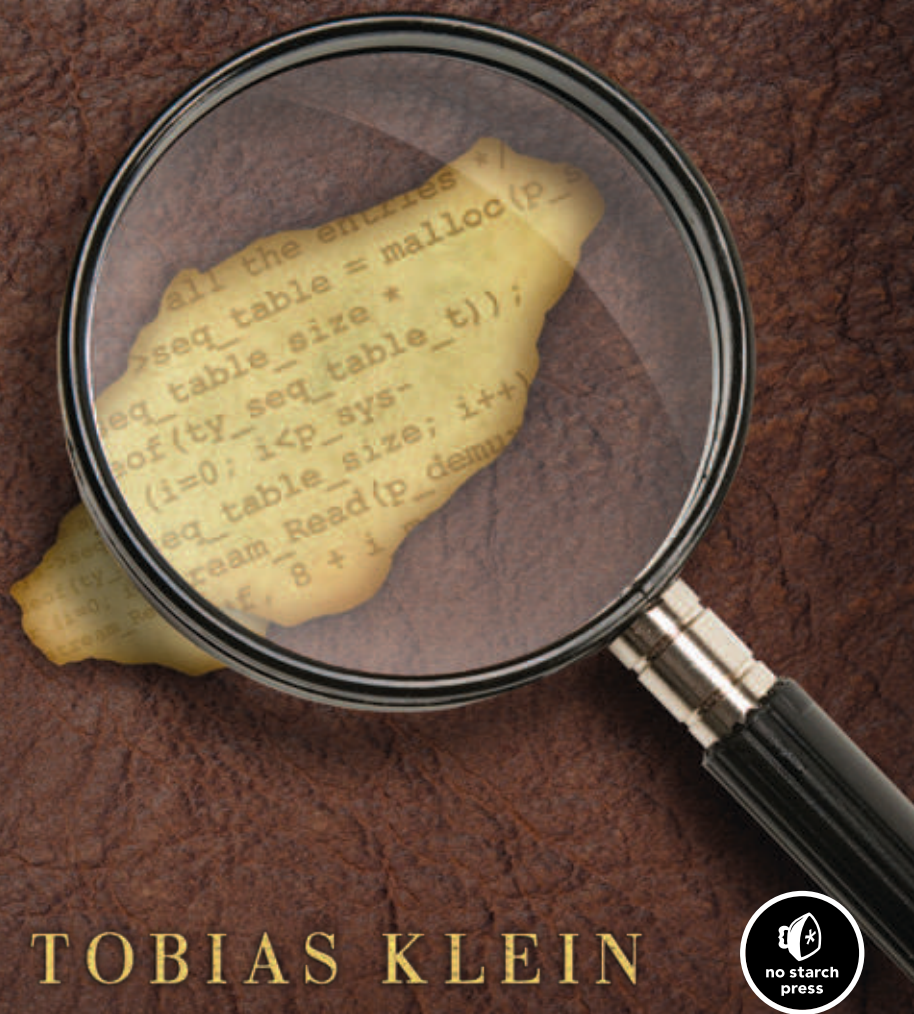


A Bug Hunter's Diary

A Guided Tour Through the Wilds
of Software Security



TOBIAS KLEIN



2

BACK TO THE '90S

Sunday, October 12, 2008

Dear Diary,

I had a look at the source code of VideoLAN's popular VLC media player today. I like VLC because it supports all different kinds of media files and runs on all my favorite operating system platforms. But supporting all those different media file formats has downsides. VLC does a lot of parsing, and that often means a lot of bugs just waiting to be discovered.

NOTE *According to Parsing Techniques: A Practical Guide by Dick Grune and Ceriel J.H. Jacobs,¹ "Parsing is the process of structuring a linear representation in accordance with a given grammar." A parser is software that breaks apart a raw string of bytes into individual words and statements. Depending on the data format, parsing can be a very complex and error-prone task.*

After I became familiar with the inner workings of VLC, finding the first vulnerability took me only about half a day. It was a classic stack buffer overflow (see Section A.1). This one occurred while

parsing a media file format called TiVo, the proprietary format native to TiVo digital recording devices. Before finding this bug, I had never heard of this file format, but that didn't stop me from exploiting it.

2.1 Vulnerability Discovery

Here is how I found the vulnerability:

- Step 1: Generate a list of the demuxers of VLC.
- Step 2: Identify the input data.
- Step 3: Trace the input data.

← I used VLC 0.9.4 on the Microsoft Windows Vista SP1 (32-bit) platform for all the following steps.

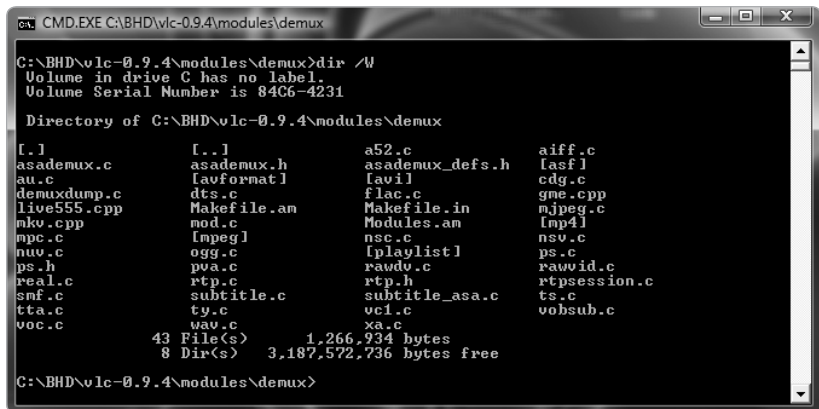
I'll explain this process in detail in the following sections.

