TRXCTF: molly

Contents

1 Overview **2**

2 Starting analysis **4**

3 The anti-debugging **5**

4 The approach **6**

5 The solution **6**

1 Overview

This challenge had 5 solves during the CTF and was marked as Insane difficult level by the authors. We are given two files, `molly.exe` and `molly_dll.dll`. The .exe tells us to put the key as an argument and it warns that no failures are allowed. Indeed, if we try to put a random key the `molly.exe` file gets substituted with a smaller file and it doesn't work anymore.

Figure 1: molly.exe executable

Figure 2: Failure

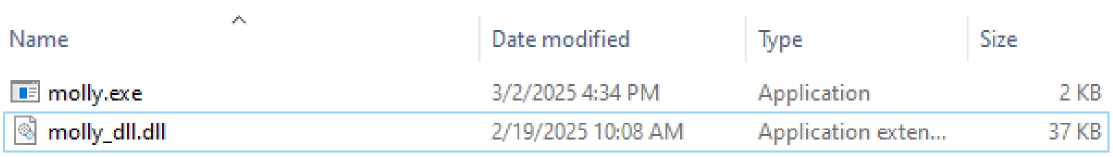

Figure 3: Shrinked file

Analyzing the `molly.exe` file with Ghidra it appears that some random bytes are at the entry point. The code is packed someway. If we try to run the program under a debugger it catches an Access Violation exception.

```
>>>>>>>>>>>>> Waiting for Debugger Extensions Gallery to Initialize completed, duration 0.828 seconds
   ----> Repository : UserExtensions, Enabled: true, Packages count: 0
   ----> Repository : LocalInstalled, Enabled: true, Packages count: 43

Microsoft (R) Windows Debugger Version 10.0.27793.1000 AMD64
Copyright (c) Microsoft Corporation. All rights reserved.

CommandLine: C:\Users\Ric\Desktop\writeup\molly\molly.exe

************* Path validation summary **************
Response                         Time (ms)     Location
Deferred                                       srv*
Symbol search path is: srv*
Executable search path is:
ModLoad: 00007ff6`281e0000 00007ff6`28269000   image00007ff6`281e0000
ModLoad: 00007ffd`73530000 00007ffd`73728000   ntdll.dll
ModLoad: 00007ffd`720a0000 00007ffd`72162000   C:\Windows\System32\KERNEL32.DLL
ModLoad: 00007ffd`71150000 00007ffd`7144f000   C:\Windows\System32\KERNELBASE.dll
ModLoad: 00007ffd`70f20000 00007ffd`71020000   C:\Windows\System32\ucrtbase.dll
ModLoad: 00007ffd`57230000 00007ffd`5724e000   C:\Windows\SYSTEM32\VCRUNTIME140.dll
ModLoad: 00007ffd`6bdc0000 00007ffd`6bdcc000   C:\Windows\SYSTEM32\VCRUNTIME140_1.dll
ModLoad: 00007ffd`24ab0000 00007ffd`24b3d000   C:\Windows\SYSTEM32\MSVCP140.dll
ModLoad: 00007ffd`59b40000 00007ffd`59b4e000   C:\Users\Ric\Desktop\writeup\molly\molly_dll.dll
(2154.1144): Break instruction exception - code 80000003 (first chance)
ntdll!LdrpDoDebuggerBreak+0x30:
00007ffd`736007a0 cc              int     3
0:000> g
(2154.1144): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
molly+0x6335c:
00007ff6`2824335c ??              ???
```

Figure 4: Access Violation Exception

```
                          **************************************************************
                          undefined entry()
                            assume GS_OFFSET = 0xff00000000
          undefined         AL:1           <RETURN>
                          entry                                       XREF[4]:     Entry P
                                                                                   1400865
      14006335c e3 6e          JRCXZ      LAB_1400633cc
      14006335e 18 d4          SBB        AH,DL
      140063360 e8 28 c7       CALL       SUB_1968cfa8d
                86 56
      140063365 a3 5e ba       MOV        [DAT_29a988b1e8a5ba5e],EAX
                a5 e8 b1
                88 a9 29
```

