Flare-On Challenge 8 Solution
By Paul Tarter, Raymond Leong, and Chuong Dong

Challenge 9: evil
**Challenge Prompt**

Mandiant’s unofficial motto is “find evil and solve crime”. Well here is evil but forget crime, solve challenge. Listen kid, RFCs are for fools, but for you we’ll make an exception :)

The challenge has 3 false flags:
- !t_$uRe_W0u1d_B3_n1ce_huh!@flare-on.com
- 1s_thi$_mY_f1aG@flare-on.com
- N3ver_G0nNa_g1ve_y0u_Up@flare-on.com

**Solution**

Over the last few years, the FLARE team has analyzed an increasing variety of stealthy passive backdoors that share a common feature of listening for and validating a specific network byte sequence before accepting tasking from a C2 server. For example, KEYPLUG.Passive (KCP listener), SADFLOWER (HTTP listener), CROSSWALK.BIN (TCP listener), and LOWKEY (UDP listener).

This challenge is a UDP passive backdoor that also validates traffic through a network bit, the evil bit, as defined in RFC 3514 (IETF, 2003). As to making exceptions for one attempting this challenge, no exception is made on difficulty, instead an exception is thrown any time a Windows API call is made (except for one call too CloseHandle… oops). The challenge also incorporates anti-analysis techniques seen by the FLARE team such as Vectored Exception Handler (VEH) hooking, detours hooking, and a healthy mix of anti-disassembly, anti-debugging, and anti-virtualization techniques.

As with any malware sample, analysis can be tackled in many different approaches. The approach of this solution is to provide the reader with an understanding of the core concepts implemented within the challenge. After covering core concepts, an approach for solving the challenge will be covered walking the reader through a start to finish analysis.

**Core Architecture Concepts**

String Obfuscation

A common method seen more frequently with string obfuscation is when a malware author uses C++ templates with the advantage of constexpr, which was introduced in C++11. The expression declares that it is possible to evaluate the value of the function or variable at compile time. This means when a string wants to be obfuscated it can have its own key and it offers the developer an option to have a templated obfuscation. This creates many nasty inline deobfuscators throughout the program. Xor obfuscation can be easier to spot at times or picked up by analysis tools like CAPA (FireEye, 2021). To combat simple detection but still keep string obfuscation to a fairly simple single-byte-xor the technique of obfuscating the xor operation was chosen and is shown in Figure 1:

```
constexpr char obfuscatedXor(char a, char b) {
    return (~((((~a) & 0xFF) | b) & (a | ((~b) & 0xFF))) & 0xFF) &
            (((~((~a) & 0xFF) | b) & (a | ((~b) & 0xFF))) & 0xFF) |
            (char)(0xB7F748F3));
}
```

As with any malware sample, analysis can be tackled in many different approaches. The approach of this solution is to provide the reader with an understanding of the core concepts implemented within the challenge. After covering core concepts, an approach for solving the challenge will be covered walking the reader through a start to finish analysis.

**Core Architecture Concepts**

String Obfuscation

A common method seen more frequently with string obfuscation is when a malware author uses C++ templates with the advantage of constexpr, which was introduced in C++11. The expression declares that it is possible to evaluate the value of the function or variable at compile time. This means when a string wants to be obfuscated it can have its own key and it offers the developer an option to have a templated obfuscation. This creates many nasty inline deobfuscators throughout the program. Xor obfuscation can be easier to spot at times or picked up by analysis tools like CAPA (FireEye, 2021). To combat simple detection but still keep string obfuscation to a fairly simple single-byte-xor the technique of obfuscating the xor operation was chosen and is shown in Figure 1:

```
constexpr char obfuscatedXor(char a, char b) {
    return (~((((~a) & 0xFF) | b) & (a | ((~b) & 0xFF))) & 0xFF) &
            (((~((~a) & 0xFF) | b) & (a | ((~b) & 0xFF))) & 0xFF) |
            (char)(0xB7F748F3));
}
```
Challenge 9: evil

Flare - On 8

Figure 1: Obfuscated XOR Operation

```cpp
if (is_encrypted())
{
    for (std::size_t i = N + 1; i != 0; i--)
        data[i] = obfuscatedXor(data[i], data[i - 1]) - 3;
    m_data[0] = obfuscatedXor(m_data[0], KEY) - 3;
}
```

Figure 2: XOR deobfuscation

```cpp
int func(void)
{
    return 3;
}

int gi = func();

int main()
{
    return gi;
}
```

Figure 3: XOR Deobfuscation Decompilation

CRT Initialization functions

Malware authors can abuse CRT Initialization to have code run before main(). A global variable can be assigned by the result of a function as shown in Figure 4: Initializing a Global Object. This challenge utilizes this functionality more than abuse it to initialize strings, maps and implement hooking. When analyzing a sample and it appears that code is being run before main, one spot to check is the CRT Initializers. (Microsoft, 2016)

Figure 4: Initializing a Global Object

API Hooking: Vectored Exception Handling

Analyzing the challenge, one will quickly see that API calls are taking place but upon static analysis there aren’t any calls to an API call as expected. There is a mix of anti-disassembly taking place in the challenge and
ultimately API hooking via VEH. VEH is built to handle exceptions prior to Structured Exception Handling (SEH). SEH is stack based and only operated on using keywords such as try or catch. VEH is registered using an API call AddVectoredExceptionHandler. VEH handlers are not tied to a specific function, nor are they tied to a stack frame. Adding a handler adds the handler to a list of exception handlers which will all run prior to SEH. In this challenge it is not as simple as cross referencing the input to AddVectoredExceptionHandler and finding the handler, this challenge resolves all APIs itself. Knowing that it takes place before main allows one to look at pre-main execution locations, in this challenge CRT initializers is where to look. Within the VEH handler for this challenge API resolution takes place and the .text section code is changed to call the appropriate API and then execution is continued. Further details are provided in Challenge Walkthrough.

API Hooking: Detours with PolyHook2

API Hooking utilizing Detours is the process of replacing the first couple bytes of a function with a jump to a new address where one can implement pre and post execution of an API before returning to the original executing code. This challenge used PolyHook2 written by FLARE reverse engineer Stephen Eckels (Eckels, 2021). PolyHook2 implements multiple hooking capabilities, it is a very robust and well written hooking engine. The library is statically linked with this challenge, this challenge’s architecture chose not to use VEH hooking from PolyHook2, mainly to recreate similarity with samples seen in the wild and to incorporate anti-disassembly techniques inline with the exception handling.

