Flare-On 6: Challenge 12 – help.zip

Challenge Author: Ryan Warns (@NOPAndRoll)

In this challenge players were given a .zip file containing a memory dump, a packet capture, and a ransom/help document. When creating this challenge, I wanted to present players with some of the techniques and APIs I’ve seen during the past year of analyzing malware for FLARE. During malware analysis it is common to have to analyze samples that have mechanical bugs (e.g. using an API incorrectly), logical flaws (e.g. not functioning correctly sometimes), or both. This challenge was a collection of user-mode and kernel-mode binaries and shellcode. Players needed to analyze these binaries to understand how they communicated with each other to solve the challenge.

INITIAL CRASH TRIAGE

The .txt file included with this challenge indicates that the crash dump, help.dmp, and packet capture, help.pcapng are the result of a system which was infected by buggy malware and eventually crashed. Skimming the PCAP we can see some nonstandard TCP communication on ports 4444, 6666, 7777, and 8888 but there are no clues as to what the data streams over these ports contain.

We can probably guess that the PCAP comes from a set of samples running when the crash dump was created, but without any hint as to how to proceed we will start our analysis with the crash dump.

Note that a significant part of this challenge involves analyzing and extracting data from the crash dump. There are a multitude of tools that support memory dump analysis, including volatility and foremost, but for this solution I’ll be using WinDbg to show how I would approach a crash dump analysis scenario.

Our first step is to triage the crash. We can use !analyze -v to get a general summary of what caused the crash. The output of this command is shown in Figure 1.
kd> !analyze -v
******************************************************************************
*                                                                             *
*                        Bugcheck Analysis                                    *
*                                                                             *
******************************************************************************

SYSTEM_THREAD_EXCEPTION_NOT_HANDLED (7e)
This is a very common bugcheck. Usually the exception address pinpoints
the driver/function that caused the problem. Always note this address
as well as the link date of the driver/image that contains this address.
Arguments:
  Arg1: ffffffffc00000005, The exception code that was not handled
  Arg2: ffffffa8003f9c621, The address that the exception occurred at
  Arg3: fffff88007c6b1b0, Context Record Address
  Arg4: fffff88007c6b958, Exception Record Address

Figure 1: Initial crash triage

From there we can use the !process command and a stack trace to figure out more
information about where we were executing at the time of the crash (Figure 2).
kd> .cxr 0xffffffff88007c6b1b0
rax=fffffff8003f9c610 rbx=fffffff80040c65c0 rcx=fffffff80036ab5c0
rdx=fffffff800033c8138 rsi=fffffff80018cc090 rdi=0000000000000001
rip=fffffff8003f9c621 rsp=fffffff88007c6bb98 rbp=0000000007c6bb0
r8=fffffff80002c3f400 r9=0000000000000000 r10=0000000000000000
r11=fffffff80002c3ae80 r12=fffffff80036ab5c0 r13=fffffff880033bdcc0
r14=0000000000000000 r15=fffffff80000b94080
iopl=0       nv up ei ng nz na pe nc
cs=0010 ss=0018 ds=002b es=002b fs=0053 gs=002b
efl=00010286
fffffff80`03f9c621 64a10000000050648925 mov eax,dword ptr
fs:[2589645000000000h] fs:0053:25896450`00000000=?????????
kd> k
# Child-SP     RetAddr     Call Site
00 ffffffff80`07c6bba8 00000000`0001093a 0xffffffff80`03f9c621
01 ffffffff80`07c6bba0 00000000`00010ba8 0x1093a
02 ffffffff80`07c6bba8 ffffffff`fffffff 0x10ba8
03 ffffffff80`07c6bb00 00000000`00000080 0xffffffff`fffffff
04 ffffffff80`07c6bb80 ffffffff880`033bdce7 0x80
05 ffffffff80`07c6bbbc0 00000000`00000000 man+0xlce7

kd> !process -1 0
PROCESS ffffffff80018cc090
  SessionId: none  Cid: 0004    Peb: 00000000  ParentCid: 0000
  DirBase: 00187000  ObjectTable: ffffffff8a0000017f0  HandleCount: 657.
  Image: System

Figure 2: Context at time of system crash

Now we know that we’re currently executing what looks like garbage code in the System
process at the time of the crash. We also see a driver, man.sys, on the call stack. We can start
our analysis with this driver, but it’s generally good practice to see if you can extract any
additional information from the crash state before digging into the code that caused it. The call
stack looks corrupted, so let’s start by observing the code that caused the crash, the man.sys
return address, and the rest of the stack. The code that caused the crash shows us why the
stack is misaligned in Figure 3.
kd> ub rip
ffffffa80`03f9c60e cc int 3
ffffffa80`03f9c60f cc int 3
ffffffa80`03f9c610 8bff mov edi,edi
ffffffa80`03f9c612 55 push rbp
ffffffa80`03f9c613 8bec mov ebp,esp
ffffffa80`03f9c615 6aff push 0FFFFFFFFFFFFFFFFh
ffffffa80`03f9c61c 68a80b0100 push 10BA8h
ffffffa80`03f9c61d 683a090100 push 1093Ah

