Flare-On 10 Challenge 13: y0da

By Mark Lechtik (@_marklech_)

Overview

`y0da.exe` revolves around an obfuscation technique applied on x64 shellcode. In fact, it's based on a real-world malware dubbed JUMPLUMP where this obfuscation was used. JUMPLUMP comes in the form of a trojanized DLL that gets infected by another malicious component named CORELUMP. The latter takes a pre-defined piece of shellcode, breaks it down into smaller pieces and incorporates them into one of several benign DLLs in the `%SYSTEM%` directory.
The premise of this obfuscation is that the smaller chunks of shellcode are chained together by unconditional jumps and end up being scattered across the `.text` section. Since they are also intertwined with other instructions from the original DLL, the disassembly of the shellcode gets broken, and its flow becomes hard to track.

Tools

There may be more than one way to solve this challenge, and therefore a variety of tools could be leveraged depending on the solution method. Nonetheless, here are the ones that I used:

- **IDA Disassembler & Hexrays Decompiler for x64**: our main task would be to clean up the disassembly of y0da.exe, thereby producing readable decompiled code.
- **IDAPython**: to clean up the disassembly we'll need to use some IDA based automation.
- **Time Travel Debugging (TTD) & WinDbg**: while not mandatory for solving the challenge, I find that the TTD provided capabilities are often useful in dealing with complex binaries. It occurs to me that it isn't being used widely by malware REs, which is why I think it warrants a basic introduction that you can find here. A more thorough walkthrough of it by Christophe Alladoum can be found here. Also, I would recommend Yarden Shafir's two part blog post on WinDbg's debugger data model that is full of interesting use cases and demos of how to harness some of its power (can be found here and here)
  TTD is facilitated by several components shipped with Windows 10 installations, one of which is tttracer.exe, typically found in the %SYSTEM% directory. This component allows capturing a trace of execution that can then be replayed in WinDbg. To do so, you can simply run tttracer.exe specifying the target executable to trace as an argument.
  The resulting .run file is the trace that can then be opened in WinDbg, allowing us to step through code forwards and backwards, query the debugger data model for useful run-time information and introduce JavaScript based automation to glean more insights from the recorded session.
- **FLOSS v2.0**: as you'll see, the challenge is sprinkled with stack-based strings and several encoded ones. I find FLOSS helpful in quickly resolving those and allowing me to navigate through them while seeing the addresses of the functions in which they were found.
- **CAPA**: being able to understand which common algorithms are found in code can help getting a better sense of an analyzed sample's inner workings. CAPA makes it especially useful in this binary, in

the face of obfuscation that makes it challenging to sift through code and manually find artifacts of known algorithms.

- **Windows 10 based FLARE VM**: older versions of Windows (e.g., Windows 7 x64) can also be used, but in the interest of using TTD, I used Windows 10. Any other common analysis tools are provided with the installation of FLARE VM.

Challenge Walkthrough

Basic Static Analysis

First let's examine static strings that we see in the binary using FLOSS and the command line:

`floss.exe -n 5 --only static -v y0da.exe`.

A partial output would look as follows:

```
--------------------------------
| FLOSS UTF-16LE STRINGS (244) |
--------------------------------
0x06bc10 ExtendViewIntoTitleBar
0x06bc70 Windows.UI.Composition.Compositor
0x06bd08 \EXPLORER.EXE
0x06bd30 {9993b795-bae5-4752-a837-faa5985c808d}
0x06bd80 {b1bb96b0-d47d-4572-9bbd-3fe3e72e8191}
0x06bdd0 {bd3fd375-f43d-4fc4-a690-11c64aaa0209}
0x06be20 Actions
0x06be78 EnterFullScreen
0x06bea8 Local\SM0:%d:%d:%hs
0x06bfb0 ntdll.dll
0x06bfe0 {24CF051B-7E9E-4730-9B52-78E64066EE6A}
0x06c050 UnknownAppFrame
0x06c070 UIA_WindowPatternEnabled
0x06c0a8 UIA_HasOwnNonClientUIATree
0x06c0e0 ApplicationFrameWindow
0x06c140 ApplicationFrameRtlTitleBar
0x06c178 ApplicationFrameTitleBarWindow
0x06c1c0 ApplicationFrameInputSinkWindow
0x06c238 Software\Microsoft\Windows\DWM
```

Figure 1: Static strings found in y0da.exe's binary

There isn't much helpful information here, except for the fact that most of these strings match a benign Windows DLL named `ApplicationFrame.dll` (SHA256: 8FA35F1694595AA5B92E67A1105AF4CC04703DFBE06E12088E68828C46F99569) that is typically found in the `%SYSTEM%` directory. In fact, the string `\System32\ApplicationFrame.dll` can be found within it as well.