Step 1: Generate a List of the Demuxers of VLC

After downloading and unpacking the source code of VLC,² I generated a list of the available demuxers of the media player.

NOTE *In digital video, demuxing or demultiplexing refers to the process of separating audio and video as well as other data from a video stream or container in order to play the file. A demuxer is software that extracts the components of such a stream or container.*

Generating a list of demuxers wasn't too hard, as VLC separates most of them in different C files in the directory `vlc-0.9.4\modules\demux\` (see Figure 2-1).



```
CMD.EXE C:\BHD\vlc-0.9.4\modules\demux
C:\BHD\vlc-0.9.4\modules\demux>dir /W
Volume in drive C has no label.
Volume Serial Number is 84C6-4231

Directory of C:\BHD\vlc-0.9.4\modules\demux

[.]                [..]                a52.c              aiff.c
asademux.c         asademux.h          asademux_defs.h  [asf]
au.c               [avformat]          flac.c            cdg.c
demuxdump.c        dts.c               Makefile.am       gme.cpp
live555.cpp         mod.c               Modules.am        mjpeg.c
mkv.cpp            [mpeg]              nsc.c             mp4l
mp3.c              ogg.c               [playlist]        nsu.c
nuv.c              pva.c               rawdv.c           ps.c
ps.h               rtp.c               rtpsession.c     rauvid.c
real.c             subtitle.c          subtitle_asa.c   rtpsession.c
smf.c              ty.c                vc1.c             ts.c
tta.c              wav.c               xa.c              vobsub.c
voc.c

            43 File(s)            1,266,934 bytes
            8 Dir(s)             3,187,572,736 bytes free

C:\BHD\vlc-0.9.4\modules\demux>
```

Figure 2-1: VLC demuxer list

Step 2: Identify the Input Data

Next, I tried to identify the input data processed by the demuxers. After reading some C code, I stumbled upon the following structure, which is declared in a header file included in every demuxer.

Source code file `vlc-0.9.4\include\vlc_demux.h`

```
[..]
41 struct demux_t
42 {
43     VLC_COMMON_MEMBERS
44
45     /* Module properties */
46     module_t    *p_module;
47
48     /* eg informative but needed (we can have access+demux) */
49     char        *psz_access;
50     char        *psz_demux;
51     char        *psz_path;
52
53     /* input stream */
54     stream_t    *s;      /* NULL in case of a access+demux in one */
[..]
```

In line 54, the structure element `s` is declared and described as `input stream`. This was exactly what I was searching for: a reference to the input data that is processed by the demuxers.

Step 3: Trace the Input Data

After I discovered the `demux_t` structure and its input stream element, I searched the demuxer files for references to it. The input data was usually referenced by `p_demux->s`, as shown in lines 1623 and 1641 below. When I found such a reference, I traced the input data while looking for coding errors. Using this approach, I found the following vulnerability.

Source code file `vlc-0.9.4\modules\demux\Ty.c`

Function `parse_master()`

```
[..]
1623 static void parse_master(demux_t *p_demux)
1624 {
1625     demux_sys_t *p_sys = p_demux->p_sys;
1626     uint8_t mst_buf[32];
1627     int i, i_map_size;
1628     int64_t i_save_pos = stream_Tell(p_demux->s);
1629     int64_t i_pts_secs;
1630
1631     /* Note that the entries in the SEQ table in the stream may have
1632     different sizes depending on the bits per entry. We store them
1633     all in the same size structure, so we have to parse them out one
1634     by one. If we had a dynamic structure, we could simply read the
1635     entire table directly from the stream into memory in place. */
```

```

1636
1637 /* clear the SEQ table */
1638 free(p_sys->seq_table);
1639
1640 /* parse header info */
1641 stream_Read(p_demux->s, mst_buf, 32);
1642 i_map_size = U32_AT(&mst_buf[20]); /* size of bitmask, in bytes */
1643 p_sys->i_bits_per_seq_entry = i_map_size * 8;
1644 i = U32_AT(&mst_buf[28]); /* size of SEQ table, in bytes */
1645 p_sys->i_seq_table_size = i / (8 + i_map_size);
1646
1647 /* parse all the entries */
1648 p_sys->seq_table = malloc(p_sys->i_seq_table_size * sizeof(ty_seq_table_t));
1649 for (i=0; i<p_sys->i_seq_table_size; i++) {
1650     stream_Read(p_demux->s, mst_buf, 8 + i_map_size);
1651 [..]

```

The `stream_Read()` function in line 1641 reads 32 bytes of user-controlled data from a TiVo media file (referenced by `p_demux->s`) and stores them in the stack buffer `mst_buf`, declared in line 1626. The `U32_AT` macro in line 1642 then extracts a user-controlled value from `mst_buf` and stores it in the signed int variable `i_map_size`. In line 1650, the `stream_Read()` function stores user-controlled data from the media file in the stack buffer `mst_buf` again. But this time, `stream_Read()` uses the user-controlled value of `i_map_size` to calculate the size of the data that gets copied into `mst_buf`. This leads to a straight stack buffer overflow (see Section A.1) that can be easily exploited.

