# The Offenders

## Intro

The Offenders was a 500pts (still 500pts, 8 solves at the moment I'm writing this WU) reverse-engineering challenge at the FCSC 2021.

The executable file is a PE 64-bits.

The description of the challenge said the following:

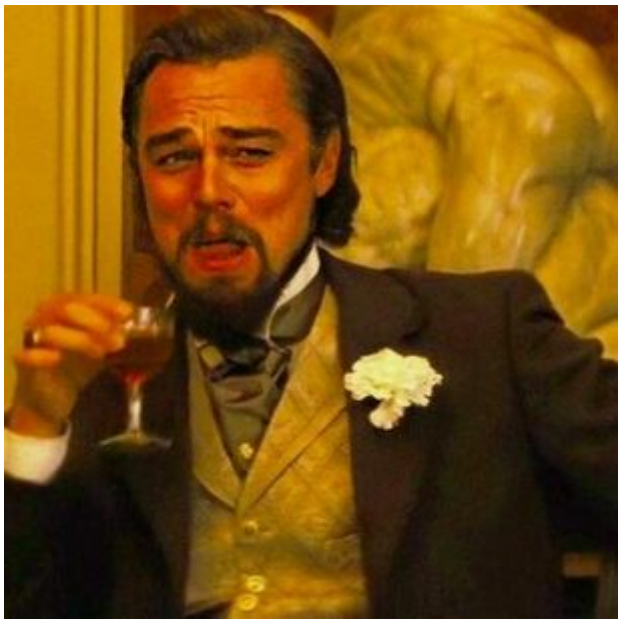
```
Our Intel Threat team came across this binary during a RetroHunt VirusTotal
searching for FCSC terms. They took a look at it and understood that the binary
expects to be executed by Windows Defender. A quick notepad reverse allowed us
the find a place starting with INPUTINPUT..., which seems to be the way the
give the program an argument (by patching it).
```

Two interesting thing are said in this description:

- We can't give the PE file an input, he reads the entry from some data section
- He expects to be executed by Windows Defender

Before I started this challenge I had no idea of how Windows Defender works, but I assumed the executable was expecting to be runned in some kind of virtual environment (with different files, ect ...).

Stringing the file gives us an interesting result: `Reminder: patch CVE-2021-1647`. To be totally honest, this string frightened me at first, because I thought they've included some exploit into the reverse challenge. I started reading some info about this CVE which is really interesting (heap buffer overflow in mpengine.dll that could allow remote code execution) but saw that no public Proof Of Concept was released, and I doubted they would release a 0day in the wild for a challenge. Either it was a lil' troll either I'm a little too stupid.



## First Look

As always I started by putting the file in IDA. Finding the main function isn't immediate, so the fastest thing to do is to check for inhabitual functions that are pretty likely to be used by the program (by example here we have `Process32FirstW`) and ask IDA to search references to this function. Luckily, there's only on reference to this function and it's the main we're looking for.

We can split the executable into five main parts:

## Process enumeration

```
hSnapshot = CreateToolhelp32Snapshot(2u, 0);
pe.dwSize = 568;
Process32FirstW(hSnapshot, &pe);
do
{
  if ( pe.szExeFile[0] == 121 )
  {
    sub_140001C60(&v62, 255i64, pe.szExeFile);
    sub_140001EC0(&v25, &v26);
  }
}
while ( Process32NextW(hSnapshot, &pe) );
```

To be totally honest, there's a lot of function in this program I don't understand, and it doesn't matter because it's not the point of the challenge. Here the function `sub_140001C60` plays the role of an `strncat` function, and the other `sub_` function, I have no idea.

But we can still get the main idea: the program take a snapshot of all the process states and iterate through running processes, and when the process name starts by char 121, 'y', it updates some variables.

## File enumeration

```
sub_1400060A0(&FileName, L"C:\\Program Files\\");
sub_140001C60(&FileName, 260i64, L"\\*");
hFindFile = FindFirstFileW(&FileName, &FindFileData);
do
{
  if ( (unsigned int)sub_140006068(FindFileData.cFileName, &unk_140016364)
    && (unsigned int)sub_140006068(FindFileData.cFileName, L"..")
    && FindFileData.dwFileAttributes & 0x10
    && FindFileData.cFileName[0] == 87 )
  {
    sub_140001C60(&v62, 255i64, FindFileData.cFileName);
  }
}
while ( FindNextFileW(hFindFile, &FindFileData) );
```

Same thing here, `sub_140006068` plays the role of `strcmp`. The program iterates through files in the directory `C:\Program Files`, and check two things for each file:

- The file attributes have the flag 0x10 enabled
- The filename starts with char code 87, 'W'

Looking at the WINAPI documentation, we learn that flag 0x10 indicates a directory, so the program is looking for directories starting by 'W'.