Anti-Disassembly

There are many forms of anti-disassembly, the purpose is to create logic problems that a disassembler or decompiler don’t handle well and thus allows the disassembly process to break. This challenge utilizes nine separate anti-disassembly instruction sets that are in line with VEH. Each set is the same size, seven bytes. The first part of a set is setup and creation of an exception; following the exception creation bytes are anti disassembly bytes. These bytes are patched by the exception handler and execution is continued.

One example is provided below as seen in Figure 5: Anti-Disassembly null deference followed by junk jumps, this example first sets up an access violation by zeroing EAX and then dereferencing a null pointer. Following the exception, a jz/jnz code jumps, which places EIP in the middle of real code. The disassembly doesn’t know how to handle this situation and the following bytes are not analyzed and code analysis is picked up later or sometimes not at all. The analysis at times after anti-disassembly cannot be reliable and it is best to NOP out the offensive code. Once this is performed as seen in Figure 6 one can perform analysis more reliably. The problem persists that the API calls are still created by an exception and thus parameters are placed on the stack, but the stack is never cleaned up. This leads to other analysis hurdles. When walking through the challenge one will notice that there isn’t a solution given to statically solve this issue for analysis, by doing so, often the decompiler is not reliable and the disassembly window must be used.

This challenge consists of an HTML file with JavaScript code named admin.html, and an img directory containing the graphics it uses. Launching the admin.html file with a web browser displays a prompt asking for a username and password. If you enter both it will enable the “Check Credentials” button. If you click this button it determines if you have entered the correct credentials or not. Error! Reference source not found. Error! Reference source not found. shows an example of the credential input screen where incorrect credentials have been entered.
Challenge 9: evil | Flare-On 8

Figure 5: Anti-Disassembly null deference followed by junk jumps

```
.text:00406514         loc_406514:
.text:00406514 E8 89 41 04 85  call near ptr 8544A6A2h
```

Figure 6: Anti-Disassembly removed

Anti-Debugging Techniques

The anti-debugging techniques used throughout this challenge are fairly common and most are handled by a plugin for x64dbg, ScyllaHide as seen in Figure 7 shows many of the options for configuring ScyllaHide (ScyllaHide, 2021). Some efforts were added to further obfuscate anti-debugging techniques, below is a list of anti-debugging techniques used:

- IsDebuggerPresent
  - Determined by checking PEB.BeingDebugged (offset 0x02) to be set.
- IsRemoteDebuggerPresent
  - Calls CheckRemoteDebuggerPresent on current process and checks if the return has a debug port.
- IsNtGlobalFlag
  - Checks the PEB.NtGlobalFlag (offset 0x68) for the following flags described in Table 1: NtGlobalFlags.

```
Flag                  Value
FLG_HEAP_ENABLE_TAIL_CHECK   0x10
FLG_HEAP_ENABLE_FREE_CHECK   0x20
FLG_HEAP_VALIDATE_PARAMETERS 0x40
Total                        0x70
```

Table 1: NtGlobalFlags

- IsSeDebugPrivsEnabled
This sample doesn’t enable DebugPrivils so if they are it is a red flag. Another test that is better for this kind of test, but not implemented is to check the parent process and see if that process has DebugPrivils enabled.

- **IsHardwareBreakpointPresent**
  - This checks if any of the hardware breakpoints are set by checking the CONTEXT record for the running thread.

- **IsTooSlow**
  - This check is a simple timing check that does a simple back to back call to GetTickCount and compares the delta and looks for a threshold of anything over two seconds constitutes debugging.

These debug checks are run periodically and randomly throughout the challenge. A random test will run randomly between zero and ten seconds. This is to bypass a common practice of letting all anti-debug logic to run and then attach to the process for debugging. There are places during the challenge that debug checks are called randomly. Not only is this a hassle, it creates the problem that if one wants to NOP out the anti-debugging and just handles the creation of debugging object, the program will crash when the object is trying to be referenced later. If the anti-debug thread is killed it will restart itself. Killing both threads is an option. All anti-debug checks will call ExitProcess if debugging is detected, because of this, modifying ExitProcess to return immediately is an option too.

The constructor for the AntiDebug class has added functionality that is only called once. Two of these fall under anti-debug and two under anti-virtualization. The two anti-debug features are as follows:

- **PatchDbgBreakPoint**
  - DbgBreakPoint is resolved and the first byte of the function is replaces with a ret instruction (0xC3).

- **PatchDbgUiRemoteBreakIn**
  - Patching DbgUiRemoteBreakIn goes a step further and inserts shellcode that will call TerminateProcess when this function is called.

Anti-Virtualization Techniques

The anti-virtualization techniques used in this challenge are meant to detect running in either VmWare or VirtualBox. The techniques are standard and well covered over the years.

VmWare Detection starts with a common VmWare detection method of getting the memory size by making using the in operation from assembly. There are some magic values that are associated with this method and therefore are obfuscated by adding two values together to create the desired value. If running outside of a VM the in call is
a privileged command and an exception will occur. Because this exception the memory will never be retrieved and still be zero. If running in a VM the in call will be successful and memory will be retrieved.

The next VmWare detection is a bit more complex and requires creating a code segment less than 4GB using the API call ZwSetLdtEntries Then transfer into it. With an exception handler in place, return out of it. If you are running natively, then EIP will still be within the CS: limits and ESP will remain unchanged; if you’re running in emulation mode, EIP will be above the CS: limit and then return address will have been popped. (eEye Research, 2006).

VirtualBox detection is done by performing a WMI query

```
SELECT * FROM Win32_PnPEntity
```

The DeviceId fields are parsed and a SHA1 hash is created, the device searched for is PCI\VEN_80EE&DEV_CAFE.