Here is the anatomy of the bug, as illustrated in Figure 2-2:

1. 32 bytes of user-controlled TiVo media file data are copied into the stack buffer `mst_buf`. The destination buffer has a size of 32 bytes.
2. 4 bytes of user-controlled data are extracted from the buffer and stored in `i_map_size`.
3. User-controlled TiVo media-file data is copied into `mst_buf` once again. This time, the size of the copied data is calculated using `i_map_size`. If `i_map_size` has a value greater than 24, a stack buffer overflow will occur (see Section A.1).

2.2 Exploitation

To exploit the vulnerability, I performed the following steps:

- Step 1: Find a sample TiVo movie file.
- Step 2: Find a code path to reach the vulnerable code.
- Step 3: Manipulate the TiVo movie file to crash VLC.
- Step 4: Manipulate the TiVo movie file to gain control of EIP.

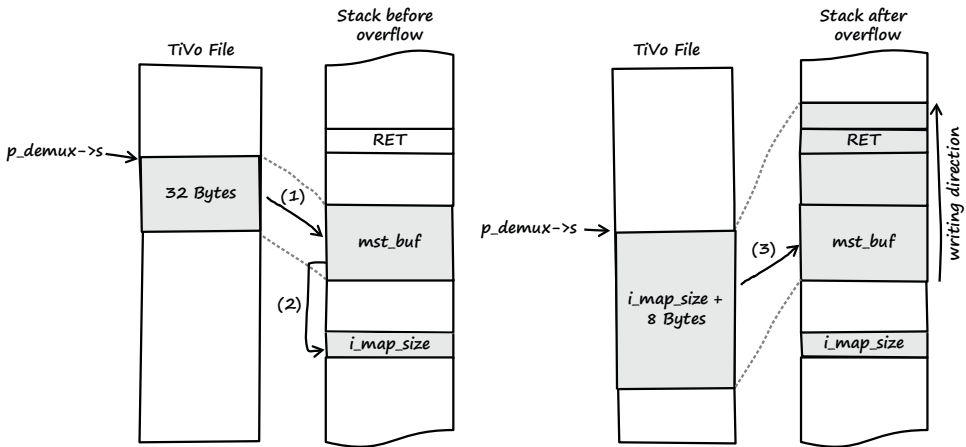


Figure 2-2: Overview of the vulnerability from input to stack buffer overflow

There's more than one way to exploit a file-format bug. You can create a file with the right format from scratch, or you can manipulate a valid preexisting file. I chose the latter in this example.

Step 1: Find a Sample TiVo Movie File

First I downloaded the following TiVo sample file from <http://samples.mplayerhq.hu/>:

← The website <http://samples.mplayerhq.hu/> is a good starting point to search for all kinds of multimedia file-format samples.

```
$ wget http://samples.mplayerhq.hu/TiVo/test-dtivo-junkskip.ty%2b
--2008-10-12 21:12:25-- http://samples.mplayerhq.hu/TiVo/test-dtivo-junkskip.ty%2b
Resolving samples.mplayerhq.hu... 213.144.138.186
Connecting to samples.mplayerhq.hu|213.144.138.186|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 5242880 (5.0M) [text/plain]
Saving to: `test-dtivo-junkskip.ty+'

100[=====] 5,242,880 240K/s in 22s

2008-10-12 21:12:48 (232 KB/s) - `test-dtivo-junkskip.ty+' saved [5242880/5242880]
```

Step 2: Find a Code Path to Reach the Vulnerable Code

I couldn't find documentation on the specifications of the TiVo file format, so I read the source code in order to find a path to reach the vulnerable code in `parse_master()`.