Figure 5: Ghidra entry point

2 Starting analysis

Since the DLLs main functions are executed before the executable entry point, the DLL is probably involved in unpacking the code and is probably also performing some anti-analysys. Putting the DLL under CAPA, something interesting shows up. The DLL calls some hardcoded syscalls. Since in windows syscalls are never called directly but always using the wrapper functions included in the API, this hints strongly to some sneaky code doing interesting stuff.

```
execute syscall (2 matches)
namespace    anti-analysis
author       @kulinacs, @mr-tz, mehunhoff@google.com
scope        basic block
references   https://github.com/j00ru/windows-syscalls
description  may be used to evade hooks or hinder analysis
basic block @ 0x180003090 in function 0x180003090
  or:
    and:
      mnemonic: syscall @ 0x180003098
      or:
        mnemonic: ret @ 0x18000309A
basic block @ 0x18000309B in function 0x18000309B
  or:
    and:
      mnemonic: syscall @ 0x1800030A3
      or:
        mnemonic: ret @ 0x1800030A5
```

Figure 6: CAPA pointing out the use of hardcoded syscalls in `molly_dll.dll`

Searching for the addresses pointed out by CAPA we can see that the binary has two hardocded syscalls, each of it is called in a different function. Searching for the hardcoded syscall number we can understand what API function is related to each syscall. Indeed, in this website all the known syscall number are listed and related to the appropriate API call. The two API call are: NtProtectVirtualMemory and NtSetInformationProcess. `NtProtectVirtualMemory` is used to change memory permission access. This could explain the Access Violation Exception.

```
                        NtProtectVirtualMemory




18000309b b8 50 00        MOV        EAX,0x50
          00 00
1800030a0 4c 8b d1        MOV        R10,RCX
1800030a3 0f 05           SYSCALL
1800030a5 c3              RET
```

Figure 7: NtProtectVirtualMemory syscall

```
                        NtSetInformationProcess

180003090 b8 1c 00         MOV         EAX,0x1c
          00 00
180003095 4c 8b d1         MOV         R10,RCX
180003098 0f 05            SYSCALL
18000309a c3               RET
```

Figure 8: NtSetInformationProcess syscall

3 The anti-debugging

If we follow the references to the function i renamed `NtProtectVirtualMemory`, we find some interesting functions. Among all stands up the function at address `180003f70`. Reversing this function (with the help of the debugger) we find out that what this function does is to search for the `.text` section in the .exe file to set the permissions of all the pages in it to `NO_ACCESS`.

The `NtSetInformationProcess` function it is also interesting. It is called in function at `180002f20`. Searching on the internet for this API call I found this interesting post. It explains how to use the `NtSetInformationProcess` to set up an hook that works both for syscalls and exceptions. It seems really related to what we have to work on. Thanks to the website, the debugger and the decompiler we can find that the callback points to the function at `1800030b0`. The callback simply jumps to the function at `1800031a8` if it is triggered by an exception or restores normal execution if it is triggered by a syscall. Following the flow of the callback we finish in the interesting function at `180003bd0`. This function changes the memory permission of the page in which the exception was called to writable and unpacks it using a simple byte mangling and xor with a key. Then it makes the page executable. So here what the DLL does: at first it makes all the `.text` permissions of the exe file with `NO_ACCESS`. Then sets the callback with the `NtSetInformationProcess`. When the .exe is executed, each time a new page of the `.text` is accessed, an Access Violation exception is triggered, so the callback is called, the page is unpacked and made executable, then the control flow is restored.

```
if (*(char *)(lVar7 + 0x18) != '\0') {
  local_res18 = 0x1000;
  local_res20 = param_2;
  pvVar3 = GetCurrentProcess();
  NtProtectVirtualMemory(pvVar3,&local_res20,&local_res18,(void *)0x4,local_res8);
  lVar7 = 0x1000;
  pbVar6 = param_2;
  i = 0;
  do {
    bVar5 = *pbVar6 >> 4 | *pbVar6 << 4;
    *pbVar6 = bVar5;
    *pbVar6 = bVar5 ^ s_forgivemefather_18000a060[i % 0xf];
    lVar7 = lVar7 + -1;
    pbVar6 = pbVar6 + 1;
    i = i + 1;
  } while (lVar7 != 0);
  local_res18 = 0x1000;
  local_res20 = param_2;
  pvVar3 = GetCurrentProcess();
  NtProtectVirtualMemory(pvVar3,&local_res20,&local_res18,(void *)0x20,local_res8);
  FUN_180004130(param_1,(longlong)param_2,'\0');
}
```