**Challenge Walkthrough**

**Triage and Basic Analysis**

This challenge initially doesn’t provide much information when performing a basic triage. Some first steps would be analyzing with a PE Viewer such as CFF Explorer, this only yields some useful information but not too much. The sample does not appear packed, but the import directory doesn’t show much of interest. The challenge only imports a handful of functions from kernel32.dll as seen in Figure 8. Scanning through the list of functions nothing is blatantly obvious to what this program does and lends to believing that the challenge resolves its own API calls.
Observing strings from the challenge yields a very large output as well as some fun trolling. The strings output verifies that this program is a C++ program, it appears to have a large amount of code, and most likely a statically linked library. A well written piece of malware usually doesn't yield many useful strings and in this case the use of PolyHook2 added a lot of strings. Some more blatantly obvious strings from the library were sanitized out for the challenge. Using rank_strings from FLARE’s StringSifter project (FLARE, 2021) allows one to rank the strings and put what is believed to be important strings towards the top of the listing. An example usage for this challenge can be seen in Figure 8 and a little of the sample output is seen in Figure 10: rank_strings output

```bash
> flarestrings evil.exe | rank_strings > strings.txt
```

**Figure 9: StringSifter Usage**

<table>
<thead>
<tr>
<th>h.MS</th>
<th>b.aI</th>
<th>i.SM</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="t_$uRe_W0u1d_B3_n1ce_huh!@flare-on.com." /></td>
<td>L.bN</td>
<td>L.KP</td>
</tr>
<tr>
<td>P.LA</td>
<td>Q.IM</td>
<td>-c.kH</td>
</tr>
<tr>
<td>Bkernel32.dll</td>
<td>vcvtsi2ssl</td>
<td>extractps</td>
</tr>
<tr>
<td>0.Ao</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 10: rank_strings output**

Running the sample from the command line with an elevated token leads to the challenge immediately exiting. Viewing the output from Procmon, the challenge simply starts and quickly exits after Loading all the images. This kind of activity would make one assume that the program quickly exited out in main and some sort of environment needs to be present or arguments must be passed. Further investigation would be required with static binary analysis. One note that can be taken from ProcMon output is a comparison of a program that was built named nothing.exe and is statically linked. Nothing.exe has about 65 functions imported from kernel32.dll, but when looking at Procmon differences there are additional libraries being loaded that aren't expected when running evil.exe. Observe the difference between Figure 11 and Figure 12 and one can notice the extra libraries, one specifically advapi32.dll. This shows that there is something more than a basic initialization that is causing additional libraries to be loaded.
Challenge 9: evil | Flare-On 8

Analysis using disassembly and Decompilation was performed using IDA Pro 7.6. Due to Ghidra’s release decompilation will be used when helpful throughout the challenge walkthrough. When first observing the challenge in IDA pro it is quickly obvious that there is anti-disassembly at play. While there is a call early on and it is valid, it is to loc_4023D0. Decompilation isn’t available in main or loc_4023D0 because of anti-disassembly.
Exception handling hooking or some very poor programming takes place very quickly in the program, the first instance of VEH hooking is in the code block `loc_4023D0` and can be found at `0x00402460`. Observing the SEH handlers shows nothing that could handle exception hooking, this would make one assume that VEH hooking is taking place. Observing further above the exception generation code two values are stored in ECX and EDX and a pointer is pushed on the stack prior to the exception. A well commented markup of this can be seen in Figure 13: VEH Hooking Setup. A quick look at google yields no results for the hex values being placed in ECX or EDX. A couple avenues of approach from here are to start fixing the anti-disassembly or investigate the null pointer dereference.

```
00402437   mov    dword ptr [ebp-10h], 246132h   ; temporary storage
0040243E   mov    dword ptr [ebp-18h], 66FFF672h ; temporary storage
00402445   add    eax, 3039h                     ; unrelated
0040244A   mov    dword_6CFBD8, eax              ; unrelated
0040244F   lea    eax, [ebp-38h]                 ; stack pointer stored in EAX
00402452   mov    [ebp-14h], eax                 ; stack pointer stored on stack
00402455   push   dword ptr [ebp-14h]            ; push stack pointer
00402458   mov    edx, [ebp-10h]                 ; move temporary hex value to EDX
0040245B   mov    ecx, [ebp-18h]                 ; move temporary hex value to ECX
0040245E   xor    eax, eax                       ; zero EAX
00402460   mov    eax, [eax]                     ; dereference NULL pointer 00402460
```

Figure 13: VEH Hooking Setup

First a quick win is to manually NOP out the exception and fix the anti-disassembly and see if a how helpful it is. To perform this in IDA synchronizing the Hex Edit window with the disassembly window and having them side by side makes life a little easier as seen in Figure 14. Pressing of F2 allows byte patching, followed by another press of F2 to save changes, patched bytes can be seen in Figure 15. This opens a lot more analysis, and even some calls into other functions. There are more API calls that are taking place and because of this there is more patching to take place.

Figure 14: Synchronized IDA View-A and HexView-1
Figure 15: NOP'd anti-disassembly and exception generation code

Placing this sample in a debugger could at least verify that VEH hooking is taking place. One step over of the exception and the challenge is sitting at the entry point of GetSystemTime. This verifies that VEH hooking is taking place. Restarting the program and setting a break on kernel32!AddVectoredExceptionHandler could possibly allow one to see the handler getting added, this approach, doesn’t seem to work. Returning to static analysis, might be helpful.

It is known that the challenge has some pre-main execution taking place. As described earlier, this challenge takes advantage of CRT Initialization functions. CRT Initialization takes place by observing the list that is parsed with __scrt_common_main_seh!_initterm. The first parameter will take the analyst to a list of functions that are called as CRT Initializers as seen in Figure 16.

Figure 16: CRT Initializers

A quick triage of functions starting from the top starts with normal CPP initializers. At some point going through the list functionality by the challenge is added in. One other approach for this is to start at the end of the list. If one starts analyzing initializers there and finds a CPP function instead of something useful the analyst didn’t waste
time looking through the whole list. Feel free to analyze as many of these at this point. For sake of walkthrough the analysis will start at the end and not continue up the list.

Starting with the last function in the list sub_402150 gives a quick win. The function prototype for AddVectoredExceptionHandler is as follows.

```c
PVOID AddVectoredExceptionHandler(
    ULONG First,
    PVECTORED_EXCEPTION_HANDLER Handler
);
```

This looks like a good match to the prototype, setting a breakpoint on the call to this function reveals a call to RtlAddVectoredExceptionHandler, which would explain the lack of breakpoint working when breaking on kernel32!AddVectoredExceptionHandler.