If a TiVo file is loaded by VLC, the following execution flow is taken (all source code references are from *vlc-0.9.4/modules/demux/Ty.c* of VLC). The first relevant function that's called is `Demux()`:

```
[..]
386 static int Demux( demux_t *p_demux )
387 {
388     demux_sys_t *p_sys = p_demux->p_sys;
389     ty_rec_hdr_t *p_rec;
390     block_t      *p_block_in = NULL;
391
392     /*msg_Dbg(p_demux, "ty demux processing" );*/
393
394     /* did we hit EOF earlier? */
395     if( p_sys->eof )
396         return 0;
397
398     /*
399     * what we do (1 record now.. maybe more later):
400     * - use stream_Read() to read the chunk header & record headers
401     * - discard entire chunk if it is a PART header chunk
402     * - parse all the headers into record header array
403     * - keep a pointer of which record we're on
404     * - use stream_Block() to fetch each record
405     * - parse out PTS from PES headers
406     * - set PTS for data packets
407     * - pass the data on to the proper codec via es_out_Send()
408
409     * if this is the first time or
410     * if we're at the end of this chunk, start a new one
411     */
412     /* parse the next chunk's record headers */
413     if( p_sys->b_first_chunk || p_sys->i_cur_rec >= p_sys->i_num_recs )
414     {
415         if( get_chunk_header(p_demux) == 0 )
416
417     }
418
419     }
420
421     }
422
423     }
424
425     }
426
427     }
428
429     }
430
431     }
432
433     }
434
435     }
436
437     }
438
439     }
440
441     }
442
443     }
444
445     }
446
447     }
448
449     }
450
451     }
452
453     }
454
455     }
456
457     }
458
459     }
460
461     }
462
463     }
464
465     }
466
467     }
468
469     }
470
471     }
472
473     }
474
475     }
476
477     }
478
479     }
480
481     }
482
483     }
484
485     }
486
487     }
488
489     }
490
491     }
492
493     }
494
495     }
496
497     }
498
499     }
500
501     }
502
503     }
504
505     }
506
507     }
508
509     }
510
511     }
512
513     }
514
515     }
516
517     }
518
519     }
520
521     }
522
523     }
524
525     }
526
527     }
528
529     }
530
531     }
532
533     }
534
535     }
536
537     }
538
539     }
540
541     }
542
543     }
544
545     }
546
547     }
548
549     }
550
551     }
552
553     }
554
555     }
556
557     }
558
559     }
560
561     }
562
563     }
564
565     }
566
567     }
568
569     }
570
571     }
572
573     }
574
575     }
576
577     }
578
579     }
580
581     }
582
583     }
584
585     }
586
587     }
588
589     }
590
591     }
592
593     }
594
595     }
596
597     }
598
599     }
600
601     }
602
603     }
604
605     }
606
607     }
608
609     }
610
611     }
612
613     }
614
615     }
616
617     }
618
619     }
620
621     }
622
623     }
624
625     }
626
627     }
628
629     }
630
631     }
632
633     }
634
635     }
636
637     }
638
639     }
640
641     }
642
643     }
644
645     }
646
647     }
648
649     }
650
651     }
652
653     }
654
655     }
656
657     }
658
659     }
660
661     }
662
663     }
664
665     }
666
667     }
668
669     }
670
671     }
672
673     }
674
675     }
676
677     }
678
679     }
680
681     }
682
683     }
684
685     }
686
687     }
688
689     }
690
691     }
692
693     }
694
695     }
696
697     }
698
699     }
700
701     }
702
703     }
704
705     }
706
707     }
708
709     }
710
711     }
712
713     }
714
715     }
716
717     }
718
719     }
720
721     }
722
723     }
724
725     }
726
727     }
728
729     }
730
731     }
732
733     }
734
735     }
736
737     }
738
739     }
740
741     }
742
743     }
744
745     }
746
747     }
748
749     }
750
751     }
752
753     }
754
755     }
756
757     }
758
759     }
760
761     }
762
763     }
764
765     }
766
767     }
768
769     }
770
771     }
772
773     }
774
775     }
776
777     }
778
779     }
780
781     }
782
783     }
784
785     }
786
787     }
788
789     }
790
791     }
792
793     }
794
795     }
796
797     }
798
799     }
800
801     }
802
803     }
804
805     }
806
807     }
808
809     }
810
811     }
812
813     }
814
815     }
816
817     }
818
819     }
820
821     }
822
823     }
824
825     }
826
827     }
828
829     }
830
831     }
832
833     }
834
835     }
836
837     }
838
839     }
840
841     }
842
843     }
844
845     }
846
847     }
848
849     }
850
851     }
852
853     }
854
855     }
856
857     }
858
859     }
860
861     }
862
863     }
864
865     }
866
867     }
868
869     }
870
871     }
872
873     }
874
875     }
876
877     }
878
879     }
880
881     }
882
883     }
884
885     }
886
887     }
888
889     }
890
891     }
892
893     }
894
895     }
896
897     }
898
899     }
900
901     }
902
903     }
904
905     }
906
907     }
908
909     }
910
911     }
912
913     }
914
915     }
916
917     }
918
919     }
920
921     }
922
923     }
924
925     }
926
927     }
928
929     }
929
930     }
931
932     }
933
934     }
935
936     }
937
938     }
939
940     }
941
942     }
943
944     }
945
946     }
947
948     }
949
950     }
951
952     }
953
954     }
955
956     }
957
958     }
959
960     }
961
962     }
963
964     }
965
966     }
967
968     }
969
970     }
971
972     }
973
974     }
975
976     }
977
978     }
979
980     }
981
982     }
983
984     }
985
986     }
987
988     }
989
990     }
991
992     }
993
994     }
995
996     }
997
998     }
999
1000    }
```

After some sanity checks in lines 395 and 413, the function `get_chunk_header()` is called in line 415.

```
[..]
112 #define TIVO_PES_FILEID ( 0xf5467abd )
[..]
1839 static int get_chunk_header(demux_t *p_demux)
1840 {
1841     int i_readSize, i_num_recs;
1842     uint8_t *p_hdr_buf;
1843     const uint8_t *p_peek;
1844     demux_sys_t *p_sys = p_demux->p_sys;
1845     int i_payload_size; /* sum of all records' sizes */
1846
1847     msg_Dbg(p_demux, "parsing ty chunk %#d", p_sys->i_cur_chunk );
1848
1849     /* if we have left-over filler space from the last chunk, get that */
1850     if ( p_sys->i_stuff_cnt > 0 ) {
```

```

1851     stream_Read( p_demux->s, NULL, p_sys->i_stuff_cnt);
1852     p_sys->i_stuff_cnt = 0;
1853 }
1854
1855 /* read the TY packet header */
1856 i_readSize = stream_Peek( p_demux->s, &p_peek, 4 );
1857 p_sys->i_cur_chunk++;
1858
1859 if ( ( i_readSize < 4 ) || ( U32_AT(&p_peek[ 0 ] ) == 0 ) )
1860 {
1861     /* EOF */
1862     p_sys->eof = 1;
1863     return 0;
1864 }
1865
1866 /* check if it's a PART Header */
1867 if( U32_AT( &p_peek[ 0 ] ) == TIVO_PES_FILEID )
1868 {
1869     /* parse master chunk */
1870     parse_master(p_demux);
1871     return get_chunk_header(p_demux);
1872 }
[..]