Figure 9: Snippet of the function at `180003bd0` which changes the memory permission access and unpacks the page

4 The approach

I wrote this Ghidra script to unpack the code of the .exe file so i could analyze it statically. But i wanted to analyze the code also attaching with a debugger. We can't put normal breakpoints since they work by substituting an instruction with the `int3` instruction and that would break the unpacking. There is also some mechanism that I could not fully understand that prevents the use of hardware breakpoints. I couldn't use the unpacked .exe as it is since the DLL would have tried to unpack it anyway breaking everything. So i came up with this idea. I use the unpacked binary as the main binary and i start it under the debugger. Then i run this windbg script. The script modifies the new memory protection access parameter of the function `NtProtectVirtualMemory`, that is called by the function that makes all the .text `NO_ACCESS`, to 0x40 (all accesses). In this way when we run the unpacked program under the debugger, the access exception is never called. I also patched the `you_wont_kill_my_allies` in the DLL with just a `ret` instruction. In this way I prevented the mechanism that destroys the .exe at each error.

5 The solution

Just searching for the text printed by the program we can find the function `140002bd0` where all the main program logic is. Analyzing it statically we find the string in the picture below. It seems that the program is a LUA interpreter.

```
puVar7 = (undefined4 *)
         "\nlocal function a(b)local c={}b=b:sub(5,-2)for d in string.gmatch(b,\"[^_]+\")do table.
         insert(c,d)end;return c end;local function e(f)if#f~=4 then return 1 end;local g=100+89-(
         84-608/((945+12+92+65-31)/57)-23)-60;local h=2370000/(12+93-95/(646/(85-3468/68)))/100-52
         -63-71;local i=149-(-15+8800/(6248/(125-(104-(17502/(72/12)+93)/35+36))))local j=(52808/(
         31-1800/(51-(95-448448/(147-83)/91)+42))-20)/66;if string.char(g,h,i,j)~=f then return 1
         end;return 0 end;local k=a(flag)local l=epic_gaming2(k[1])l=l+e(k[2])l=l+epic_gaming3(k[3
         ])l=l+epic_gaming1(k[4])return l\n"
;
```

Figure 10: String that hints toward LUA interpreter

What the string does is to check if the input is in the form of `TRX{xx_d3@r_xx_xx}` where the placeholder `xx` are verified by the functions `epic_gaming2`, `epic_gaming3`, `epic_gaming1`. Somewhere below we find the binding of these function to some C code.

```
else {
  pbVar13 = (byte *)FUN_140006a20();
  FUN_140006a30(pbVar13);
  lua_pushcclosure(pbVar13,FUN_140002920,"HAHAHHA1",0,0);
  lua_setfield(pbVar13,-0x2712,(undefined8 *)"epic_gaming1");
  lua_pushcclosure(pbVar13,FUN_1400029c0,"HAHAHHA2",0,0);
  lua_setfield(pbVar13,-0x2712,(undefined8 *)"epic_gaming2");
  lua_pushcclosure(pbVar13,&LAB_140002a80,"HAHAHHA3",0,0);
  lua_setfield(pbVar13,-0x2712,(undefined8 *)"epic_gaming3");
```

Figure 11: Binding of epic_gaming functions

In the picture below there are the tree epic_gaming function and they are pretty easy to understand. They just perform some checks on parts of the flag.