This confirmation allows the analyst to mark dword_D16E4 as RtlAddVectoredExceptionHandler and mark sub_406AD0 as exception_handler.

The prototype for a PVECTORED_EXCEPTION_HANDLER is

```c
LONG PvectoredExceptionHandler(
    _EXCEPTION_POINTERS *ExceptionInfo
)
```

Marking this in the handler within IDA will make the handler a lot easier to understand. Decompilation is available within this function as no anti-disassembly takes place. A marked up Decompilation is provided with Figure 17: exception_handler decompilation and further explanation is followed. Some of this is determined by static analysis and some through dynamic analysis.

```
```

Figure 17: exception_handler decompilation

Line 13–20 is a common pattern seen for creating a static variable. A static variable is one that can be initialized or assigned and hold its value through follow on calls. It is implemented by the Microsoft compiler with a global variable for the static variable and an initialization variable. Stepping through dynamically and stepping over Line
18 shows that this function resolves an address using two values. The function resolved is `VirtualProtect`. Line 22 resolves the API to the values placed in ECX and EDX prior to exception generation. The function then changes the address where EIP is to RWX which allows a patch to be written on line 27 at EIP+3. The bytes that are overwritten are FF D0 or `call EAX`. The handler then increments EIP by 3 in the context record and continues execution. This technique allows the code after the execution to be anti-disassembly bytes, but then be overwritten by the handler and be functional code when the exception continues execution with the return value `EXCEPTION_CONTINUE_EXECUTION`.

API Hashes

Finding the exception handler is a huge step forward in analyzing this program, even though at this point there is no knowledge to what the program does. From here it is possible to determine how the values correlate to being resolved to an API. A common method for this approach is creating some sort of hash and then searching through the exported names of a DLL and find a hash that matches.

Analyzing `GetProcAddressEx (0x004054B0)` starts by first by searching a `std::map` for an entry using the first parameter to `GetProcAddressEx` as a key in the map. The Value in the map is a library module that has been loaded. If the key doesn’t exist in the `dll_map_modules`, then the function will search another `std::map` with the same id and find a string that has been deobfuscated as explained in Core Architecture Concepts above. This functionality allows the program to cache its loaded libraries and only call `LoadLibrary` on them once. Digging deep into this portion of code will lead one to more functions that are in the CRT Initialization section that initialize these maps and deobfuscate the strings into them.

Once a module is acquired the exports are parsed and a hash is performed on the export to check against the second parameter to `GetProcAddressEx`. The hashing algorithm isn’t too hard to find even without marking up the IDB with all the PE parsing. The hashing algorithm is shown in Figure 18: API Hashing Algorithm and can easily be rewritten in python. Doing so allows an analyst to create a header file of all API hashes and import it to IDA for marking up the IDB. Example code is provided in Figure 26: `generate_hashes.py`.

![Figure 18: API Hashing Algorithm](image)

Using the enums that `generate_hashes.py` creates, one can import these hashes into IDA pro by either pasting them into a new Local Type from the Local Types window subview, alternatively the header files can be import through “File->Load File->Parse C Header File...” Once imported right click and choose “synchronize to IDB”. Once performed return to address 0x0040243E and resolve the enum 0x66FFF6725 to `GetSystemTime`. Now throughout the rest of the analysis APIs can be resolved.
Anti-Disassembly

One could manually patch each instance, although as manual patching gets tedious and possibly the analyst notices that there are repeats of anti-disassembly bytes. The option for creating a script to patch out all instances of anti-disassembly becomes an option. A script Figure 27: nop_anti_dis.py is provided to show functionality to NOP out anti-disassembly. The script provides the byte sequences to search for and NOP, they are also provided in Table 2. Note that this script is meant to be run on a binary, sadly this means one would have to start the IDB fresh. Another option is to use the IDA API to create all the patches.

| 33 C0 F7 F0 EB 00 Eb | 33 C0 8B 00 74 03 75 | 33 C0 F7 F0 33 C0 74 |
| 33 C0 8B 00 EB FF E8 | 33 FF F4 33 C0 74 | 33 C0 F7 F0 E8 FF D2 |
| 33 F6 F7 F6 E8 FF D2 | 33 FF F7 F7 33 C0 74 | 33 C0 F7 F0 5B 5D C3 |

Table 2: Anti-Disassembly Exception Generating Byte Sequences

AntiDebug Class

Scrolling through main and marking up the API calls reveals that main creates two objects that appear to be C++ classes, some of the constructor code appears to be inlined with main. One snippet of code around address 0x004064E9 is a mutex that is created, one can follow the xref to the mutex name and decode the string or dynamically view it later; the mutex appears to be another troll _tsTHl$_mY_f1aG@flare-on.com, but the mutex handle is saved into an object showing that it might be used for synchronization.

After initializing the objects, four threads are started, after the threads are started there is a WaitForMultipleObjects called, this call is only waiting on two objects. Cleanup at the end of main is helpful for marking up objects or learning what the program might have been using. A very big help is that closesocket is called twice at the end of main, followed by WSACleanup. This is the first time that networking is determined to be a factor in this challenge.

One thing that can be helpful at this point is take note that new is called three time. Each of these calls look related to C++ objects, this shows the size of the objects and allows one to start modeling the structure as
reversing continues. The first object is of size 0x18 and passed into a constructor \texttt{sub_4023D0}. If one was to take a route of dynamically debugging before static analysis it would be discovered that stepping over \texttt{sub_4023D0} leads to the program terminating. After the 18-byte object is created another object of size 0x1CC is created followed by a final object that is 0x08 bytes.

Function \texttt{sub_4023D0} appears to handle anti-debugging routines that were explained in core architecture. When the object is returned it is stored in the next object at offset 0x1AC. Skipping over this constructor to avoid anti-debugging and anti-virtualization and returning a NULL pointer for the anti-debug class object causes the program to exit. To be able to debug this program better, one must understand the anti-debugging class object.

The anti-debug and anti-virtualization object constructor start at \texttt{sub_4023D0} and will further be named \texttt{AntiDebug\_constructor} and the object will be named \texttt{AntiDebug}. \texttt{AntiDebug} appears to implement \texttt{rand()} and \texttt{srand()} manually instead of calling the API, this is due to the functions not being exported functions and could not be resolved through VEH hooking. The hard coded constant 0x41C64E6D is a common constant used for \texttt{srand()}.