## Config file

```
hFile = CreateFileW(L"C:\\Mirc\\mirc.ini", 0x80000000, 1u, 0i64, 3u, 0x40000080u, 0i64);
ReadFile(hFile, &Buffer, 0x63u, &NumberOfBytesRead, 0i64);
for ( i = 0; i < NumberOfBytesRead; ++i )
{
  if ( *(&Buffer + (signed int)i) == 35 )
  {
    sub_140001D80(&v58, 7i64, L"%hs", &v60[i]);
    sub_140001C60(&v62, 255i64, &v58);
    break;
  }
}
CloseHandle(hFile);
```

Here the binary reads the file `C:\Mirc\mirc.ini`, note that the `0x40000080` flag says `FLAG_ATTRIBUTE_NORMAL | FILE_FLAG_DELETE_ON_CLOSE`, and that `FILE_FLAG_DELETE_ON_CLOSE` made my debugger `x64dbg` fails to read file for some reasons.

After reading the file, we parse his entry: we're looking for the first line starting by char code 35, '#'.

## Register enumeration

```
Class = 0;
memset(&v67, 0, 0x206ui64);
cchClass = 260;
cSubKeys = 0;
RegOpenKeyExW(HKEY_LOCAL_MACHINE, L"SOFTWARE", 0, 0x20019u, &phkResult);
RegQueryInfoKeyW(
  phkResult,
  &Class,
  &cchClass,
  0i64,
  &cSubKeys,
  &cbMaxSubKeyLen,
  &cbMaxClassLen,
  &cValues,
  &cbMaxValueNameLen,
  &cbMaxValueLen,
  &cbSecurityDescriptor,
  &ftLastWriteTime);
if ( cSubKeys )
{
  for ( dwIndex = 0; dwIndex < cSubKeys; ++dwIndex )
  {
    cchName = 255;
    if ( !RegEnumKeyExW(phkResult, dwIndex, &Name, &cchName, 0i64, 0i64, 0i64, &ftLastWriteTime) && Name == 102 )
      sub_140001C60(&v62, 255i64, &Name);
  }
}
RegCloseKey(phkResult);
```

This part works on the same idea: we enumerate registers keys in `HKEY_LOCAL_MACHINE\SOFTWARE` and look for keys starting by char code 102, 'f'.

## Entry check

```
v2 = sub_14000620C(&v62);
sub_140002200(&v63, &v62, (unsigned int)(2 * v2));
sub_140002310(&v63, &v61, 29i64);
sub_1400024A0(&v61, aFcscInputinput, 29i64);
sub_1400024A0(&v61, &unk_140021020, 29i64);
if ( !(unsigned int)sub_140006520("FCSC{", aFcscInputinput, 5i64)
  && !(unsigned int)sub_140006520("Reminder: patch CVE-2021-1647", &v61, 29i64) )
{
  hObject = CreateFileW(L"C:\\result.txt", 0x40000000u, 0, 0i64, 1u, 0x80u, 0i64);
  WriteFile(hObject, "Congratz", 8u, &NumberOfBytesWritten, 0i64);
  CloseHandle(hObject);
}
return 0i64;
```

So now we reach the check. A lot of totally weird dark magic jutsus are applied before the check, and i didn't reverse them because this wasn't necessary. Just note that `sub_1400024A0` XOR two bytes arrays. We just have to get the thing that our flag is XORED with and we'll be good !

As we can see, the flag is xored against `&v61` which was previously generated with the different environment variables (registries, files, ect...) the binary gathered. Once it's xored, it's compared with `Reminder: patch CVE-2021-1647`. All we have to do now is to recover `v61` just before it's getting xored with our flag.

# How 2 Offend The Offenders

There's still one problem: we need to recover the good `v61`, and since this variable depends on the environment, we can't just run it on windows with x64dbg and expect to get the flag. This is where this gets interesting. Knowing the binary was expected to get launched by Windows Defender, I watched the presentation of Alexei Bulazel (0xAlexei) gave during BlackHat USA 2018: Windows Offender: Reverse Engineering Windows Defender's Antivirus Emulator. This presentation is exactly we're looking for: how our binary is supposed to get emulated by Windows Defender.