Decompile: FUN_140002920 - (molly_unpacked_2.exe)

```c

undefined8 FUN_140002920(byte *param_1)

{
  longlong lVar1;
  longlong lVar2;
  int iVar3;

  lVar1 = FUN_140005140(param_1,1,(ulonglong *)0x0);
  lVar2 = -1;
  do {
    lVar2 = lVar2 + 1;
  } while (*(char *)(lVar1 + lVar2) != '\0');
  if (lVar2 == 0x2c) {
    iVar3 = 0;
    while (lVar2 = (longlong)iVar3,
          (ushort)(((short)*(char *)(lVar2 + lVar1) +
                   (ushort)(byte)"0n0_d@y_w3_w1ll_m33t                        "[lVar2]) *
                  (short)iVar3) == USHORT_ARRAY_1400668d0[lVar2]) {
      iVar3 = iVar3 + 1;
      if (0x2b < iVar3) {
        FUN_1400041a0((longlong)param_1,0);
        return 1;
      }
    }
  }
  FUN_1400041a0((longlong)param_1,1);
  return 1;
}

```

Figure 12: epic_gaming1

Decompile: FUN_1400029c0 - (molly_unpacked_2.exe)

```c

void FUN_1400029c0(byte *param_1)

{
  byte bVar1;
  char *flag_piece;
  longlong lVar2;
  int iVar3;
  undefined8 *puVar4;
  uint uVar5;
  undefined auStack_48 [32];
  undefined4 local_28;
  undefined8 local_20;
  ulonglong local_18;

  local_18 = DAT_140080040 ^ (ulonglong)auStack_48;
  flag_piece = (char *)FUN_140005140(param_1,1,(ulonglong *)0x0);
  lVar2 = -1;
  do {
    lVar2 = lVar2 + 1;
  } while (flag_piece[lVar2] != '\0');
  if (lVar2 == 7) {
    local_28 = 0xefbeadde;
    local_20 = 0x8dd4bc8b8e9de8;
    puVar4 = &local_20;
    uVar5 = 0;
    do {
      bVar1 = *(byte *)(((longlong)flag_piece - (longlong)&local_20) + (longlong)puVar4);
      *(byte *)puVar4 = *(byte *)puVar4 ^ *(byte *)((longlong)&local_28 + (ulonglong)(uVar5 & 3));
      if ((int)(char)bVar1 != (uint)*(byte *)puVar4) goto LAB_140002a5b;
      uVar5 = uVar5 + 1;
      puVar4 = (undefined8 *)((longlong)puVar4 + 1);
    } while ((int)uVar5 < 7);
    iVar3 = 0;
  }
  else {
LAB_140002a5b:
    iVar3 = 1;
  }
```

Figure 13: epic_gaming2

Decompile: FUN_140002a80 - (molly_unpacked_2.exe)

```c

undefined8 FUN_140002a80(byte *param_1)

{
  char *pcVar1;
  longlong lVar2;

  pcVar1 = (char *)FUN_140005140(param_1,1,(ulonglong *)0x0);
  lVar2 = -1;
  do {
    lVar2 = lVar2 + 1;
  } while (pcVar1[lVar2] != '\0');
  if ((((lVar2 == 10) && ((ushort)((short)*pcVar1 << 8 | (short)pcVar1[1]) == 0x3070)) &&
      ((ushort)((short)pcVar1[2] << 8 | (short)pcVar1[3]) == 0x336e)) &&
     ((((ushort)((short)pcVar1[4] << 8 | (short)pcVar1[5]) == 0x3530 &&
       ((ushort)((short)pcVar1[6] << 8 | (short)pcVar1[7]) == 0x7572)) &&
      ((ushort)((short)pcVar1[8] << 8 | (short)pcVar1[9]) == 0x6333)))) {
    FUN_1400041a0((longlong)param_1,0);
    return 1;
  }
  FUN_1400041a0((longlong)param_1,1);
  return 1;
}

```

Figure 14: epic_gaming3

This is a simple python script that builds the flag ensuring the checks are passed. Running it we obtain the flag:

TRX{600dby3_d3@r_0p3n50urc3_AHR0cHM6Ly9wYXN0ZWJpbi5jb20vcmF3L1RZc0NLNEt4}

Figure 15: Solved!