Following \texttt{rand()} and \texttt{srand()}, \texttt{DbgBreakPoint} can be seen getting patched to return when called. This is shown in Figure 20: Patch Out \texttt{DbgBreakPoint}. Notice the 0xC3 at address 0x4024CA, this is the patch being written to \texttt{DbgBreakPoint}. Notice that \texttt{DbgBreakPoint} is resolved using \texttt{GetProcAddressEx} to determine the patching location.

![Figure 20: Patch Out DbgBreakPoint](image)

\texttt{DbgUiRemoteBreakin} is patched in a similar fashion except shellcode to call \texttt{TerminateProcess} is called instead of a simple ret instruction. The code can be seen in Figure 21: Patch \texttt{DbgUiRemoteBreakin} and the shellcode is observed to start building at 0x402521. The shellcode can be read as:
Figure 21: Patch DbgUIRemoteBreakin

The next section of code as seen in Figure 22 are two function calls, one is for detecting VmWare as described in core architecture, as well as detect VirtualBox. Some constants are added together to form signatures, this is done to evade detection tools. As both functions will exit if virtualization is detected, but nothing is passed into the functions such as the AntiDebug object, it is safe to say, NOP'ing out this code is can be performed to avoid anti-virtualization.

A std::map is populated with many debug functions starting at address 0x004025BB. After populating the map, the functions are looped through and each are called. This isn’t the last time that the debug functions will be referenced, therefore returning NULL from this function is not an option. Multiple options can be taken to bypass the debugging, one is to modify your debugger so ExitProcess returns happily, this can be a pain from the exception handling, but works. Another way is that if there aren’t any functions in the map then functions can’t be called. Sadly, this program when randomly calling functions doesn’t check for the map to be empty so a crash will occur. The easiest work around is let IsDebuggerPresent get added to the map and then either patch the PEB, let ScyllaHide do it for you or the most work is to patch the function that implements IsDebuggerPresent to ret 4.
Figure 22: Debug std::map initialization and calling debug functions

The large 0x1CC object has inline construction and doesn't give much information away currently. The trolling mutex is created and stored in the large object. Next four threads are created, they are separated into two different functionalities. The first of the two threads grab's a random value modulus of debug_functions_map.size and calls the function that corresponds to that value. Therefore, allowing one function to reside is needed because no check on the map being empty occurs which would mean the zeroth function would always be called. At address 0x0040274C a thread is created and then waited on, this is to allow the program to run without too much delay while debug functions are ran in the background. The debug thread runs a random debug function anywhere from zero to ten seconds.

The next thread found at 0x00402D5 with a good name of restarter thread is passed in the AntiDebug object calls WaitForSingleObject on EAX+8, which happens to be the debug thread previously started. If the debug thread is killed this current thread will CloseHandle on the trolling mutex, which appears to be used to check if the debug thread has been started, and then restarts the debug thread.

At this point the AntiDebug structure can be well defined as

<table>
<thead>
<tr>
<th>Address</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
<td>debug_object</td>
<td>struc ; (sizeof=0x18, mappedto_96)</td>
</tr>
<tr>
<td>00000000</td>
<td>unknown</td>
<td>dd ?</td>
</tr>
<tr>
<td>00000004</td>
<td>mutex</td>
<td>dd ?</td>
</tr>
</tbody>
</table>
The following code shifts into more core functionality of this challenge. Before the final two threads are created some initialization takes place for the larger object which will be forward referred to `backdoor_object`. If one has been still having the program exit when run even with all anti-debugging patches the following snippet of code might be why.

```
.text:004065E2                 cmp     [ebp+argc], 2
.text:004065E6                 mov     dword ptr [ebx+1C0h], 0
.text:004065F0                 mov     [ebx+1C8h], eax
.text:004065F6                 mov     [ebp+anti_debug_object], ebx
.text:004065F9                 jnz     loc_406786
.text:004065FF                 mov     eax, [ebp+argv]
.text:00406602                 sub     esp, 0Ch
.text:00406605                 mov     ecx, ebx
.text:00406607                 push    dword ptr [eax+4] ; pExceptionObject
.text:0040660A                 call    sub_403A70
```

The challenge checks that an argument was passed to the challenge, then pushes `argv[1]` onto the stack and calls `sub_403A70`, this functions appears to be an initialization function for the backdoor. There is a lot of network setup going. This function will help layout the structure of the backdoor for analysis further forward. Stepping through `backdoor_init` the function calls `inet_addr()` on the argument passed in on the command line followed by setting up to raw sockets, one appears to be a passive listening socket for UDP that has a timeout set for ten seconds. The second socket is a raw UDP socket that has `IP_HDRINCL` set for it. Some synchronization objects are setup: a mutex and semaphore. The function wraps up by calling a random debug test, this would be why a NULL object returned from the debug constructor wouldn’t work. At this point a roughed-out structure for the backdoor is follows

```
00000000 backdoor_object struc ; (sizeof=0x1CC, mappedto_100)
00000000 wsa_data       WSADATA ?
00000190 sock_raw        dd ? ; XREF: backdoor_init+1CC/w
00000194 sock_passive_listener dd ? ; XREF: backdoor_init+87/w
00000194 init_13241104   dd ? ; XREF: backdoor_init+118/r
00000194 init_1226       dd ? ; XREF: backdoor_init+2B/w
00000194 init_1A0        dd ? ; XREF: backdoor_init+1F/w
000001A4 field_1A4       dd ?
000001A8 field_1A8       dd ?
000001AC field_1AC       dd ?
000001B0 field_1B0       dd ?
000001B4 field_1B4       dd ?
000001B8 field_1B8       dd ?
000001BC field_1BC       dd ?
000001C8 semaphore       dd ? ; XREF: backdoor_init+23A/w
000001C4 mutex           dd ? ; XREF: backdoor_init+277/w
000001C8 debug_object    dd ? ; XREF: backdoor_init:loc_403D00/r
000001CC backdoor_object ends
```

Listener Thread
The listener thread starts to utilize the structures that have been initialized and the work put forward to this point is very helpful in static analysis. Stepping through the thread started at address 0x00404310, which will be named listener_thread, first calls recvfrom on the passive listening socket which confirms this to be a listening thread at least. At the beginning of the function a block of memory is allocated with the size of 1500 bytes, which stands out as an IPV4 packet size. Upon successful recvfrom on the passive listening socket some validation takes place.

