인 2바이트를 참조해야 하는 상황이다. 실제로 여기서 newSize는 1 + 1 / 2 + 1의 계산 과정을 거쳐 2가 되고, 이에 따라 2바이트의 버퍼를 새롭게 할당해야 한다. 따라서 poc.py에서 남은 데이터 1바이트와 쓰레기 값 1바이트를 가져와 UTF-8로 변환하게 된다.

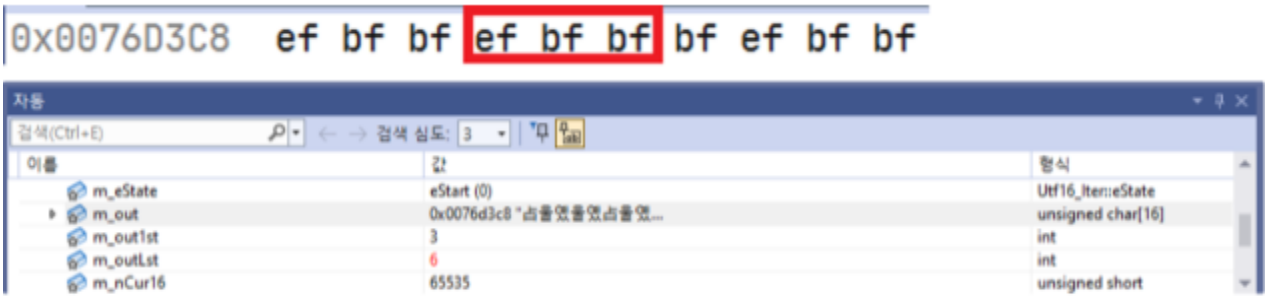


[그림 24] m_pRead 쓰레기 값 - 1

m_pRead는 UTF-16 버퍼의 시작 위치를 가리키고 있는데, 해당 위치인 0x7360069에는 초기화되지 않은 쓰레기 값이 들어 있다.

[그림 25] m_pRead 쓰레기 값 - 2

해당 쓰레기 값은 첫 번째 fread 호출 때 남은 poc.py의 데이터 '\xff'이다.



[그림 26] UTF-8 변환 크기 파악

UTF-8은 가변 길이이기 때문에 m_out1st와 m_outLst 변수 값의 오프셋을 통해 크기를 파악해야 한다. [그림 26]을 보면 m_out1st는 UTF-8로의 변환 결과를 참조하기 시작하는 인덱스, m_outLst는 마지막 경계 인덱스이므로 0xffff가 UTF-8로 변환될 때 3~5 인덱스를 참조하게 되어 3바이트로 변환된다는 것을 알 수 있다.

[그림 27] 힙 버퍼 오버플로우 발생

실제로, 변환된 결과는 3바이트로 2바이트에서 1바이트 넘치게 되므로 힙 버퍼 오버플로우가 발생하는 것을 확인할 수 있다.

즉, poc 파일 바이트 크기가 홀수라면 UTF-16을 UTF-8로 변환하는 과정에서 버퍼 크기 계산에 오류가 발생하여 힙 버퍼 오버플로우가 발생하게 된다. 만약 오버플로우가 발생한 1바이트에 다른 메모리를 덮어쓴다면 임의 코드가 실행될 수 있다.

e) 패치

```
size_t newSize = len + len / 2 + 1;
```

[그림 28] newSize 패치 이전 코드

```
size_t newSize = (len + len % 2) + (len + len % 2) / 2;
```

[그림 29] newSize 패치 이후 코드

UTF-16을 UTF-8로 변환하는 과정에서 계산하는 새로운 버퍼 크기의 계산식이 패치되었다. 위와 같은 코드를 사용하면 len의 값이 홀수라도 3바이트 UTF-8로 변환될 때 힙 버퍼 오버플로우가 일어나지 않도록 크기 할당이 이루어지게 된다.

04. 참고 문헌

[1]
https://securitylab.github.com/advisories/GHSL-2023-092_Notepad__/, Jaroslav Lobacevski

[2]
https://github.com/webraybtl/CVE-2023-40031/blob/main/notepad++%E5%A0%86%E7%BC%93%E5%86%B2%E5%8C%BA%E6%BA%A2%E5%87%BA%E6%BC%8F%E6%B4%9ECVE-2023-40031%E5%88%86%E6%9E%90%E4%B8%8E%E5%A4%8D%E7%8E%B0.docx, webraybtl

[3]
https://learn.microsoft.com/ko-kr/cpp/sanitizers/asan?view=msvc-170, Microsoft