```

In line 1856 of `get_chunk_header()`, the user-controlled data from the TiVo file is assigned to the pointer `p_peek`. Then, in line 1867, the process checks whether the file data pointed to by `p_peek` equals `TIVO_PES_FILEID` (which is defined as `0xf5467abd` in line 112). If so, the vulnerable function `parse_master()` gets called (see line 1870).

To reach the vulnerable function using this code path, the TiVo sample file had to contain the value of `TIVO_PES_FILEID`. I searched the TiVo sample file for the `TIVO_PES_FILEID` pattern and found it at file offset `0x00300000` (see Figure 2-3).

```

00300000h: F5 46 7A BD 00 00 00 02 00 02 00 00 00 01 F7 04 ; ðFz%.
00300010h: 00 00 00 08 00 00 00 02 3B 9A CA 00 00 00 01 48 ; .....;šĚ....H

```

Figure 2-3: `TIVO_PES_FILEID` pattern in TiVo sample file

Based on the information from the `parse_master()` function (see the following source code snippet) the value of `i_map_size` should be found at offset 20 (0x14) relative to the `TIVO_PES_FILEID` pattern found at file offset `0x00300000`.

```

[..]
1641     stream_Read(p_demux->s, mst_buf, 32);
1642     i_map_size = U32_AT(&mst_buf[20]); /* size of bitmask, in bytes */
[..]

```

At this point, I had discovered that the TiVo sample file I downloaded already triggers the vulnerable `parse_master()` function, so it wouldn't be necessary to adjust the sample file. Great!

register) was pointing to an invalid memory location (indicated by the message Access violation when executing [20030000] in the status bar of the debugger). This might mean that I could easily gain control of the instruction pointer.

Step 4: Manipulate the TiVo Movie File to Gain Control of EIP

My next step was to determine which bytes of the sample file actually overwrote the return address of the current stack frame so that I could take control of EIP. The debugger stated that EIP had a value of 0x20030000 at the time of the crash. To determine which offset this value is found at, I could try to calculate the exact file offset, or I could simply search the file for the byte pattern. I chose the latter approach and started from file offset 0x00300000. I found the desired byte sequence at file offset 0x0030005c, represented in little-endian notation, and I changed the 4 bytes to the value 0x41414141 (as illustrated in Figure 2-6).

```
00300050h: 56 4A 00 00 03 1F 6C 49 6A A0 25 45 00 00 03 20 ; VJ...lIj %E...
                                     ↓
00300050h: 56 4A 00 00 03 1F 6C 49 6A A0 25 45 41 41 41 41 ; VJ...lIj %AAAA
```

Figure 2-6: New value for EIP in TiVo sample file

I then restarted VLC in the debugger and opened the new file (see Figure 2-7).

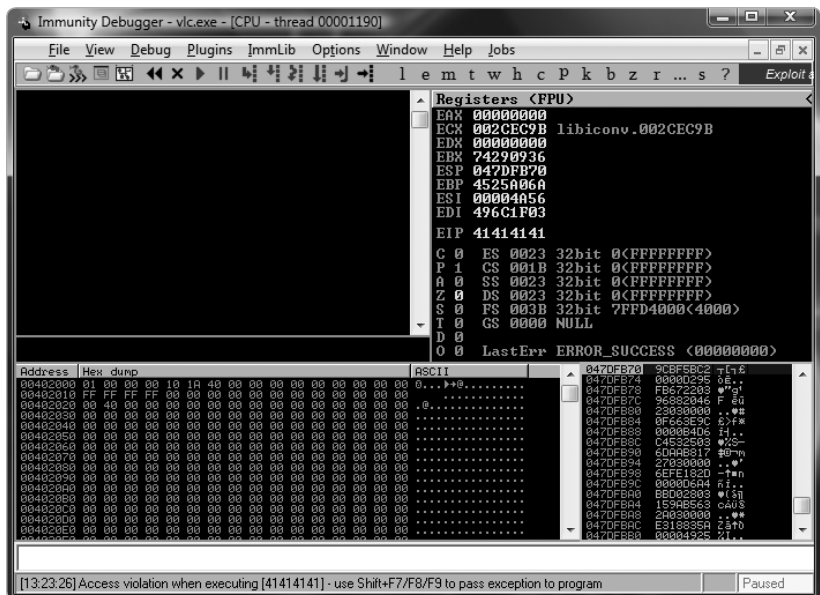


Figure 2-7: EIP control of VLC media player

EIP = 41414141 . . . Mission EIP control accomplished! I was able to build a working exploit, intended to achieve arbitrary code execution, using the well-known `jmp reg` technique, as described in “Variations in Exploit Methods Between Linux and Windows” by David Litchfield.⁴

Since Germany has strict laws against it, I will not provide you with a full working exploit, but if you’re interested, you can watch a short video I recorded that shows the exploit in action.⁵

2.3 Vulnerability Remediation

Saturday, October 18, 2008

Now that I’ve discovered a security vulnerability, I could disclose it in several ways. I could contact the software developer and “responsibly” tell him what I’ve found and help him to create a patch. This process is referred to as *responsible disclosure*. Since this term implies that other means of disclosure are irresponsible, which isn’t necessarily true, it is slowly being replaced by *coordinated disclosure*.