A more useful run would entail analyzing stack strings (with a minimal length of 5 characters) using the command "`floss.exe -n 5 --only stack -v y0da.exe`". Here's a partial output of it:

```
----------------------------
| FLOSS STACK STRINGS (100) |
----------------------------
Function      Function Offset    Frame Offset    String
-----------   ----------------   -------------   ---------------------------------------------------------------------------------------
0x1800216f6  0x180014d24        0x224           '-._'7'
0x1800216f6  0x180014d24        0x1f1           To know the secret, you want?
0x1800216f6  0x180014d24        0x1d2           Me the password, give:
0x1800216f6  0x180014d24        0x1af           _.-'`-._
0x1800216f6  0x180014d24        0x12e           |/  \/  \|
0x1800216f6  0x180014d24        0x113           /  ()  ()  \
0x1800216f6  0x180014d24        0xf7            / \   ()   / \
0x1800216f6  0x180014d24        0xda               /   \/||\/   \
0x1800216f6  0x180014d24        0xbc              /\/\/\
0x1800216f6  0x180014d24        0x9d             /\  /\  /\
0x1800216f6  0x180014d24        0x7d            /_\oo/_\
0x1800216f6  0x180014d24        0x5c              `--...__|`-._  _.-'|__...--'
0x1800216f6  0x180014d24        0x21            \t\t  NO.
0x1800216f6  0x1800523bb        0x214              /'.-c
0x180026aea  0x180014d24        0x11            [+] Listening...
0x18003a5e1  0x18001d430        0x34            M4ST3R
0x18003c5e2  0x180036162        0x338           "No! Try not. Do. Or do not. There is no try."
0x18003c5e2  0x180036162        0x220           "Size of code matters not.
0x18003c5e2  0x18000254b        0x3b0           Y0da's life
0x18003c5e2  0x18000254b        0x390           "The greatest teacher failure is."
0x18003c5e2  0x18000254b        0x368           "Won this job in a raffle I did, think you?"
0x18003c5e2  0x18000254b        0x308           "Fear of malware is the path to the dark side."
0x18003c5e2  0x18000254b        0x2d8           "Truly wonderful the mind of a reverse engineer is."
0x18003c5e2  0x18000254b        0x2a0           "Packers, crypters, shellcode. The dark side are they."
0x18003c5e2  0x18000254b        0x268           "A Jedi's strength flows from their knowledge of assembly."
0x18003c5e2  0x18000254b        0x220           "Size of code matters not. Look at me. Judge me by my size, do you?"
0x18003c5e2  0x18000254b        0x1d0           "A Jedi uses the Force for knowledge and defense, never for attack."
0x18003c5e2  0x18000254b        0x180           "Train yourself to let go of the decompiler you are afraid to lose."
0x18003c5e2  0x18000254b        0x130           "Obfuscation leads to anger. Anger leads to hate. Hate leads to suffering."
0x18003c5e2  0x18000254b        0xe0            "If no mistake you have made, losing you are. A different game you should play."
0x18003c5e2  0x180014d24        0x3b0           Y0da's life tip #0x%x:
0x18003c5e2  0x180001923        0x3b0           Y0da's life tip
0x18003cede  0x180014d24        0xca0           ComSpec
0x18003cede  0x180014d24        0xc98           ws2_32.dll
0x18003cede  0x180014d24        0xc88           user32.dll
0x18003cede  0x180014d24        0xc78           SYSTEMROOT
```

Figure 2: Stack strings found in y0da.exe by FLOSS

We see that most of the interesting strings appear as stack strings, including ASCII art and ones that indicate the user must provide a password to get a secret. In addition, we can look for any encoded strings using the command line "`floss.exe -n 5 --only decoded -v y0da.exe`" where we'll find the following curious string that's worth keeping in mind:

```
--------------------------------
|  FUNCTION at 0x1800559b0 (1) |
--------------------------------
Offset     Called At     String
--------   -----------   --------------------------------
(STACK)    0x1800231fd   Q4T23aSwLnUgHPOIfyKBJVM5+DXZC/Re=
```