First, the packet is verified to be IPPROTO_UDP, next the UDP header is retrieved and the length is verified to be greater than zero, the payload is saved off and the last two tests are the validation that is pertinent to this challenge: the UDP destination port must be 4356 and the Evil Bit must be set per RFC 3514.

Once network validation is completed the challenge allocates another memory block and starts copying over network data from the IP and UDP packet. Most of this is straightforward if one has the IPV4 and UDP packets in IDA. Create a new structure of size 0x14 and populates it with the appropriate titles. A more difficult part to understand what is immediately going on is the final part of the function starting at 0x00404525. A high-level look at this section of code shows a mutex being waited on, once the mutex is obtained a new object is altered along with a call to a function that reeks of C++ std code. After this object is altered ReleaseSemaphore is called followed by ReleaseMutex. The function then frees the received IPV4 packet and returns to another call of recvfrom. High level this looks like some sort of shared data structure that is protected by a mutex and the count of items is managed by a semaphore. The std data structure being used is std::queue. The backdoor code can now be updated to look as follows:

```
00000000 backdoor_object struc ; (sizeof=0x1CC, mappedto_100)
00000000 wsa_data   WSADATA ?
00000190 sock_raw   dd ?                ; XREF: backdoor_init+1CC/w
00000194 sock_passive_listener dd ?     ; XREF: backdoor_init+B7/w
00000194                ; backdoor_init+118/r ...
00000198 listen_port dw ?                ; XREF: backdoor_init+2B/w
0000019A port_1324 dw ?                  ; XREF: backdoor_init+1F/w
000001A0 finished dd ?                   ; XREF: listener_thread+21/r
000001A4 field_1A4 dd ?                  ; XREF: listener_thread+264/r
000001A8 field_1A8 dd ?                  ; XREF: listener_thread+26A/o
000001AC queue std_queue ?                ; XREF: listener_thread+264/r
000001AC                ; listener_thread+26A/o
000001C0 semaphore dd ?                   ; XREF: backdoor_init+23A/w
000001C4 mutex dd ?                       ; XREF: backdoor_init+277/w
000001C4                ; listener_thread+229/r
000001C8 debug_object dd ?                ; XREF: backdoor_init:loc_403D00/r
000001C8                ; listener_thread:loc_404360/r ...
000001CC backdoor_object ends

The shared structure that is queued is as follows:
```

```
00000000 listener_object struc ; (sizeof=0x14, mappedto_130)
00000000 saddr dd ?                      ; XREF: listener_thread+1E2/w
00000004 daddr dd ?                      ; XREF: listener_thread+1E4/w
00000008 sport dw ?                      ; XREF: listener_thread+1E7/w
0000000A unused dw ?                     ; XREF: listener_thread+1F3/w
0000000C udp_payload dd ?                ; XREF: listener_thread+1F3/w
00000010 udp_payload_len dd ?            ; XREF: listener_thread+1F3/w
00000014 listener_object ends
```
Challenge 9: evil | Flare-On 8

Handler Thread

This leaves a final thread that is created in main at address 0x00404680, named handler_thread. This thread confirms the suspicion of shared data structure between a listener and a handler with the reverse synchronization objects protecting the std::queue. First the handler waits on the semaphore, followed by waiting on the mutex protecting the queue, an object is removed from the queue and placed in what IDA has labeled Block. At address 0x404793 the UDP payload is retrieved and stored in EAX. A switch case is handled on the UDP payload at offset 0. Initially the labels can be renamed to their operation code value: {loc_404F23 : handle_01, loc_404904 : handle_02, loc_40485A : handle_03}. There is some follow up functionality after the switch case that appears to be a default handler.

Working though the different function handlers each one parses and operates on the payload differently. Following functions within the handlers show a function that deals with decryption and another that deals with sending a response. Each will be briefly described below but not in as much detail as previously. Some of the key functions or handlers that support the handler thread will be covered in the next follow sections.

Backdoor Send 0x00403D40

As noted earlier the backdoor creates two sockets, one is a passive listening raw socket, the second is a raw socket that requires the IP header to be included when sending. This option was chosen to have more control over the sending and received ports, also, this allowed the evil bit to be set on outgoing packets to conform to RFC standards. With IPV4 and UDP headers in the IDB the markup of this function isn't too difficult, it just builds a full packet to send starting with the IP header ensuring the evil bit is set, followed by a UDP header and the payload is passed into this function to be sent.

Decrypt 0x004067A0

This function is obviously crucial as it is a final key in decrypting the flag. There is a final gotcha in this function that will be covered later. First, the function allocates a block of memory in the size of 0x1C bytes, this allocated memory is used for importing a key in the format of PLAINTEXTKEYBLOB. This is seen following the allocation as shown in Figure 23. Two large factors to determine that this structure is a PLAINTEXTKEYBLOB is first the structure is being passed to CryptImportKey and second are the structure values. The second parameter to CryptImportKey is a “PUBLICKEYSTRUC, also known as the BLOBHEADER structure, indicates a key’s BLOB type and the algorithm that the key uses.” (Microsoft, 2021). The structure is defined as:

```c
typedef struct _PUBLICKEYSTRUC {
    BYTE bType;
    BYTE bVersion;
    WORD reserved;
    ALG_ID aiKeyAlg;
} BLOBHEADER, PUBLICKEYSTRUC;
```

Looking at the disassembly the compiler optimized the bType and bVersion together to be 0x208, bType 0x08 is a PLAINTEXTKEYBLOB. Documentation for the WinCrypt library has never been very clear, Microsoft states,

> The CryptImportKey function can be used to import a plaintext key for symmetric algorithms; however, we recommend that, for ease of use, you use the CryptGenKey function instead. When you import a plaintext key, the structure of the key BLOB that is passed in the pbData parameter is a PLAINTEXTKEYBLOB.