On the other hand, I could sell my findings to a *vulnerability broker* and let him tell the software developer. Today, the two primary players in the commercial vulnerability market are Verisign’s iDefense Labs, with its Vulnerability Contribution Program (VCP), and Tipping Point’s Zero Day Initiative (ZDI). Both VCP and ZDI follow coordinated-disclosure practices and work with the affected vendor.

Another option is *full disclosure*. If I chose full disclosure, I would release the vulnerability information to the public without notifying the vendor. There are other disclosure options, but the motivation behind them usually doesn’t involve fixing the bug (for example, selling the findings in underground markets).⁶

In the case of the VLC vulnerability described in this chapter, I chose coordinated disclosure. In other words, I notified the VLC maintainers, provided them with the necessary information, and coordinated with them on the timing of public disclosure.

After I informed the VLC maintainers about the bug, they developed the following patch to address the vulnerability:⁷

```
--- a/modules/demux/ty.c
+++ b/modules/demux/ty.c
@@ -1639,12 +1639,14 @@ static void parse_master(demux_t *p_demux)
    /* parse all the entries */
    p_sys->seq_table = malloc(p_sys->i_seq_table_size * sizeof(ty_seq_table_t));
    for (i=0; i<p_sys->i_seq_table_size; i++) {
-       stream_Read(p_demux->s, mst_buf, 8 + i_map_size);
+       stream_Read(p_demux->s, mst_buf, 8);
        p_sys->seq_table[i].l.timestamp = U64_AT(&mst_buf[0]);
        if (i_map_size > 8) {
+           msg_Err(p_demux, "Unsupported SEQ bitmap size in master chunk");
+           stream_Read(p_demux->s, NULL, i_map_size);
            memset(p_sys->seq_table[i].chunk_bitmask, i_map_size, 0);
```

```

+     } else {
+         stream_Read(p_demux->s, mst_buf + 8, i_map_size);
+         memcpy(p_sys->seq_table[i].chunk_bitmask, &mst_buf[8], i_map_size);
+     }
}

```

The changes are quite straightforward. The formerly vulnerable call to `stream_Read()` now uses a fixed size value, and the user-controlled value of `i_map_size` is used only as a size value for `stream_Read()` if it is less than or equal to 8. An easy fix for an obvious bug.

But wait—is the vulnerability really gone? The variable `i_map_size` is still of the type signed int. If a value greater than or equal to `0x80000000` is supplied for `i_map_size`, it's interpreted as negative, and the overflow will still occur in the `stream_Read()` and `memcpy()` functions of the `else` branch of the patch (see Section A.3 for a description of unsigned int and signed int ranges). I also reported this problem to the VLC maintainers, resulting in another patch:⁸

```

[..]
@@ -1616,7 +1618,7 @@ static void parse_master(demux_t *p_demux)
{
    demux_sys_t *p_sys = p_demux->p_sys;
    uint8_t mst_buf[32];
-   int i, i_map_size;
+   uint32_t i, i_map_size;
    int64_t i_save_pos = stream_Tell(p_demux->s);
    int64_t i_pts_secs;
[..]

```

Now that `i_map_size` is of the type unsigned int, this bug is fixed. Perhaps you've already noticed that the `parse_master()` function contains another buffer overflow vulnerability. I also reported that bug to the VLC maintainers. If you can't spot it, then take a closer look at the second patch provided by the VLC maintainers, which fixed this bug as well.

One thing that surprised me was the fact that none of the lauded exploit mitigation techniques of Windows Vista were able to stop me from taking control of EIP and executing arbitrary code from the stack using the `jmp reg` technique. The security cookie or /GS feature should have prevented the manipulation of the return address. Furthermore, ASLR or NX/DEP should have prevented arbitrary code execution. (See Section C.1 for a detailed description of all of these mitigation techniques.)

To solve this mystery, I downloaded Process Explorer⁹ and configured it to show the processes' DEP and ASLR status.

NOTE To configure Process Explorer to show the processes' DEP and ASLR status, I added the following columns to the view: **View ▶ Select Columns ▶ DEP Status** and **View ▶ Select Columns ▶ ASLR Enabled**. Additionally, I set the lower pane to view DLLs for a process and added the "ASLR Enabled" column.

The output of Process Explorer, illustrated in Figure 2-8, shows that VLC and its modules use neither DEP nor ASLR (this is denoted by an empty value in the DEP and ASLR columns). I investigated further to determine why the VLC process does not use these mitigation techniques.