Figure 3: Code responsible for the crash

We should already be suspicious of this code since it shows an unusual mixture of 32-bit and 64-bit registers, but for now note the four push instructions in this snippet account for 32 bytes on the stack. We can assume this function starts at fffffffa80`03f9c610 based on the int 3 instructions. Now let’s look at the stack (Figure 4):

kd> dqs @rsp
ffffff80`07c6bb98 00000000`0001093a
ffffff80`07c6bbba0 00000000`00010ba8
ffffff80`07c6bbba8 ffffffff ffffffff
ffffff80`07c6bbb0 00000000`00000080
ffffff80`07c6bbb8 fffffff80`033bdce7 man+0x1ce7
ffffff80`07c6bbbc0 00000000`00000000
ffffff80`07c6bbcc8 00000000`00000000
ffffff80`07c6bbd0 00000000`00000000
ffffff80`07c6bbd8 00000000`00000000
ffffff80`07c6bbe0 fffffff80`03f9c610
ffffff80`07c6bbe8 00000000`00000000
ffffff80`07c6bbf0 00000000`00000000
ffffff80`07c6bbf8 fffffff80`02d5cb8a nt!PspSystemThreadStartup+0x5a
ffffff80`07c6bc00 fffffff80`036ab5c0
ffffff80`07c6bc08 00000000`00000000
ffffff80`07c6bc10 00000000`00000000

Figure 4: Stack at the time of the crash

This stack confirms that man+0x1ce7 is the code that called our garbage code. We can also see PspSystemThreadStartup on the stack, which is an internal function that is called as part of PsCreateSystemThread. It’s a reasonable guess that this function is the entry point for our

1 https://docs.microsoft.com/en-us/windows/hardware/drivers/ddi/content/wdm/nf-wdm-pscreatesystemthread
system thread (Figure 5), so let’s look at this function further.

```plaintext
kd> ub man+0x1ce7 L9
man+0x1cbf:
fffff880`033bdcbf cc      int     3
fffff880`033bdcc0 48894c2408 mov qword ptr [rsp+8],rcx
fffff880`033bdcc5 4883ec38 sub rsp,38h
fffff880`033bdcc9 488b442440 mov rax,qword ptr [rsp+40h]
fffff880`033bdccce 488b4058 mov rax,qword ptr [rax+58h]
fffff880`033bdcd2 4889442440 mov qword ptr [rsp+40h],rax
fffff880`033bdcd7 488d155aa40000 lea rdx,[man+0xc138]
(fffff880`033c8138)]
fffff880`033bdcd8 488b4c2440 mov rcx,qword ptr [rsp+40h]
fffff880`033bdce3 ff542420 call qword ptr [rsp+20h]
```

**Figure 5: System thread entry point**

This is a small function whose only job is to call a function from a structure passed as a parameter. Luckily for us this parameter is also saved on the stack. We can walk backwards through our garbage function and our system thread entry point to find that the parameter is stored at `fffff880`07c6bc00 on the stack and contains `ffffffa80`036ab5c0. WinDbg contains many commands to attempt to automatically identify and parse structures in memory and it’s usually helpful to take a couple guesses in cases like this. As seen in Figure 6, we can use the `!object` command in this case to figure out that the parameter is a DRIVER_OBJECT.
Figure 6: Inspecting parameter object

The DRIVER_OBJECT structure contains the start address and size of the PE corresponding to the driver, so we can dump this to disk with the .writemem command (Figure 7).

Figure 7: Extracting problem driver

A quick glance at this driver confirms it is a 32-bit driver file, which explains the crash since we’re running a 64-bit operating system.

We now know that man.sys is likely responsible for crashing the system because it attempted to load and execute a 32-bit driver on a 64-bit system. At this point we could continue to poke around the memory dump with various tools, but since we have some context regarding what
man.sys can do let’s start our analysis on that.

**MAN.SYS ANALYSIS**

The `lm2` command in WinDbg will give us the memory region associated with man.sys. However, as seen in Figure 8, after writing this file to disk we’ll quickly notice that the file’s PE headers have been wiped.

```
kd> lm m man
Browse full module list
start        end                  module name          (deferred)
fffff880`033bc000  ffffff880`033cb000    man

kd> .writemem C:\Users\ryan.warns\FLARE_On_2019\man_dumped fffff880`033bc000
   ffffff880`033cb000-1
```

*Figure 8: Dumping man.sys*

At this point we could begin our analysis by pretending this binary is a shellcode buffer, but it usually helps to attempt to rebuild a PE header so we can rely on IDA and other analysis tools. Rebuilding PE headers is a common problem when dumping binaries from memory. For a completely wiped header, a straightforward approach is to take a full PE header from another binary and fix up the offsets for each section to match what is in memory.

Loaded PEs generally have a section alignment which results in padding being added to the end of the actual data. For the dumped memory associated with man.sys we can skim the data to look for many consecutive NULL bytes to make a reasonable guess about where sections begin, then look at the data in the section to guess about which section it is. For this challenge we only need to identify the .text section properly, and then the rest of the sections can be treated as data.

WinDbg can also help us guess the sections correctly. The DRIVER_OBJECT structure we saw earlier also contains the entry point for the PE in the DriverInit member. If we can find the DRIVER_OBJECT for man.sys, the entry point is likely in the .text section. We can use the `!object` command to list all the DRIVER_OBJECTs on the system. From there we just need to find a DRIVER_OBJECT that has an entry point (Figure 9) in the range for man.sys.

---

2 https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/lm--list-loaded-modules-
3 https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/object
Figure 9: Extracting the entry point for man.sys

This means the entry point for man.sys is at 0xfffff880`033c1110 and also helps us identify where the .text section is. From here we can continue scanning for padding sections and rebuilding our PE header to get something usable in IDA. Figure 10 shows the full correct section, but anything reasonably close is good enough.

Note that for the rest of this analysis all addresses used will be relative to where man.sys is loaded in the dump, ffff800075060.
Most malware rootkits are software drivers, meaning they are not directly tied to or responsible for communicating with a piece of hardware and instead process arbitrary requests from user-mode callers. In user mode this communication is done via calls to DeviceIoControl with a handle to our driver’s DEVICE_OBJECT. In the kernel, this API results in the I/O Manager delivering an IRP_MJ_DEVICE_CONTROL request to our driver. A summary of this communication is shown in Figure 11.

