**24.01.17 보고서**

**서강대학교 20191244 김현승**

* **Analyze vulnerability in dact\_common.c**

퍼징 결과를 분석했을 때, dact\_common.c의 dact\_process\_file() 함수에서 stack-based buffer overflow가 발생했음을 알 수 있었다. 코드를 분석해보면 dact\_process\_file() 함수는 현재 모드(compression 또는 decompression)에 따라서 파일을 읽거나 쓰는 파일 처리 작업을 하고 있다. 취약점이 트리거 된 부분은 decompression이므로 해당 부분을 살펴보자.

if (mode==DACT\_MODE\_DECMP) {

        dact\_ui\_status(DACT\_UI\_LVL\_GEN, "Decompressing.");

        dact\_hdr\_ext\_clear();

        read\_de(src, &magic, 4, sizeof(magic));

        if (magic!=DACT\_MAGIC\_NUMBER) {

            dact\_ui\_status(DACT\_UI\_LVL\_GEN, "Bad DACT magic, checking others...");

            return(dact\_process\_other(src,dest,magic,options));

        }

        read(src, &version[0], 1);

        read(src, &version[1], 1);

        read(src, &version[2], 1);

#ifndef DACT\_DONT\_SUPPORT\_OLDDACT

        if (DACT\_VERS(version[0], version[1], version[2])<DACT\_VERS(0, 8, 39)) {

            PRINTERR("\*\*WARNING\*\* This file uses the old DACT file header, support will go away in future versions for this.");

            read\_de(src, &filesize, 4, sizeof(filesize));

            hdr\_reg\_size=24;

        } else {

            read\_de(src, &filesize, 8, sizeof(filesize));

        }

#else

        read\_de(src, &filesize, 8, sizeof(filesize));

#endif

        read\_de(src, &blk\_cnt, 4, sizeof(blk\_cnt));

        read\_de(src, &blksize\_uncomp, 4, sizeof(blksize\_uncomp));

        read(src, &file\_opts, 1);

        read\_de(src, &file\_extd\_size, 4, sizeof(file\_extd\_size));

        while (file\_extd\_read<file\_extd\_size) {

            x=0;

            read(src, &ch, 1);

            if (ch!=DACT\_HDR\_NOP) read\_de(src, &x, 2, sizeof(x));

            switch (ch) {

                case DACT\_HDR\_CRC0:

                    read\_de(src, &crcs[2], 4, sizeof(crcs[2]));

                    if (crcs[4]!=0 && crcs[2]!=crcs[4]) {

                        dact\_ui\_status(DACT\_UI\_LVL\_GEN, "CRC error.");

                        if (!options[DACT\_OPT\_NOCRC])

                            return(0);

                    }

                    break;

                case DACT\_HDR\_CRC1:

                    read\_de(src, &crcs[3], 4, sizeof(crcs[3]));

                    if (crcs[5]!=0 && crcs[3]!=crcs[5]) {

                        dact\_ui\_status(DACT\_UI\_LVL\_GEN, "CRC error.");

                        if (!options[DACT\_OPT\_NOCRC])

                            return(0);

                    }

                    break;

                case DACT\_HDR\_URL:

                    hdr\_buf=malloc(x+1);

                    read\_f(src, hdr\_buf, x);

                    hdr\_buf[x]=0;

                    file\_extd\_urls[file\_extd\_urlcnt++]=parse\_url\_subst(hdr\_buf,filename); // 취약점 발생

                    free(hdr\_buf);

                    break;

위는 dact\_process\_file()의 압축 해제 관련 부분 코드의 일부이다. read\_de, read, read\_f 등의 함수로 src file descriptor가 가리키는 파일(압축 해제할 파일)에서 내용을 읽어와 변수들에 저장하고 있다. read\_de, read, read\_f 가 호출되는 부분만 모아서 보면 다음과 같다.

read\_de(src, &magic, 4, sizeof(magic));

read(src, &version[0], 1);

read(src, &version[1], 1);

read(src, &version[2], 1);

read\_de(src, &filesize, 8, sizeof(filesize));

read\_de(src, &blk\_cnt, 4, sizeof(blk\_cnt));

read\_de(src, &blksize\_uncomp, 4, sizeof(blksize\_uncomp));

read(src, &file\_opts, 1);

read\_de(src, &file\_extd\_size, 4, sizeof(file\_extd\_size));

// 이후에는 while문 안에서 ch에 따라 읽음

read(src, &ch, 1);

if (ch!=DACT\_HDR\_NOP) read\_de(src, &x, 2, sizeof(x));