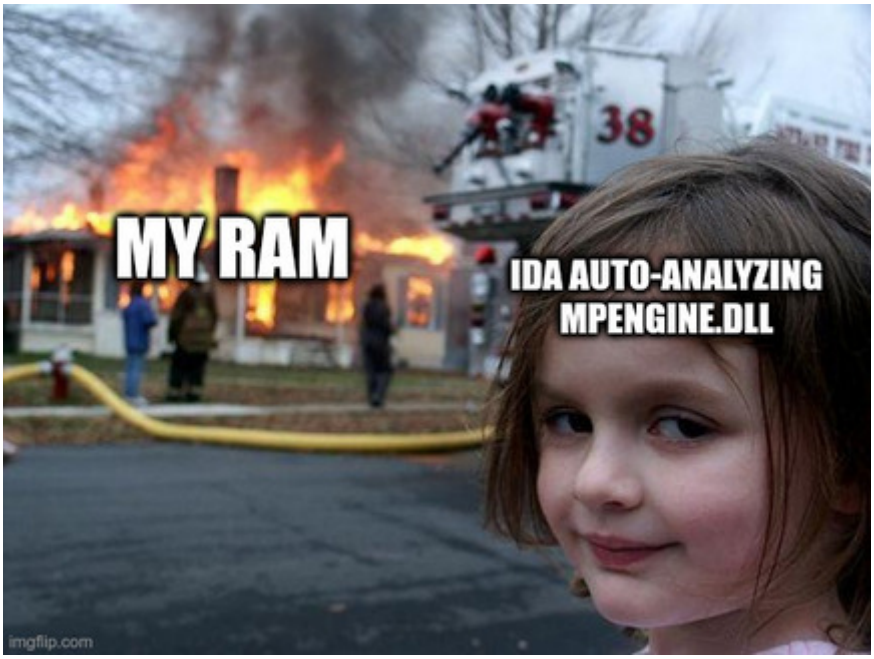
Windows Defender actually simulate a whole exploitation system: this fake OS have his own characteristics: his own files, his own registries, ect ... We just need to know those characteristics to get the flag. Luckily, Alexei released tools at , these tools are based on the `loadlibrary` project, which contains an mpclient (an ELF that uses loadlibrary to load mpengine.dll, the Windows Defenders emulator) executable ported on Linux.

Once I installed loadlibrary, I followed the instructions to download mpam-fe.exe and extracted the files from it using cabextract.

Sadly, there's a little issue: I couldn't find any PDB file online linked to this version of `mpengine.dll`, and you need to know the offsets of a few symbols to apply the loadlibrary's patch developed by 0xAlexei. This path is really interesting because it allows us to hook a function named `OutputDebugStringA`, which is useful because it allows us to communicate from the inside of `mpengine.dll`. More specifically: by hooking `OutputDebugStringA` we could leak every informations we need about the fake system emulated by `mpengine.dll`.

So, we had no PDB file, but 0xAlexei left instructions to still find the offsets we need: I had to disassemble `mpengine.dll` (which made my computer really suffer, this file is ***HUGE***) and:

- find a reference to the string "INVALID STRING", there's only one function that does it and it is `RVA_pe_read_string_ex`
- seach for sequence of bytes "BB 14 80 B2", just before this sequence we can see a function pointer, which is our `RVA_FP_OutputDebugStringA`
- disassemble the function `RVA_FP_OutputDebugStringA`, some functions are called in it but we can find a reference to `RVA_Parameters1` in it




Once we have our three offsets, plug them into the patch of 0xAlexei, build the whole thing and we're ready to go !

I built an executable that gave me the following:

- List of directories from `C:\Program Files`
- List of processes
- List of registry keys inside `HKLM\SOFTWARE`
- Content of `C:\Mirc\mirc.ini`

To give a small example, here's a program that prompt the content of `C:\Mirc\mirc.ini`:

```
#include <windows.h>

int main() {
  char Buffer[0x64];
  int NumberOfBytesRead;
  HANDLE hFile = CreateFileW(
      L"C:\\Mirc\\mirc.ini",
      GENERIC_READ,
      FILE_SHARE_READ,
      0,
      OPEN_EXISTING,
      FILE_ATTRIBUTE_NORMAL,
      0
  );
  OutputDebugStringA(Buffer);
  CloseHandle(hFile);
}
```

Which gives us:

```
[+] MpEngine.dll base at 0x5a100000
[+] Setting hooks and resolving offsets
[+] Parameters<1>::Parameters<1>        RVA: 0x5832e6   Address: 0x5a6832e6
[+] pe_read_string_ex:                  RVA: 0x58b0d2   Address: 0x5a68b0d2
[+] OutputDebugStringA FP:              RVA: 0x009f78   Address: 0x5a68a650
[+] OutputDebugStringA FP replaced:     0x566239c0
```

```
[+] Done setting hooks and resolving offsets!
main(): Scanning ../offender/search.exe...
EngineScanCallback(): Scanning input
[+] OutputDebugStringA(pe_vars_t * v = 0x0x58230f70)
[+] Params[1]:  0x22fe64
[+] OutputDebugStringA: "[chanfolder]
n0=#Blabla
n1=#End
"
```

I just had to do that for everything we need and I came up with this list:

- Process: yahoomessenger.exe
- Directory inside `C:\\Program Files`: WebMoney
- Registry key inside `HKLM\SOFTWARE`: fuckyou
- And the constant of mirc.ini that I showed earlier

Knowing all that, I created the directory, the key and the config file I needed inside a Windows 7 VM. For the process, I just created an executable named yahoomessenger.exe with a `while (1) { sleep(100000); }` inside. I deleted the keys & file that could parasite the result (it's a VM after all, who cares). I then runned the binary inside x64dbg, got the generated string xored it and:

`FCSC{HelloFromEmulatedWorld!}`