![Diagram of communication with software drivers](image)

Figure 11: Communication with software drivers

The I/O Handler functions are present in the DRIVER_OBJECT: the MajorFunction member of this structure is an array of function members that Microsoft exposes which map to standard I/O functionality such as reading, writing, etc. For this driver all of the routines are the same function, `ffff8800`033c10b0. Tracing this function through a wrapper function we can find

---

4 https://docs.microsoft.com/en-us/windows-hardware/drivers/kernel/irp-major-function-codes
the IRP_MJ_DEVICE_CONTROL handler at ffff880`033c0de0.

DeviceIoControl takes a command code as a parameter which is specific to the driver being communicated with. man.sys’s IRP_MJ_DEVICE_CONTROL handler accepts the following codes:

1. 0x237BE8
2. 0x23BEAC
3. 0x22AF2C
4. 0x22AF34
5. 0x2337BC
6. 0x22AF28
7. 0x2248D0
8. 0x22F378
9. 0x23EAF0

Note that the first four commands all call the same function with varying parameters, and if we check the cross references to PsCreateSystemThread based on our initial analysis it is only called once, in the function responsible for handling command 0x2337BC.

We already have a good idea that ffff880`033bddb0 is responsible for loading and executing a payload in kernel mode. Most of the process for in-memory loading a driver process is similar to public techniques for in-memory loading DLLs. However, in attempting to identify what each function in this routine is doing we’ll quickly notice string encrypting being used by this driver.

The function at address ffff880`033bd190 is used to decrypt stack strings using RC4. Since we don’t want to deal with decrypting each string individually it would be good to solve all strings with a script or tool. Each string has a different key, and the way this function is implemented in the binary means a scan with FLOSS won’t work. Instead, using flare-emu we can write a simple script to decrypt each string based on the cross references to the RC4 function. A sample script is shown in Figure 12.

5 https://github.com/stephenfewer/ReflectiveDLLInjection
6 https://github.com/fireeye/flare-floss
7 https://github.com/fireeye/flare-emu
Figure 12: flare-emu script to decrypt stack strings

With our decrypted strings annotated we can confirm that the function at `fffff880`033bddb0` contains code to load a driver: after the standard PE loading steps (memory mapping, relocations, etc.), this function calls `ObCreateObject` and `ObInsertObject` to create a DRIVER_OBJECT for the loaded payload.

With this function confirmed we can move on. The function at `fffff880`033bec50` is called for multiple commands and during `DriverEntry` so let’s look at that function next. A quick side-by-side comparison with our driver loading function from earlier will make it apparent that this is also loading PE files. For example, the function responsible for fixing relocations is present in both functions. The calls to `KeStackAttachProcess` and `ZwAllocateVirtualMemory` (hidden by string obfuscation) are a giveaway that this function is loading and running code in user mode instead of kernel mode.

Finally, at the end of this function `man.sys` will either run function `fffff880`033bffe0` or build a structure and insert it into a list. We’ll analyze this function shortly, but for now we know some of the structure that’s stored in the list at `fffff880`033c8158` (Figure 13):
typedef struct _INJECTED_PAYLOADS
{
    ULONG numberFromParam; // comes from IOCTL param structure
    LIST_ENTRY link; // Flink and Blink
    KMUTEX payloadMutex;
    ULONG_PTR baseAddress; // base address of loaded code
    ULONG_PTR numberFromParam2; // comes from IOCTL param structure
    ULONG_PTR loadedDllSize; // Full size of loaded payload
    BOOLEAN param1; // Set on which IOCTL is called
    BOOLEAN param2; // Set on which IOCTL is called
    ULONG_PTR numberFromParam3; // comes from IOCTL param structure
    PEPROCESS process; // process that this payload is inject
} INJECTED_PAYLOADS, *PINJECTED_PAYLOADS;

Figure 13: User-mode payload structure (so far)

Note that the EPROCESS in this structure is derived from a call to
PsLookupProcessByProcessId based on a PID that is passed in as part of the IOCTL
parameter.

We can also use the !list command in WinDbg to see what data is in this list at the time of
the crash to attempt to deduce what some of these other fields are. A trimmed output from
this command is in Figure 14.
From this we can already see a hint if we skimmed the PCAP from earlier – some of these structures contain the port numbers seen in the PCAP at offset 0x68 (numberFromParam3 from before).

If the fifth parameter is FALSE the driver also performs RC4 encryption on the user-mode payload after it is loaded. However, rather than using a hardcoded key the driver uses part of the structure from Figure 13 as a key. Everything from baseAddress onward is used as the key to encrypt the payload in memory. You can use the command in Figure 15 to list the RC4 key for key for each payload.
We can see that the fifth parameter is only TRUE in the call from DriverEntry, meaning that when this driver starts one of the first things it does is inject some kind of payload.

The payload is stored at `fffff880`033c3110 in the driver and is injected into PID 876 which for this dump is the `svchost.exe -k netsvcs` process. We'll analyze the user-mode payloads in the next section but for now let's finish up the driver analysis so we have all the context we need.