파일의 앞에서부터 4바이트 magic, 3바이트 version, 8바이트 filesize, 4바이트 blk\_cnt, 4바이트 blksize\_uncomp, 1바이트 file\_opts, 4바이트 file\_extd\_size, 1바이트 ch, 2바이트는 x이다. 고정적으로 file\_extd\_size까지 총 28바이트를 읽고 이후 while문에서 file\_extd\_read가 file\_extd\_size보다 크거나 같아질 때 까지 파일을 읽어 ch, x, hdr\_buf에 저장하고 있다. ch는 DACT\_HDR\_XXX, 즉 header 값이고, x는 각 헤더에 맞게 읽어야 할 사이즈, hdr\_buf는 헤더에 저장된 내용을 받는다. file\_extd\_read는 각 루프마다 x+3만큼의 값이 누적되고 있다. +3을 해주는 이유는 ch, x를 각각 1, 2바이트씩 읽기 때문이다. ch가 DACT\_HDR\_NOP인 경우 x를 읽지 않으므로 x값을 -2로 설정하는 코드가 있다.

#define DACT\_HDR\_CRC0   0

#define DACT\_HDR\_CRC1   1

#define DACT\_HDR\_TIME   2

#define DACT\_HDR\_PERM   3

#define DACT\_HDR\_NAME   4

#define DACT\_HDR\_MD5SUM 5

#define DACT\_HDR\_DESC   6

#define DACT\_HDR\_URL    7

#define DACT\_HDR\_URLFILE    8

#define DACT\_HDR\_CIPHER 9

#define DACT\_HDR\_NOP    10

#define DACT\_HDR\_IDXDATA    11

#define DACT\_HDR\_SFXLEN 12

위는 dact.h에 정의된 ch값에 들어갈 DACT\_HDR\_XXX들의 값이다. 이 사실들에 기반해 초기 test case로 사용한 hello.txt.dct 파일을 분석해보자.

dmskhs0912@khs-ubuntu:~/afl\_project/dacttest/input$ xxd hello.txt.dct

00000000: 4443 54c3 0008 2a00 0000 0000 0000 0600 DCT...\*.........

00000010: 0000 0100 0000 0b00 0000 001a 0400 0968 ...............h

00000020: 656c 6c6f 2e74 7874 0000 0412 e602 1f01 ello.txt........

00000030: 0004 12e6 021f 000b 6865 6c6c 6f0a 0000 ........hello...

00000040: 0000 00 ...

**magic version filesize blk\_cnt blksize\_uncomp file\_opts file\_extd\_size ch x NAME CRC(0, 1)**

file\_extd\_size가 0x1a, 즉 26바이트 이므로 0x12e6021f까지 총 26바이트를 while문 안에서 읽는다. 이후에는 while문을 탈출해 읽게 된다.

while (1) {

            if (read(src, &algo, 1)==0) break;

            if (algo==0xff) break; /\* 0xff is reserved for EOF \*/

            read\_de(src, &blksize, blksize\_size, sizeof(blksize));

            if ((in\_buf=malloc(blksize))==NULL) {

                PERROR("malloc");

                free(out\_buf);

                return(0);

            }

            read\_f(src, in\_buf, blksize);

while문 탈출 후 여러 다른 작업 후 위의 while문에서 다시 파일을 읽는다. 파일로부터 1바이트 algo, blksize\_size바이트 blksize, blksize 바이트 in\_buf를 읽는다. 여기서 blksize\_size는 블록 사이즈를 저장하는 수의 사이즈이다. blksize만큼 데이터를 쪼개서 저장하는듯 하다. dact는 파일을 여러 블록으로 쪼개서 각 블록에 적절한 압축 알고리즘을 사용하고 있는데, 여기서 해당 블록에 적용된 압축 알고리즘을 가리키는 값이 algo이다. 위 hello.txt.dct 내용을 다시 살펴보자.

dmskhs0912@khs-ubuntu:~/afl\_project/dacttest/input$ xxd hello.txt.dct

00000000: 4443 54c3 0008 2a00 0000 0000 0000 0600 DCT...\*.........

00000010: 0000 0100 0000 0b00 0000 001a 0400 0968 ...............h

00000020: 656c 6c6f 2e74 7874 0000 0412 e602 1f01 ello.txt........

00000030: 0004 12e6 021f 000b 6865 6c6c 6f0a 0000 ........hello...

00000040: 0000 00 ...