> You can use the PLAINTEXTKEYBLOB type with any algorithm or type of key combination supported by the CSP in use. (Microsoft, 2021)
The next parameter for the structure is the aiKeyAlg, 0x6802, CALG_SEAL. CALG_SEAL is stream cipher that according to Microsoft is not supported within the documentation and header files it has the comment, “Deprecated. Do not use”. The plaintext key blob has the following structure:

```c
typedef struct PlainTextKeyBlob
{
    BLOBHEADER hdr;
    DWORD keySize;
    BYTE bytes[];
} PlainTextKeyBlob, * PPlainTextKeyBlob;
```

For this challenge the key size is always 16 bytes so the following IDA structure could be used and Figure 24 shows an updated markup of the IDB using this structure. The rest of decrypt is standard decryption using the Windows API. The question to why CALG_SEAL is being used and if it is really supported is left unanswered currently.

```asm
00000000 PlainTextKeyBlob struc ; (sizeof=0x1C, mappedto_132)
00000000 blob_header     BLOBHEADER ?
00000008 keysize         dd ?
0000000C key             db 16 dup(?)
0000001C PlainTextKeyBlob ends
```

Figure 23: Initializing PlainTextKeyBlob

Figure 24: Initializing PlainTextKeyBlob with defined structures
Default Operation Handler 0x00: Echo

All the messages being sent to this challenge have the following header:

```c
typedef struct _BackdoorHeader {
    uint32_t opcode;
    uint32_t len
} BackdoorHeader;
```

The Echo command uses the following message format

```c
typedef struct _BackdoorCommand {
    BackdoorHeader hdr;
    char payload[0];
} BackdoorCommand, BackdoorResponse;
```

The same structure is used for the response. Echo takes a message and echoes it back. This operation is useful for simplifying how the network communications works.

Operation Handler 0x01: Rickroll

The challenge authors felt a FlareOn year should not go by without getting Rickrolled at least once. This challenge will decrypt a bitmap image and display it within the challenge's console window and send back the message, N3ver_G0nNa_g1ve_y0u_Up@flare-on.com. The key used for Rickroll is separate from the final flag key. The key is intentionally set to match a __stdcall or __cdecl function prologue and will display as code IDA. There are multiple ways to get Rickrolled in this challenge, one is by sending this opcode and the other is by sending an invalid GetFlag message.

```
55 8B EC 64 A1 00 00 00 6A ff 68 d4 21 41 00 50
```

The packet format for Rickroll doesn't require a payload and thus the length in the header is set to zero. The response uses a BotResponse packet.

```c
typedef struct _BotCommand {
    BackdoorHeader hdr;
} BackdoorCommand_Rickroll;
```

Operation Handler 0x02: Set Encryption Key

When sending the opcode 0x03 to get the flag an uninitialized key is used. Initializing this key is done by sending four packets of type BotCommand with opcode 0x02. The messages are first string compared against for strings that have been deobfuscated. If a string matches, then a CRC32 value is generated on the string including the null character. Table 3: Password and CRC32 values contains the four passwords and include the generated CRC32.
Table 3: Password and CRC32 values

The correct key for the challenge is

E3 FC 31 F4 D8 E9 B0 78 77 06 6B 5A A2 4F 5B 95

Operation Handler 0x03: Get Flag

Finally, after sending all the correct passwords one can request the flag by sending a packet in the format:

typedef struct _BotCommand_Get_Flag
{
  BackdoorHeader hdr;
  USHORT sub_cmd;
  char payload[0];
} BotCommand_Get_Flag;

The response will still be a BackdoorResponse. The sub_cmd must be 0x5D4A, also known as the IMAGE_DOS_SIGNATURE. If one is to not send the proper sub_cmd, then the key gets zeroed. Send a proper sub_cmd and the flag gets decrypted and sent back.

n0_mOr3_eXcEpti0n$_p1ea$e@flare-on.com

The Final Gotcha: Hooking CryptImportKey

So, you have done all the reversing and you sit down to write a quick python script to communicate with this challenge and notice you have to figure out how to do raw sockets to set that evil bit, or maybe you just decide to patch that little check out. Well if you go the networking route Figure 28: backdoor_client.py is provided to give an option on how to interact with this challenge. What if you said, I know what it is doing, I think it will be easier to just find an implementation of SEAL and decrypt the flag. Here is the gotcha, this challenge doesn’t use SEAL. There is a detours hook that can be easily overlooked in the backdoor_object initializer. The hook is set in function 0x4060D0 and called from 0x00403A7E. This section of code only enables the hook, it has already been setup in a CPP Initializer. There is some giveaway in this function though and that is that it resolves CryptImportKey. The global variable 0x6D1690 can be cross referenced to a CPP Initializer at address: 0x004020F0, hook_CryptImportKey_init. The initialization function resolves CryptImportKey and then passes it to a function that also contains the hook for CryptImportKey found at 0x0040660E, hook_CryptImportKey. The hook function is simple in just checking if CALG_SEAL is used, if it is then the algorithm is replaced with CALG_RC4 as seen below. This operation should not be bothered by any other cryptography operations as Microsoft doesn’t support CALG_SEAL. So, if one wants to manually decrypt the flag, CALG_RC4 must be used.
```c
int _stdcall hook_CryptImportKey(int a1, PlainTextKeyBlob *a2, int a3, int a4, int a5, int a6)
{
    if ( a2->blob_header.aiKeyAlg == CALG_SEAL )
        a2->blob_header.aiKeyAlg = CALG_RC4;
    GetProcAddressEx((int)&unk_523422, CryptImportKey);
    return CryptImportKey_4realz(a1, a2, a3, a4, a5, a6);
}
```