IOCTLS 0x22F378 and 0x23EAF0 are similar conceptually: both deal with tasking one of the user-mode payloads to do something. Let’s start with 0x22F378 which calls function `fffff880`033c0580. This function uses the string encoding we’ve seen several times before to encode ZwFreeVirtualMemory, which already tells us that this function is likely interacting with user-mode memory. The input to this function comes from the DeviceIoControl call in user mode and has the structure shown in Figure 16.

```c
typedef struct _TASK_USER_MODE
{
    ULONG moduleID;
    ULONG userModeCommandCode;
    PVOID bufferForUserModePayload;
    ULONG userModeBufferSize;
} TASK_USER_MODE, *PTASK_USER_MODE;
```

This function starts by calling `fffff880`033bf960, which uses moduleID from the input parameter to search the list described earlier to find the corresponding INJECTED_PAYLOADS structure. From there the function at `fffff880`033bffe0 is responsible for proxying the input data into the correct target process and executing the payload for that INJECTED_PAYLOADS structure. This consists of several steps:

1. Attach to the EPROCESS from the INJECTED_PAYLOADS structure
2. Call the function at `fffff880`033bf310 to allocate user-mode memory
3. The function at `fffff880`033bf510 will resolve NtCreateThreadEx and call it to execute a user-mode thread
   a. This function also decrypts the payload during executing, using the same RC4 key described earlier
   b. Inside of INJECTED_PAYLOADS, offset 0x58 is the offset in the user-mode payload to execute
4. Copy return data from user-mode payload back to the IOCTL caller
For step 3, the driver allocates a shared structure to proxy input/output data to the user-mode payload (Figure 17):

```c
typedef struct _CMD_HEADER
{
    ULONG cmdCode;       // comes from IOCTL 0x22F378 parameter
    PVOID inData;        // comes from IOCTL 0x22F378 parameter
    ULONG inDataLen;     // comes from IOCTL 0x22F378 parameter
    ULONG uSetByUM;
    BOOLEAN bSetByUM;
    PVOID outData;
    ULONG outDataLength;
} CMD_HEADER, *PCMD_HEADER;
```

**Figure 17:** Structure to communicate with user-mode payloads

The last four members of this structure are set by the user-mode payload. After the user-mode payload executes the function at `fffff880`033c0580 checks bSetByUM. If this member is set to TRUE by the user-mode payload then the function at `fffff880`033bf9e0 is called before the output data is returned.

This function is structured similarly to the previous function: the module list is searched, input data is copied to the proper process, and the function at `fffff880`033bf510 is called to execute a user-mode thread. The one difference is that instead of checking or the moduleID in the INJECTED_PAYLOADS structure as before the param1 member is checked. For the module list in the crash dump this corresponds to the payload at `0x02100000` inside of `dwm.exe` (EPROCESS `fffffa80`035fd060).

Let’s look at IOCTL `0x23EA`F0 to wrap up our analysis of `man.sys`. This IOCTL handler is the function at `fffff880`033bfed0 and is again structured like the previous two functions we’ve analyzed – the module list is searched, and a user-mode payload is executed. For this function the list is searched twice: once for a comparison to the param2 member, and once per module ID, before calling the function at `fffff880`033bfd60 which has the same pattern of allocating user-mode data and executing a thread. As seen in Figure 18, we now have a slightly clearer idea of what the INJECTED_PAYLOADS structure should look like:
typedef struct _INJECTED_PAYLOADS
{
    ULONG moduleId;
    LIST_ENTRY link;
    KMUTEX payloadMutex;
    ULONG_PTR baseAddress; // base address of loaded code
    ULONG_PTR runOffset; // Offset in payload to execute
    ULONG_PTR loadedDllSize; // Full size of loaded payload
    BOOLEAN moduleFlag1; // Checked during IOCTL 0x23EAF0
    BOOLEAN moduleFlag2; // Checked after a payload returns
    ULONG possibleExfilPort; // comes from IOCTL param structure
    PEPROCESS injectedProcess;
} INJECTED_PAYLOADS, *PINJECTED_PAYLOADS;

Figure 18: Updated INJECTED_PAYLOADS structure

To understand how these flags fit together (and tie to our PCAP) we need to analyze the user-mode payloads.

**USER-MODE PAYLOADS**

So far, we have a general idea that man.sys is responsible for loading, managing, and tasking different user-mode payloads. Each payload has a corresponding INJECTED_PAYLOADS structure which contains the process and address where it’s located, flags that dictate how to call that payload, and potentially which port to use to exfiltrate any return data. A summary of the payloads we’ve seen far is summarized in Figure 19.

<table>
<thead>
<tr>
<th>Module ID</th>
<th>Injected Process</th>
<th>Run code address</th>
<th>Exfil port(?)</th>
<th>Flag 1</th>
<th>Flag 2</th>
<th>Notes / Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xDEDEDEDE</td>
<td>netsvcs svchost</td>
<td>0x2101660</td>
<td>Garbage?</td>
<td>Yes</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>0xBEBEBEBE</td>
<td>dwm.exe</td>
<td>0xD72100</td>
<td>Garbage?</td>
<td>No</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>0xDEFA8474</td>
<td>procexp64.exe</td>
<td>0x2934090</td>
<td>6666</td>
<td>No</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>0xDEDA474</td>
<td>explorer.exe</td>
<td>0x2412Ab0</td>
<td>7777</td>
<td>No</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>0xFABADADA</td>
<td>explorer.exe</td>
<td>0x2403810</td>
<td>8888</td>
<td>No</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>?</td>
<td>netsvcs svchost</td>
<td></td>
<td>Garbage?</td>
<td>No</td>
<td>No</td>
<td>Injected in DriverEntry</td>
</tr>
</tbody>
</table>

Figure 19: Summary of loaded user-mode payloads

Starting with payloads we’ve already seen during analysis, let’s begin with the DLL that’s injected during DriverEntry. This DLL is not encrypted and is stored in man.sys’ .data section so
we don’t need to hunt for it.