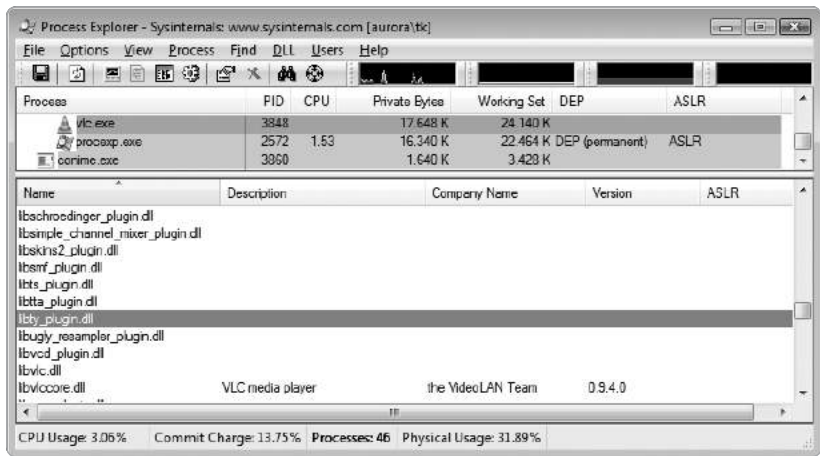


Figure 2-8: VLC in Process Explorer

DEP can be controlled by system policy through special APIs and compile-time options (see Microsoft's Security Research and Defense blog¹⁰ for more information on DEP). The default system-wide DEP policy for client operating systems such as Windows Vista is called OptIn. In this mode of operation, DEP is enabled only for processes that explicitly opt in to DEP. Because I used a default installation of Windows Vista 32-bit, the system-wide DEP policy should be set to OptIn. To verify this, I used the `bcdedit.exe` console application from an elevated command prompt:

```
C:\Windows\system32>bcdedit /enum | findstr nx
nx                               OptIn
```

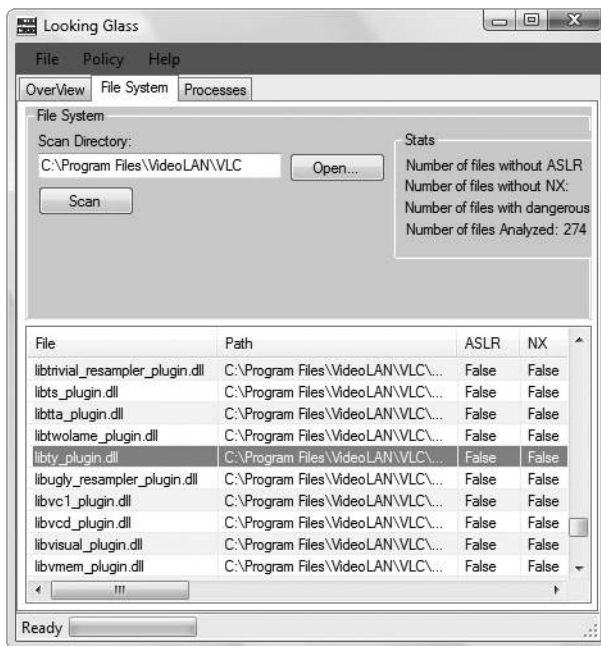
The output of the command shows that the system was indeed configured to use the OptIn operation mode of DEP, which explains why VLC doesn't use this mitigation technique: The process simply doesn't opt in to DEP.

There are different ways to opt a process in to DEP. For example, you could use the appropriate linker switch (`/NXCOMPAT`) at compile time, or you could use the `SetProcessDEPPolicy` API to allow an application to opt in to DEP programmatically.

To get an overview of the security-relevant compile-time options used by VLC, I scanned the executable files of the media player with LookingGlass (see Figure 2-9).¹¹

NOTE *In 2009, Microsoft released a tool called `BinScope Binary Analyzer`, which analyzes binaries for a wide variety of security protections with a very straightforward and easy-to-use interface.*¹²

LookingGlass showed that the linker switch for neither ASLR nor DEP was used to compile VLC.¹³ The Windows releases of VLC media player are built using the Cygwin¹⁴ environment, a set of utilities designed to provide the look and feel of Linux within the Windows operating system. Since the linker switches that I mentioned are supported only by Microsoft's Visual C++ 2005 SP1 and later (and thus are not supported by Cygwin), it isn't a big surprise that they aren't supported by VLC.



← Exploit mitigation techniques of Microsoft's Visual C++ 2005 SP1 and later:

- `/GS` for stack cookies/canaries
- `/DYNAMICBASE` for ASLR
- `/NXCOMPAT` for DEP/NX
- `/SAFESEH` for exception handler protection

Figure 2-9: LookingGlass scan result of VLC

See the following excerpt from the VLC build instructions:

```
[..]  
Building VLC from the source code  
=====  
[..]  
- natively on Windows, using cygwin (www.cygwin.com) with or without the POSIX  
emulation layer. This is the preferred way to compile vlc if you want to do it on  
Windows.  
[..]  
UNSUPPORTED METHODS  
-----  
[..]  
- natively on Windows, using Microsoft Visual Studio. This will not work.  
[..]
```

At the time of this writing, VLC didn't make use of any of the exploit mitigation techniques provided by Windows Vista or later releases. As a result, every bug in VLC under Windows is as easily exploited today as 20 years ago, when none of these security features were widely deployed or supported.

2.4 Lessons Learned

As a programmer:

- Never trust user input (this includes file data, network data, etc.).
- Never use unvalidated length or size values.
- Always make use of the exploit mitigation techniques offered by modern operating systems wherever possible. Under Windows, software has to be compiled with Microsoft's Visual C++ 2005 SP1 or later, and the appropriate compiler and linker options have to be used. In addition, Microsoft has released the *Enhanced Mitigation Experience Toolkit*,¹⁵ which allows specific mitigation techniques to be applied without recompilation.