**algo blksize block data**

hello.txt의 내용은 “hello”이다. 0번 알고리즘을 사용하고 있는데, algorithms.h 파일에서 살펴보면 Uncompressed에 해당한다. 즉, 압축 알고리즘을 사용하지 않은 블록이다. 내 생각에는 블록의 데이터가 너무 작기 때문에 아마 압축이 적용되지 않은 것 같다(이 부분을 확인하려면 압축하고 적절한 알고리즘을 선택하는 부분의 코드를 살펴보아야 하지만 exploit에는 관계가 없으므로 생략하였다). blksize가 0xb, 즉 11바이트이고 다음 11바이트만큼이 블록 데이터에 해당한다.

기본적인 dct 파일의 포맷을 이해했으니 이제 취약점이 트리거된 targetcrash 파일을 분석해보고 왜 BOF가 발생했는지 확인해보자. BOF가 ch값이 DACT\_HDR\_URL(7) 일 때 발생했으므로 해당 부분을 중심적으로 살펴본다.

dmskhs0912@khs-ubuntu:~/afl\_project/dacttest/output/crashes$ xxd targetcrash

00000000: 4443 54c3 4444 4354 c301 2a00 0000 0100 DCT.DDCT..\*.....

00000010: 0600 0600 0000 0100 0707 0707 0707 000b ................

00000020: 0000 1d04 1a04 0009 0000 0006 001a 0400 ................

00000030: 0968 0707 0707 0007 0400 0707 0000 0001 .h..............

00000040: 8000 000b 0007 0707 0707 0707 1a1f 6f00 ..............o.

00000050: 0000 0968 4443 54c3 656c 6c07 0707 0707 ...hDCT.ell.....

00000060: 0700 0b00 001d 041a 0400 0900 0000 0600 ................

00000070: 1a04 0009 6865 6c6c 0707 0707 0707 0707 ....hell........

00000080: 0707 0707 0707 0707 0000 0001 0000 000b ................

00000090: 0000 00 ...

**file\_extd\_size ch x CRC0 URL**

ch가 DACT\_HDR\_URL일 때 x(size)가 0x700, 십진수로 1792 바이트이다. 따라서 남은 부분을 모두 URL로 읽게 된다. DACT\_HDR\_URL을 처리하는 부분의 코드를 다시 살펴보자.

case DACT\_HDR\_URL:

                    hdr\_buf=malloc(x+1);

                    read\_f(src, hdr\_buf, x);

                    hdr\_buf[x]=0;

                    file\_extd\_urls[file\_extd\_urlcnt++]=parse\_url\_subst(hdr\_buf,filename); // 취약점 발생

                    free(hdr\_buf);

                    break;

src로부터 x만큼, 여기서는 0x700 바이트만큼 읽어 hdr\_buf에 저장하고 file\_exd\_urls 배열에 적절한 파싱 작업 후 내용을 저장하게 된다. file\_extd\_urls 배열은 256개의 원소를 갖는 배열이고, file\_extd\_urlcnt는 dact\_process\_file에서 0으로 초기화 되어있다. 즉, DACT\_HDR\_URL의 case가 발생할 때마다 urlcnt가 하나씩 증가해 적절하게 배열에 내용을 저장하고 있다. 위 targetcrash 파일의 경우 read\_f로 남은 모든 파일의 내용을 읽은 후 file\_extd\_urls[0]에 내용을 저장하고 break를 만나 file\_extd\_read에 x+3, 즉 0x703만큼 값을 더한 후 다시 while문으로 돌아간다.

        while (file\_extd\_read<file\_extd\_size) {

            x=0;

            read(src, &ch, 1);

            if (ch!=DACT\_HDR\_NOP) read\_de(src, &x, 2, sizeof(x));

            switch (ch) {

현재 file\_extd\_read 값은 0x703, file\_extd\_size는 0x1000600이므로 x값을 0으로 초기화 후 ch값을 읽게 된다. 하지만, 파일의 모든 내용을 읽었으므로 ch값은 변하지 않고 그대로 DACT\_HDR\_URL값을 유지하게 되고, 아래의 if문을 통과해 x값을 읽지만 역시 x는 0으로 초기화된 상태에서 변하지 않는다. 이제 다시 case DACT\_HDR\_URL로 분기해 file\_extd\_urlcnt가 1 증가하고 file\_extd\_read 값은 3 증가한 채로 while문을 반복하게 된다. 한 번 반복할 때마다 file\_extd\_read 값이 3씩 증가하게 되는데, file\_extd\_size는 0x1000600이므로 충분히 많이 반복하며 file\_extd\_urlcnt 값을 증가시키게 된다. 즉 file\_extd\_urls 배열의 범위가 넘는 부분까지 스택의 내용이 덮어 씌워지게 되어 스택 버퍼 오버플로우가 발생하는 것이다.