CONTROL DLL

Note that in memory the PE header is clobbered as we’ve seen a couple times already, but if we write the DLL from the driver’s .data section and open it in IDA we can see that this PE only contains one export and the only imports are standard C runtime functions. This is a pattern that we’ll see with the rest of the user-mode payloads and for reasons that will soon become clear we can effectively analyze these DLLs as shellcode without worrying about rebuilding the PE header.

For this analysis I’ll reference the DLL embedded in the driver’s .data section (md5: 45a1bba04a93500c24010017d738e040) loaded at 0x180000000. The address that is executed is offset 0x3F80 which is also the DLL’s only export.

This DLL listens on port 4444. We also see the same RC4 string encryption that we saw in man.sys to encrypt WS2_32 API names. This DLL also has another pattern that we’ll see throughout the user-mode payloads. The function at 0x18000424A has the prototype shown in Figure 20.

```
ULONG_PTR resolveAndCallFunc(HMODULE targetDll, char *import, ULONG_PTR numParams, ...);
```

**Figure 20**: Dynamic import lookup function

This function uses the first two parameters to resolve an API via in-memory export parsing and then call the original function, passing the rest of the parameters.

The function at 0x180002BD0 is responsible for receiving a command from the network and sending a corresponding command to the driver. Many of these commands end up at the function at 0x180001A80 which sends an IOCTL to a driver. The full mapping of the network commands is shown in Figure 21.
<table>
<thead>
<tr>
<th>Network Command ID</th>
<th>Result</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xCD762BFA</td>
<td>Inject Payload DLL</td>
<td>Flags aren’t set</td>
</tr>
<tr>
<td>0x34B30F38</td>
<td>Inject Payload DLL</td>
<td>Flag 1 is TRUE</td>
</tr>
<tr>
<td>0x8168AD41</td>
<td>Inject Payload DLL</td>
<td>Flag 2 is TRUE</td>
</tr>
<tr>
<td>0x427906F4</td>
<td>Open handle and send IOCTL to driver</td>
<td></td>
</tr>
<tr>
<td>0xD180DAB5</td>
<td>Send IOCTL 0x23EAF0</td>
<td>Task user mode payload</td>
</tr>
<tr>
<td>0xD44D6B6C</td>
<td>Send IOCTL 0xD44D6B6C</td>
<td>Load a driver payload</td>
</tr>
</tbody>
</table>

**Figure 21: Breakdown of network tasking**

Command 0xD180DAB5 also sends a second IOCTL: after sending the IOCTL to task one of the user-mode payloads and getting the data it then sends IOCTL 0x23EAF0, which from our earlier analysis looks up the INJECTED_PAYLOADS module where the second flag is set.

To summarize, this DLL listens on port 4444 and is responsible for sending IOCTLs to load and task payloads that are managed by man.sys. We’ve also confirmed a pattern where INJECTED_PAYLOADS where flags are set are used in combination with IOCTL 0x23EAF0 when tasking user-mode payloads.

**FLAG 1 – SVCHOST.EXE DLL**

This DLL is loaded at address 0x00d70000 inside svchost.exe (PID 876). This DLL is very straightforward – it contains a single export which corresponds to the offset stored in the INJECTED_PAYLOADS list. It uses the same RC4-based string encryption and the same function to dynamically resolve and call functions.

This DLL is responsible for sending data over the network. The host IP to send to is hardcoded to 192.168.1.243 which we can see in our PCAP. The export for this DLL takes the following structure as a parameter (Figure 22):

```c
typedef struct _NETWORK_EXFIL
{
    ULONG port;
    ULONG dataLength;
    PVOID exfilData;
} NETWORK_EXFIL, *PNetwork_EXFIL;
```

**Figure 22: Structure used to exfil data**

Tracing back through IOCTL 0x23EAF0 in man.sys we can see that the port to exfil through comes from the INJECTED_PAYLOADS structure for the module, confirming our guess from earlier.
Now let’s look at the other flag in our INJECTED_PAYLOADS structure to fully understand how tasking flows through these payloads.

### FLAG 2 – DWM.EXE DLL

This DLL is loaded at address 0x02100000 inside dwm.exe. This DLL contains a single export which corresponds to the offset stored in the INJECTED_PAYLOADS list. It uses the same RC4-based string encryption and the same function to dynamically resolve and call functions.

This DLL is responsible for encrypting data. It takes the following structure as a parameter (Figure 23):

```c
typedef struct _CRYPTO_STRUCT
{
    ULONG inDataLength;
    PVOID inData;
    ULONG outDataLength;
    PVOID outData;
} CRYPTO_STRUCT, *PCRYPTO_STRUCT;
```

**Figure 23:** Structure used to encrypt data

The algorithm to encrypt data passed to this routine is as follows:

- Resolve and call RtlGetCompressionWorkSpaceSize and RtlCompressBuffer on inData
- Call GetUserName
- Use the username as an RC4 key on the compressed data
- Store the result in outDataLength and outData

We can extract the username by attaching to dwm.exe and using the !envvar command, as shown in Figure 24.

```
kd> .process /p /r 0xfffffa80035fd060
Implicit process is now ffffffa80`035fd060
kd> !envvar USERNAME
    USERNAME = FLARE ON 2019
```

**Figure 24:** Extracting the username from the crash dump

Note that the NULL byte is included in the RC4 key.

Finally, remember that this DLL is called as part of IOCTL 0x22F378 in man.sys if the target user-mode payload sets a Boolean in the structure from Figure 17. In other words, each payload
decides if the data it returns should be encrypted using the payload in dwm.exe.