As a user of media players:

- Don't ever trust media file extensions (see Section 2.5 below).

2.5 Addendum

Monday, October 20, 2008

Since the vulnerability was fixed and a new version of VLC is now available, I released a detailed security advisory on my website (Figure 2-10 shows the timeline).¹⁶ The bug was assigned CVE-2008-4654.

NOTE According to the documentation provided by MITRE,¹⁷ Common Vulnerabilities and Exposures Identifiers (also called CVE names, CVE numbers, CVE-IDs, and CVEs) are “unique, common identifiers for publicly known information security vulnerabilities.”

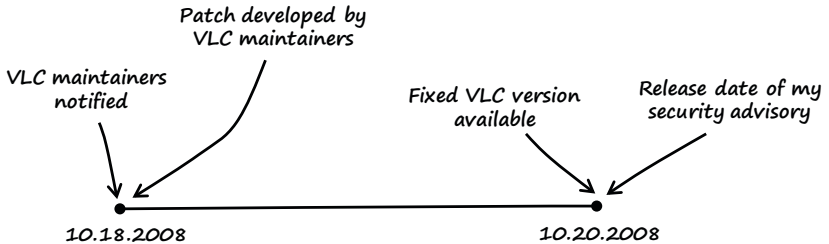


Figure 2-10: Timeline of the vulnerability

Monday, January 5, 2009

In reaction to the bug and my detailed advisory, I got a lot of mail with various questions from worried VLC users. There were two questions that I saw over and over:

I have never heard of the TiVo media format before. Why would I ever open such an obscure media file?

Am I secure if I don't open TiVo media files in VLC anymore?

These are valid questions, so I asked myself how I would normally learn about the format of a media file I downloaded via the Internet with no more information than the file extension. I could fire up a hex editor and have a look at the file header, but to be honest, I don't think ordinary people would go to the trouble. But are file extensions trustworthy? No, they aren't. The regular file extension for TiVo files is *.ty*. But what stops an attacker from changing the filename from *fun.ty* to *fun.avi*, *fun.mov*, *fun.mkv*, or whatever she likes? The file will still be opened and processed as a TiVo file by the media player, since VLC, like almost all media players, does not use file extensions to recognize the media format.

Notes

1. See Dick Grune and Cerie J.H. Jacobs, *Parsing Techniques: A Practical Guide*, 2nd ed. (New York: Springer Science+Business Media, 2008), 1.
2. The vulnerable source code version of VLC can be downloaded at <http://download.videolan.org/pub/videolan/vlc/0.9.4/vlc-0.9.4.tar.bz2>.

3. Immunity Debugger is a great Windows debugger based on OllyDbg. It comes with a nice GUI and a lot of extra features and plug-ins to support bug hunting and exploit development. It can be found at <http://www.immunityinc.com/products-immdbg.shtml>.
4. See David Litchfield, "Variations in Exploit Methods Between Linux and Windows," 2003, http://www.nccgroup.com/Libraries/Document_Downloads/Variations_in_Exploit_methods_between_Linux_and_Windows.sflb.ashx.
5. See <http://www.trapkit.de/books/bhd/>.
6. For more information on responsible, coordinated, and full disclosure as well as the commercial vulnerability market, consult Stefan Frei, Dominik Schatzmann, Bernhard Plattner, and Brian Trammel, "Modelling the Security Ecosystem—The Dynamics of (In)Security," 2009, <http://www.techzoom.net/publications/security-ecosystem/>.
7. The Git repository of VLC can be found at <http://git.videolan.org/>. The first fix issued for this bug can be downloaded from <http://git.videolan.org/?p=vlc.git;a=commitdiff;h=26d92b87bba99b5ea2e17b7ea39c462d65e9133>.
8. The fix for the subsequent VLC bug that I found can be downloaded from <http://git.videolan.org/?p=vlc.git;a=commitdiff;h=d859e6b9537af2d7326276f70de25a840f554dc3>.
9. To download Process Explorer, visit <http://technet.microsoft.com/en-en/sysinternals/bb896653/>.
10. See <http://blogs.technet.com/b/srd/archive/2009/06/12/understanding-dep-as-a-mitigation-technology-part-1.aspx>.
11. LookingGlass is a handy tool to scan a directory structure or the running processes to report which binaries do not make use of ASLR and NX. It can be found at <http://www.erratasec.com/lookingglass.html>.
12. To download BinScope Binary analyzer, visit <http://go.microsoft.com/?linkid=9678113>.
13. A good article on the exploit mitigation techniques introduced by Microsoft Visual C++ 2005 SP1 and later: Michael Howard, "Protecting Your Code with Visual C++ Defenses," *MSDN Magazine*, March 2008, <http://msdn.microsoft.com/en-us/magazine/cc337897.aspx>.
14. See <http://www.cygwin.com/>.
15. The Enhanced Mitigation Experience Toolkit is available at <http://blogs.technet.com/srd/archive/2010/09/02/enhanced-mitigation-experience-toolkit-emet-v2-0-0.aspx>.
16. My security advisory that describes the details of the VLC vulnerability can be found at <http://www.trapkit.de/advisories/TKADV2008-010.txt>.
17. See <http://cve.mitre.org/cve/identifiers/index.html>.