Figure 3: String decoded by FLOSS

We will run CAPA to try and identify any algorithms used by `y0da.exe`. The output indicates that MD5 and Mersenne Twister are leveraged:

```
hash data with MD5
namespace  data-manipulation/hashing/md5
scope      function
matches    0x1800126AB

generate random numbers using a Mersenne Twister (6 matches)
namespace  data-manipulation/prng/mersenne
scope      function
matches    0x180020404
           0x180020C60
           0x180025980
           0x18002B450
           0x180058580
           0x180063054
```

Figure 4: CAPA detecting the usage of MD5 and Mersenne Twister by y0da.exe

If we inspect the headers of the executable, we'll see that the import directory is blank, which is indicative of the fact that usage of any Windows API functions requires their underlying modules to be loaded by the code itself and their addresses to be resolved dynamically during run-time:

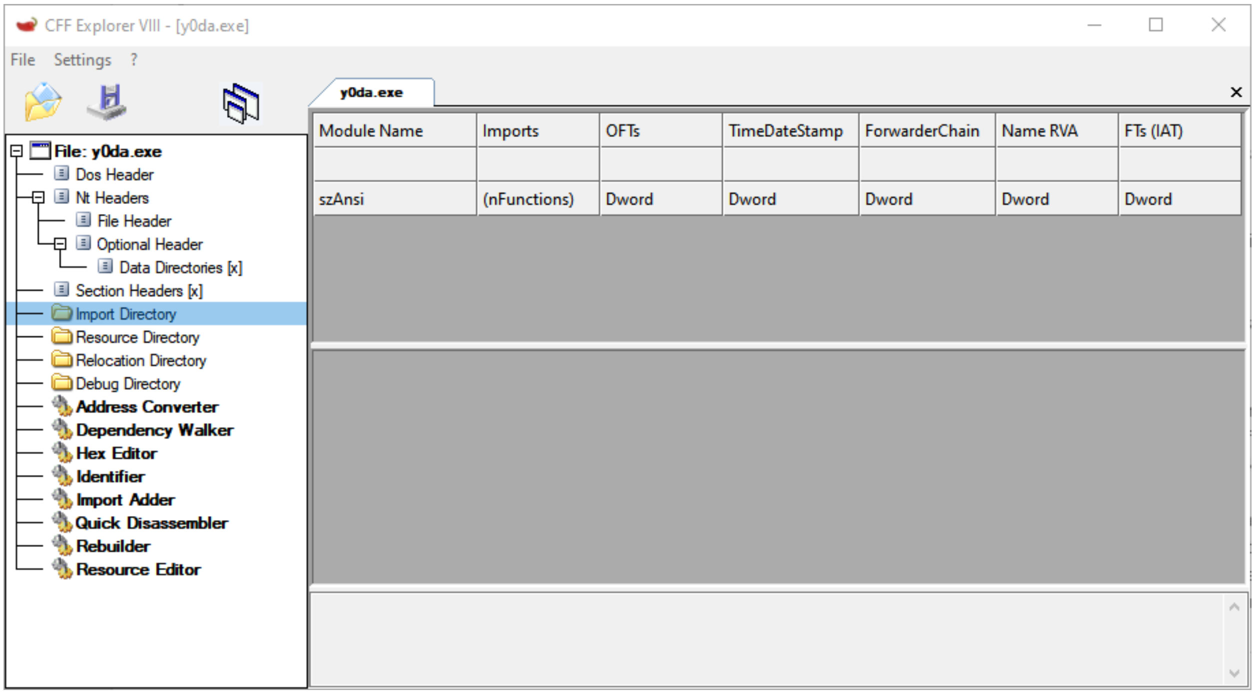

Figure 5: Blank import table in y0da.exe

In addition, a glance at the resource directory with a tool like Resource Hacker shows an unusual resource named `Y0D4` of type `M4ST3R` which contains a high-entropy binary blob that we may also want to keep in mind going forward:

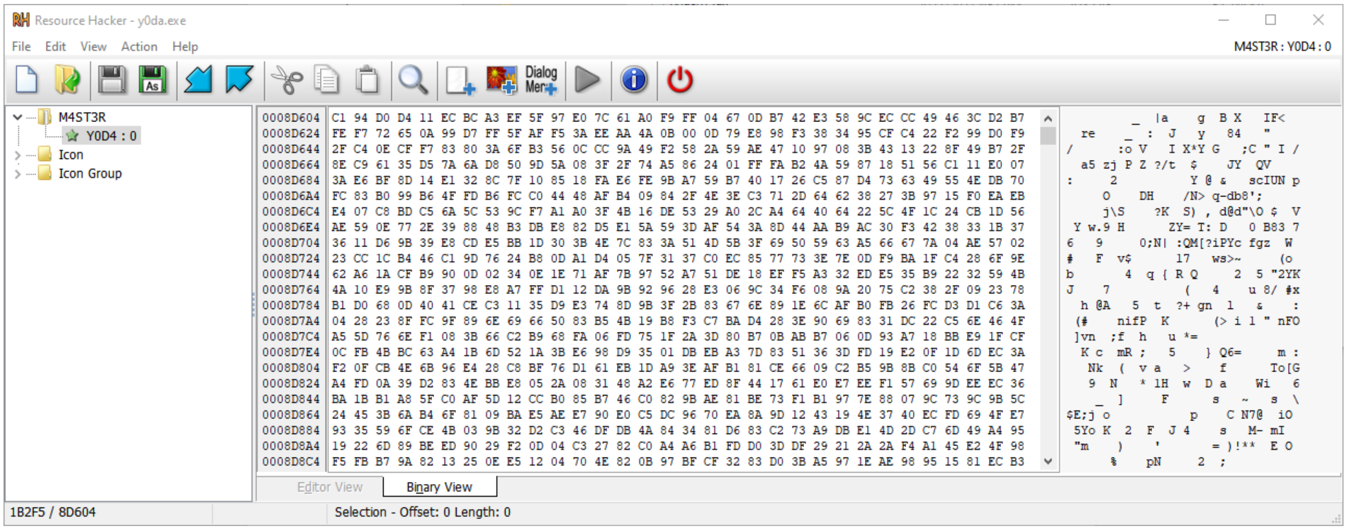

Figure 6: Binary blob found as a resource named Y0D4

Basic Dynamic Analysis

When we run the executable, we'll see a FLARE banner followed by a string that indicates that the process is listening, likely for incoming connections.

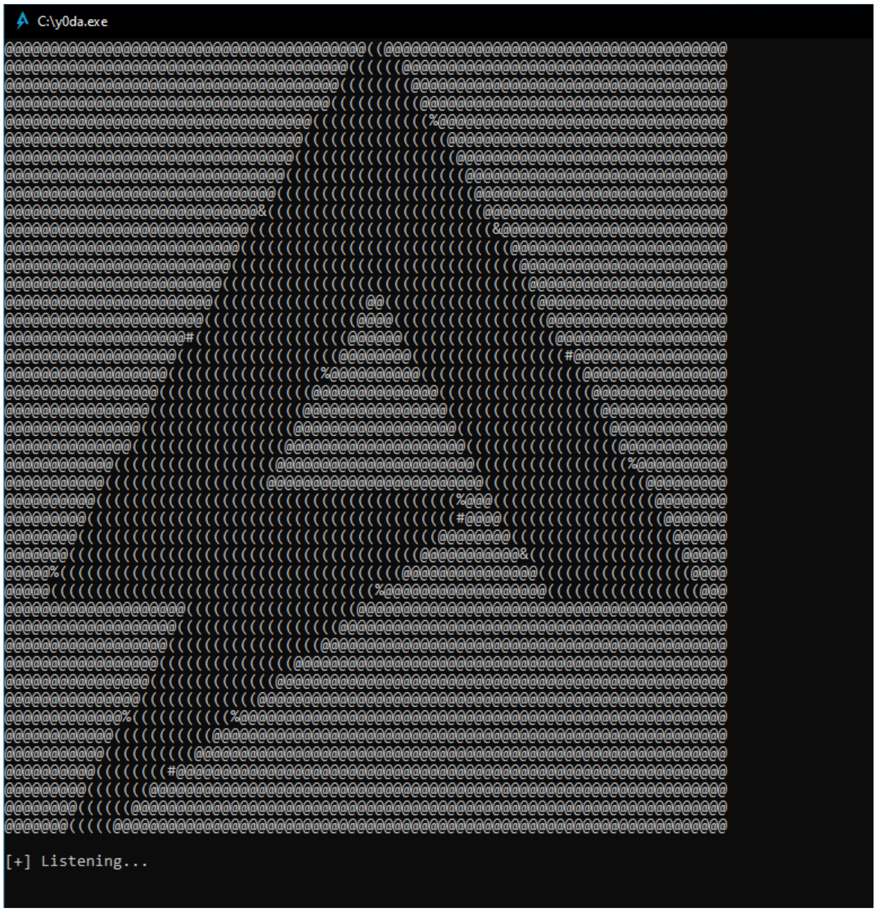

Figure 7: Console output when running y0da.exe

We can easily corroborate this by checking if there are any connections associated with the `y0da.exe` process in TCPView and see that the process is indeed listening on TCP port 1337.