We now have a full understanding of what the INJECTED_PAYLOADS structure looks like, as shown in Figure 25.

```c
typedef struct _INJECTED_PAYLOADS
{
    ULONG moduleId;
    LIST_ENTRY link;
    KMUTEX payloadMutex;
    ULONG_PTR baseAddress;
    ULONG_PTR runOffset;
    ULONG_PTR loadedDllSize;
    BOOLEAN isExfilDll;   // Sends data to 192.168.1.243
    BOOLEAN isCryptoDll; // Used to encrypt some payloads’ return
data
    ULONG exfilPort;     // Set per payload at injection time
    PEPROCESS injectedProcess;
} INJECTED_PAYLOADS, *PINJECTED_PAYLOADS;
```

**Figure 25:** Full INJECTED_PAYLOADS structure

We also have a general idea of how payloads are tasked in this framework:

1. Control DLL receives command 0xD180DAB5 over the network
2. Control DLL sends IOCTL 0x22F378 telling man.sys to task a user-mode payload
3. man.sys finds the correct payload by moduleId and creates a user-mode thread
   a. The payload is decrypted only while the task is running
4. The target payload specifies whether its data should be encrypted
   a. If so, man.sys calls the payload in dwm.exe to encrypt the data
5. The data is returned to the Control DLL
6. The Control DLL sends IOCTL 0x23EAF0
7. man.sys finds the exfil payload in PID 876 which sends the data

This is shown in Figure 26.
Now that we have a general idea of how tasking flows through these payloads let’s look at what capabilities our samples provide. There are three total payloads we need to analyze – we’ll reference them by their exfil port for this analysis. We’ve already seen the tricks that these payloads use so this next analysis is straightforward.

### USER MODE PAYLOAD 1 – PORT 6666

This payload is injected inside procexp64.exe (PID 2508) and loaded at address 0x02930000. The run tasking offset in the INJECTED_PAYLOADS structure is 0x4090.

This payload contains the same RC4-based string encoding and dynamic function resolution present in other payloads. Note that as before we can rebuild the PE header but the use of dynamic function resolution means we can also analyze this DLL as a shellcode payload.

The function that corresponds to offset 0x4090 takes the CMD_HEADER structure from Figure 17 as a parameter. This payload contains functionality to interact with the file system and accepts five different commands. Note that some of these commands instruct man.sys to encrypt the data returned using the payload in dwm.exe.

A summary of this payload is shown in Figure 27.
### User Mode Payload 2 – Port 7777

This payload is injected inside explorer.exe (PID 1124) and loaded at address 0x02410000. The run tasking offset in the INJECTED_PAYLOADS structure is 0x2AB0.

This payload does not take command codes like the previous payload. This payload contains the same RC4-based string encoding and dynamic function resolution present in other payloads. Note that as before we can rebuild the PE header but the use of dynamic function resolution means we can also analyze this DLL as a shellcode payload.

This payload is responsible for taking a single screenshot and returning it as a bitmap. This screenshot is not encrypted using the payload in dwm.exe.

### User Mode Payload 3 – Port 8888

This payload is injected inside explorer.exe (PID 1124) and loaded at address 0x02400000. The run tasking offset in the INJECTED_PAYLOADS structure is 0x3810.

This payload does not take command codes like file payload. This payload contains the same RC4-based string encoding and dynamic function resolution present in other payloads. Note that as before we can rebuild the PE header but the use of dynamic function resolution means we can also analyze this DLL as a shellcode payload.

This payload implements keylogging functionality. There is a single input parameter – how long, in milliseconds, to perform keylogging. The payload also keeps track of which window is active while keylogging and will prepend the keylogger output with the name of the window where keystrokes were recorded.

We now know all the pieces to complete our payload table from earlier (Figure 28):
At this point it appears we have all the pieces to decrypt the PCAP – once we decrypt the data returned by these payloads we should be able to hunt for the flag. However, quickly glancing at the PCAP we’ll notice that we still can’t decrypt the payloads. The most obvious clue that something is wrong is that none of the streams coming over port 4444 contain the command codes we saw during our earlier analysis. We’re still missing a piece to decrypt the data in the PCAP.

CRASH TRIAGE REVISITED

We saw all the code responsible for communicating over port 4444 when we analyzed the control DLL. We can also quickly confirm that neither the driver nor the user-mode payload performed any hooking prior to executing the control payload.

This means that any transformation of network traffic is probably happening somewhere in kernel mode. Recall that the crash was due to improperly loading a kernel driver, and that in the control DLL we saw code to send IOCTLs to an arbitrary device.

There’s two main WinDbg commands we can use to attempt to find the missing payload. All allocations performed by `man.sys` use `ExAllocatePoolWithTag` with the tag FLAR. We can use `!poolfind` to see what kind of allocations are present at the time of the crash (Figure 29).
kd> !poolfind FLAR

Scanning large pool allocation table for tag 0x52414c46 (FLAR)
(fffffa80028ce000 : fffffff800298e000)

fffffa80042d0000 : tag FLAR, size 0xf000, Nonpaged pool
fffffa8001eddc70 : tag FLAR, size 0x90, Nonpaged pool

Searching nonpaged pool (fffffa8001802000 : fffffff805f000000) for tag 0x52414c46 (FLAR)

Figure 29: Finding allocations by man.sys

Dumping and searching this memory reveals some strings that could potentially belong to a PE. As expected, the loaded headers have been overwritten. To figure out where the payload is, remember that as part of loading kernel payloads man.sys will create a DRIVER_OBJECT for the new PE (Figure 30). We can revisit our !object output from Figure 2 to track down the other payload.