Figure 25: hook_CryptImportKey
import pefile
import sys
import os
M32 = 0xffffffff

def main():
    if len(sys.argv) != 3:
        print("usage: generate_hashes <in.dll> <out.hashes>")
        sys.exit(0)

d = [pefile.DIRECTORY_ENTRY["IMAGE_DIRECTORY_ENTRY_EXPORT"]]
pe = pefile.PE(sys.argv[1], fast_load=True)
pe.parse_data_directories(directories=d)

    names = [e.name for e in pe.DIRECTORY_ENTRY_EXPORT.symbols]
    names = [i for i in names if i]

    with open(sys.argv[2], 'wb') as f:
        f.write(b"enum %s_hash
n" % os.path.split(sys.argv[1])[-1].split('.')[0].encode())
        f.write(b"{
"
        for name in sorted(names):
            h = 0x40
            for x in name:
                h = x - 0x45523f21 * h & M32
            f.write(b"    %s = 0x%x,
" % (name, h))
        f.write(b"};")

if __name__ == '__main__':
    main()
import sys

callfuncs = [
    b'\x33\x0C\xF7\xF0\xEB\x00\xEB',
    b'\x33\x08\xB\x00\xEB\xFF\xE8',
    b'\x33\xF6\xF7\xF6\xEB\xFF\xD2',
    b'\x33\x8B\x00\x74\x03\x75',
    b'\x33\xFF\xF4\xF4\x33\x0C\x74',
    b'\x33\xFF\xF7\xF7\x33\x0C\x74',
    b'\x33\x74\xF7\xF0\x33\x0C\x74',
    b'\x33\x0F\xF0\xEB\xFF\xD2',
    b'\x33\x0F\xF7\x5B\x5D\xC3'
]

def main():
    if len(sys.argv) != 3:
        print("usage: nop_anti_dis.py <in out>\n")
        sys.exit(0)

    with open(sys.argv[1], 'rb') as f:
        d = f.read()

    for cf in callfuncs:
        d = d.replace(cf, nops)

    with open(sys.argv[2], 'wb') as f:
        f.write(d)

if __name__ == "__main__":
    main()
from ctypes import *
import socket
from scapy.all import *

ADDR = "127.0.0.1"
PORT = 4356
SPORT = 8546

FLAG_SUB_COMMAND = 0x5A4D

RICKROLL_TYPE_OPCODE = 1
RICKROLL_TYPE_FAIL = 2

class Header(Structure):
    _pack_ = 1
    _fields_ = [
        ('cmd', c_int),
        ('len', c_int),
    ]

def EchoRequest_factory(msg):
    class EchoRequest(Structure):
        _pack_ = 1
        _fields_ = [
            ('hdr', Header),
            ('msg', c_char * len(msg))
        ]
    er = EchoRequest()
    er.hdr.cmd = 0
    er.hdr.len = len(msg)
    er.msg = msg
    return er

def RickRoll_factory():
    class RickRoll(Structure):
        _pack_ = 1
        _fields_ = [
            ('hdr', Header),
            ('msg', c_char)
        ]
    er = RickRoll()
    er.hdr.cmd = 1
    er.hdr.len = 0
    return er

def SetKey_factory(msg):
    class SetKey(Structure):
        _pack_ = 1
        _fields_ = [
            ('hdr', Header),
            ('msg', c_char * len(msg))
        ]
    sk = SetKey()
    sk.hdr.cmd = 2
    sk.hdr.len = len(msg)
sk.msg = msg
return sk

def GetFlag_factory(sub_command = 0):
    class GetFlag(Structure):
        _pack_ = 1
        _fields_ = [
            ('hdr', Header),
            ('sub_command', c_ushort)
        ]
gk = GetFlag()
gk.hdr.cmd = 3
gk.hdr.len = 2
gk.sub_command = sub_command
return gk

def Flag(pkt):
    hdr = Header()
    memmove(byref(hdr), pkt, sizeof(hdr))
    flag = pkt[sizeof(hdr):pkt[sizeof(hdr)] + hdr.len].decode()
return flag

def sendto(sock, data, daddr, saddr, dport, sport):
    packet = IP(ttl=128, dst=daddr, src=saddr, flags=4, frag=0)/UDP(sport=sport, dport=dport)/bytes(data)
    packet = bytes(packet)
    sock.sendto(packet,(daddr,dport))

def Echo():
    sock = socket.socket(socket.AF_INET, socket.SOCK_RAW, socket.IPPROTO_UDP)
    sock.setsockopt(socket.IPPROTO_IP, socket.IP_HDRINCL, 1)
    rsock = socket.socket(family=socket.AF_INET, type=socket.SOCK_DGRAM)
    rsock.bind((ADDR,SPORT))
    er = EchoRequest_factory(b'Hello Echo Request')
    sendto(sock, er, ADDR, ADDR, PORT, SPORT)
data, addr = rsock.recvfrom(1024)
print(Flag(data))

def RickRoll(rr_type):
    sock = socket.socket(socket.AF_INET, socket.SOCK_RAW, socket.IPPROTO_UDP)
    sock.setsockopt(socket.IPPROTO_IP, socket.IP_HDRINCL, 1)
    rsock = socket.socket(family=socket.AF_INET, type=socket.SOCK_DGRAM)
    rsock.bind((ADDR,SPORT))

    if rr_type == RICKROLL_TYPE_OPCODE:
        msg = RickRoll_factory()
    elif rr_type == RICKROLL_TYPE_FAIL:
        msg = GetFlag_factory()
    sendto(sock, msg, ADDR, ADDR, PORT, SPORT)
data, addr = rsock.recvfrom(1024)
print(Flag(data))
def GetFlag():
    sock = socket.socket(socket.AF_INET, socket.SOCK_RAW, socket.IPPROTO_UDP)
    sock.setsockopt(socket.IPPROTO_IP, socket.IP_HDRINCL, 1)

    rsock = socket.socket(family=socket.AF_INET, type=socket.SOCK_DGRAM)
    rsock.bind((ADDR, SPORT))

    sk = SetKey_factory(b's3cret\x00')
    sendto(sock, sk, ADDR, ADDR, PORT, SPORT)

    sk = SetKey_factory(b'5Ex\x00')
    sendto(sock, sk, ADDR, ADDR, PORT, SPORT)

    sk = SetKey_factory(b'g0d\x00')
    sendto(sock, sk, ADDR, ADDR, PORT, SPORT)

    sk = SetKey_factory(b'L0ve\x00')
    sendto(sock, sk, ADDR, ADDR, PORT, SPORT)

    gk = GetFlag_factory(FLAG_SUB_COMMAND)
    sendto(sock, gk, ADDR, ADDR, PORT, SPORT)

    data, addr = rsock.recvfrom(1024)
    print(Flag(data))

Echo()
RickRoll(RICKROLL_TYPE_OPCODE)
RickRoll(RICKROLL_TYPE_FAIL)
GetFlag()

Figure 28: backdoor_client.py
References