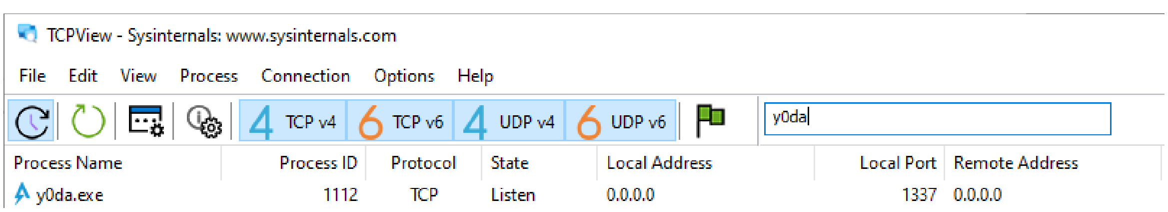

Figure 8: Output of of TCPView when running y0da.exe

If we connect to this port via netcat on the localhost we get a Yoda banner, followed by what appears to be a standard cmd.exe shell.

Figure 9: Shell output when connecting to the server run by y0da.exe on port 1337

Looking at the process tree that is generated as a result of executing `y0da.exe` in Process Hacker, we observe that it spawns a `cmd.exe` process, which we can assume is the shell that we are controlling via netcat.

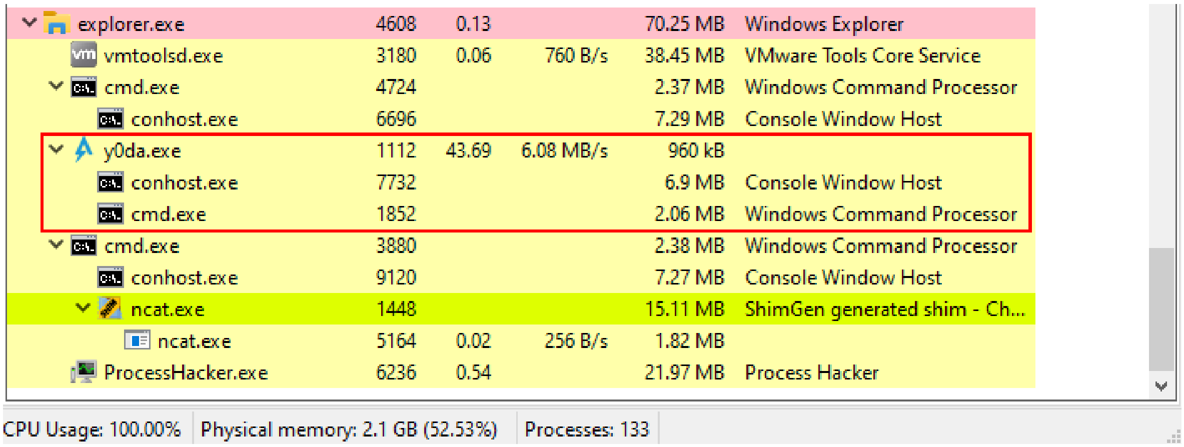

Figure 10: Process tree created as a result of running y0da.exe

If we switch to the Threads tab we'll see 3 active threads, one of which is the main thread, the code of which starts at 180032701, the executable's entry point.