kd> !object \Driver

Object: fffffff8a000075060  Type: (ffffff8a0001846a30) Directory
ObjectHeader: fffffff8a000075030 (new version)
HandleCount: 0  PointerCount: 109
Directory Object: fffffff8a000004720  Name: Driver

<table>
<thead>
<tr>
<th>Hash Address</th>
<th>Type</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>fffffff80018841c0 Driver</td>
<td>vdrvroot</td>
</tr>
<tr>
<td></td>
<td>&lt; ... &gt;</td>
<td></td>
</tr>
<tr>
<td>fffffff8001e1f840 Driver</td>
<td>Null</td>
<td></td>
</tr>
<tr>
<td>fffffff800428d9a0 Driver</td>
<td>FLARE_Loaded_0</td>
<td></td>
</tr>
<tr>
<td>13 fffffff80036ab5c0 Driver</td>
<td>FLARE_Loaded_1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt; ... &gt;</td>
<td></td>
</tr>
</tbody>
</table>

Figure 30: Drivers loaded by man.sys

Dumping FLARE_LOADED_1 will show what appears to be an incomplete object, which makes sense given that the system crashed in the middle of loading a kernel payload. FLARE_LOADED_0 is a payload that was previously loaded and is our best guess for what’s modifying the network traffic on the system. The DRIVER_OBJECT contains where the driver is loaded as well as several entry points we can use to help IDA process the file (Figure 31).
As before, the PE header has been overwritten. A quick scan through the binary will show that this driver is using imported functions rather than the runtime lookup seen in the user-mode payloads so it would speed analysis to repeat the PE-building process we did with man.sys. The sections in the original compiled binary are given in Figure 32.
A quick look through the binary will show it using multiple functions imported from fwpkc1nt.sys. This driver implements Windows Filtering Platform (WFP) APIs to modify network traffic on the infected system. Note that for the next section we will use addresses relative to where the driver was loaded, `0xffffffa80`042d0000.

### WINDOWS FILTERING PLATFORM (WFP) PAYLOAD

The Windows Filtering Platform was added in Vista and allows drivers to install callbacks at different layers of the network stack. At a high level, the WFP framework consists of sublayers and filters. Each sublayer is a collection of filters, and a filter corresponds to a direction and layer in the networking stack. For example, a filter could be responsible for parsing outbound TCP stream data or connections at the IP layer. When a packet is being transmitted or received by the system, the OS iterates through each sublayer (weighed by priority) and consults each filter in that sublayer. Filters are responsible for parsing the packet data available to their layer (i.e. packet headers, stream data, etc.) and deciding whether a packet should be allowed or blocked. Once any filter decides if a packet should be allowed or dropped the OS stops iterating through sublayers.

Filter layers in this framework:

- Can process inbound traffic, outbound traffic, or both
- Can block or allow all or part of a packet
- Can modify a packet prior to allowing it
- Can inject entirely new data into the network stream

A full list and description of filter layers is available on MSDN89.

---


This driver consists of primarily two workflows: receiving its configuration via an IOCTL, and registering network callbacks. WFP drivers can be tedious to analyze the first time: there are multiple APIs that can to be called to access packet data, most WFP APIs use a large number of nested structures as parameters, and APIs are connected to each other either using GUIDs passed as parameters or IDs generated by WFP API calls. To limit crawling through architecture documentation we’ll start with the IOCTL handler.

This driver uses the WDF framework, but the function at ffffffff`042d5f44 creates another DEVICE_OBJECT with the name FLND which is used to handle IOCTLs. The processing function for IRP_MJ_DEVICE_CONTROL is at ffffffff`042d5d38.

The function at ffffffff`042d5b0c takes the DEVICE_OBJECT and calls the appropriate WFP functions to install network callbacks. We will call this function add_layers. FwpmSubLayerAdd creates a new sublayer for every DeviceIoControl call.

Installing a filter consists of two APIs: FwpsCalloutRegister and FwpmFilterAdd. The FwpsCalloutRegister takes a FWPS_CALLOUT structure which specifies which functions to call for a filter. Note that both the API and the structure have different versions depending on the OS version. For our analysis we only need to focus on the classifyFn function. For WFP functions this member is the function responsible for processing packet data and deciding whether it should be allowed or blocked. The FwpmFilterAdd function is used to add a filter to a layer. This API takes an array of FWPM_FILTER structures where each specify when a filter function (e.g. direction, target port etc.).

The functions at ffffffff`042d4650 and ffffffff`042d4a30 use these functions to add callouts at different points during a connection. The configuration for the installed filter (e.g. which layer to filter, direction, and port) are passed as parameters to these two functions. These functions install two filters: a FWPM_LAYER_ALE_FLOW_ESTABLISHED_V4 filter which is called when a connection has been established, and a FWPM_LAYER_STREAM_V4 filter which allows the driver to have access to the data sent over a TCP connection. Based on the parameters in the FWPM_FILTER structures, the first filter is used to build a structure associated with a connection over a specific port/direction, and the second filter is the one that is used to access data coming across a TCP connection.

Going back to the add_layers function we can see that all of the information used to add the sublayer and appropriate filters is populated in a structure. Each structure is then inserted into a global list at ffffffff`042d80d0. We’ll call this list callback_list. We can also see that all of the parameters relevant to installing the filters (port, direction, etc.) comes as an input parameter to the IRP_MJ_DEVICE_CONTROL handler.

Associated a stream callout with an inbound connection is done using more indirection in the WFP API. The function at ffffffff`042d5c90 searches the callback_list by ID. This ID is
generated from FwpsCalloutRegister. Once the global structure is found the stream layer ID (FWPS_LAYER_STREAM_V4) and the stream layer callout ID (generated by the second call to FwpsCalloutRegister) are passed to FwpsFlowAssociateContext. This API allows a driver to specify a parameter to be passed to another callout’s (the FWPM_LAYER_STREAM_V4 filter callout in this case) classifyFn. This parameter is yet another structure which contains a pointer to the structure in callback_list. We will call this structure flow_context.

The stream layer’s callout is responsible for accessing data passed over TCP streams. This function has some utility functionality to copy data to and from a network stream but the function we are interested in is at fffffffa80`042d2204. This function has a flow_context structure as a parameter and calls FwpsStreamInjectAsync to inject data into the TCP stream. To see which data was injected we need to look at the function at fffffffa80`042d28c8. This function copies the existing stream data and grabs offset 0x16 from the callback_list structure and uses the next 8 bytes as an XOR key over the old stream data. These bytes also come from the original list insertion in the IRP_MJ_DEVICE_CONTROL handler.

This means that in addition to the possible/optional encryption we saw in the user mode crypto payload in dwm.exe all traffic over ports specified in the global list are also XOR encoded.

We’ve now analyzed enough of the WFP driver to understand how it is modifying traffic. To summarize:

- The control DLL in user mode has a command to send IOCTLS to arbitrary drivers
- When the WFP gets IOCTL 0x13FFFC (the only accepted code) it installs WFP layers to track connections and stream data
- Each IOCTL results in a global structure being allocated, which designates the port and direction of a connection to monitor, as well as an XOR key
- When data is being transmitted or received over one of these ports, the classifyFn for the stream layer callout XORs this data with what’s in the global list
- All data sent to or from the control or plugin DLLs is XOR’d

We now have all the information necessary to decrypt traffic sent to and from these samples. We can dump the callback_list to extract all XOR keys needed to decode the PCAP traffic. A complete list of the plugin table is shown in Figure 33.
<table>
<thead>
<tr>
<th>Payload</th>
<th>Port</th>
<th>Direction</th>
<th>XOR key</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control</td>
<td>4444</td>
<td>In</td>
<td>0x5d 0xf3 0x4a 0x48 0x48 0x48 0xdd 0x23</td>
</tr>
<tr>
<td>File</td>
<td>6666</td>
<td>Out</td>
<td>0xd5 0x69 0x94 0xfa 0x25 0xec 0xdf 0xda</td>
</tr>
<tr>
<td>Screenshot</td>
<td>7777</td>
<td>Out</td>
<td>0x4a 0x1f 0x4b 0x1c 0xb0 0xd8 0x25 0xc7</td>
</tr>
<tr>
<td>Keylog</td>
<td>8888</td>
<td>Out</td>
<td>0xf7 0x8f 0x78 0x48 0x47 0x1a 0x44 0x9c</td>
</tr>
</tbody>
</table>

**Figure 33: Summary of WFP keys for user-mode plugins**

**PLUGIN DATA**

Most of the data in the PCAP is tasking the keylogger and screenshot plugins. Looking through the screenshots we see some innocuous system usage, followed by the infected user opening KeePass. Looking at Figure 34, we see that the flag we need is in the database.

**Figure 34: The flag we need to solve the challenge is hinted in one of the screenshots**
We can also see that after running a find-like command to locate it, the only file that is pulled from the infected system is the KeePass database, keys.kbd. KeePass databases are protected with a master password, but since we have traffic from a keylogging plugin we can grab the password from this. Looking through the keylogger output we can find something that looks like a password: th1s1sth33nd111. However, if we try this password on the decrypted KeePass database (MD5: c36a854510bb52933714e2dc2871aeb8) it fails to decrypt.

Without any other clues we need to revisit the output data from the plugins. We’ve also already seen in both the help.txt file and the fact that this challenge started with a BSOD that the samples on this system have bugs and other logic flaws. In fact, if we compare the screenshot output to the keylogger output we’ll see some discrepancies. Some of these discrepancies can be seen when comparing Figure 35 and Figure 36.

Figure 35: Sample keylogger output
Comparing the output between screenshots and keyloggers, we’ll see that the keylogger does not handle punctuation and capitalization (for letters or special characters). This means that the correct master password to the KeePass database is a mutation of th1s1sth33nd111 with the insertion of punctuation, likely underscores to keep with the Flare-On key formats.

This means that our only option is to attempt to use the data we know as a basis to write a minimal brute force for the password. For this version of KeePass we can use a tool like John the Ripper or Hashcat to perform the brute force. To put a feasible boundary on the brute force we can generate a custom key list using our known seed password and possible character combinations. One possible script to do this is shown in Figure 37.

---

from itertools import product

ccharset = {
    'd': ['d', 'D'],
    'e': ['e', 'E'],
    '3': ['3', '#'],
    'h': ['h', 'H'],
    'i': ['i', 'I'],
    'n': ['n', 'N'],
    's': ['s', 'S'],
    't': ['t', 'T'],
    '1': ['1', '!'],
    ' ': [' ', '_', '-', '']
}

def generate(password):
    letters = []
    for v in password:
        if v in ccharset.keys():
            letters.append(ccharset[v])
        else:
            letters.append(v)
    return [''.join(item) for item in product(*letters)]

with open('wordlist', 'wb') as f:
    wordlist = generate('th1s is th3 3nd111')
    f.write(wordlist)

Figure 37: Sample script to generate a keylist to brute force our KeePass database

Once this keylist is generated, our brute force will give us the correct master password: Th1s_iS_th3_3Nd!!!. Using this to open the database will give us the final flag: f0110w_th3_br34dcrumbs@flare-on.com