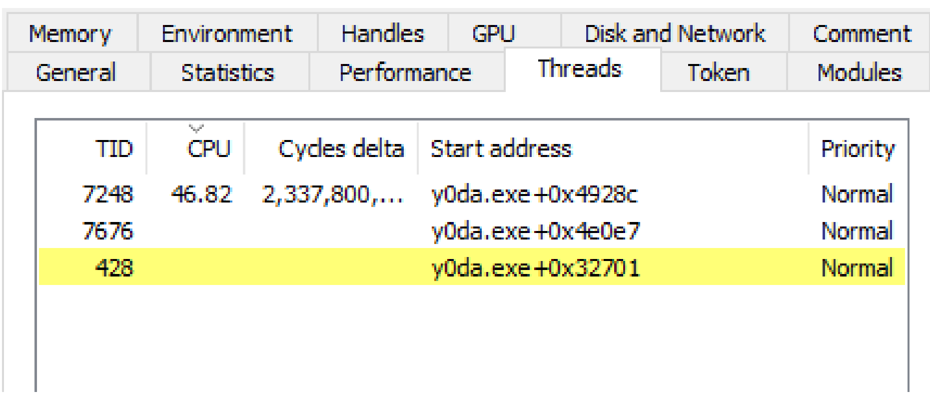

Figure 11: Threads run in the context of y0da.exe's process

Looking into the stack of the thread that corresponds to the function 18004e0e7 we see that it invokes a `recv` API call, which we can assess is related to receiving input for the shell via the established TCP connection.

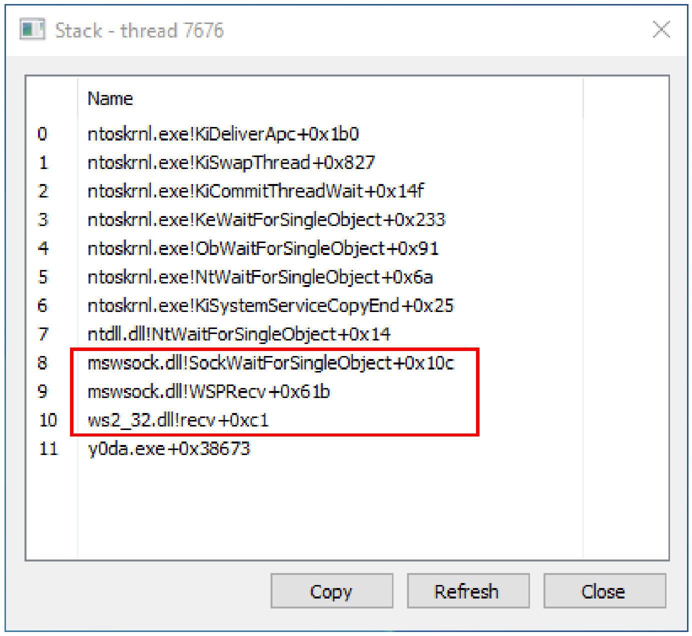

Figure 12: Stack trace of the thread at address 0x18004e0e7

The other thread that corresponds to the function 18004928c calls `PeekNamedPipe` which would be consistent with reading the shell's output via a pipe before sending it back on the TCP channel.

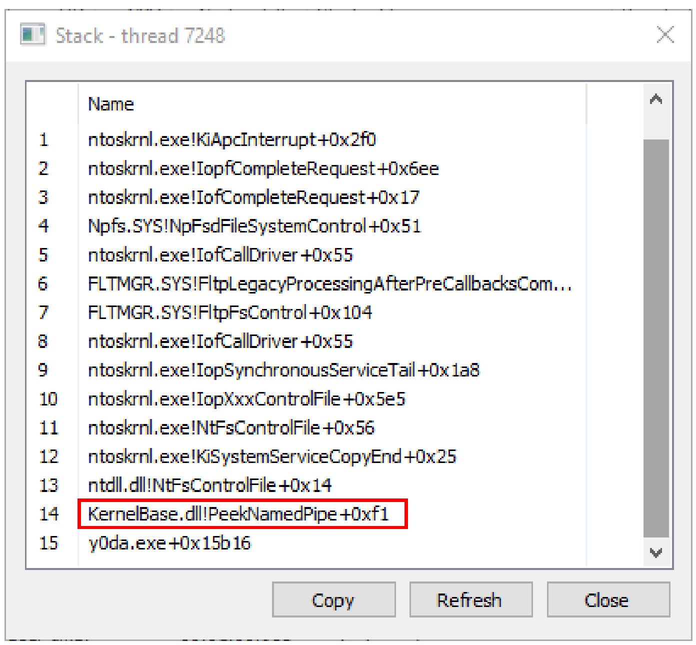

Figure 13: Stack trace of the thread at address 0x18004928c

All the above behavior would be consistent with an implementation of a Windows bind shell, as outlined in the figure below. That said, we still need to figure out how to interact with it so that we get the flag.

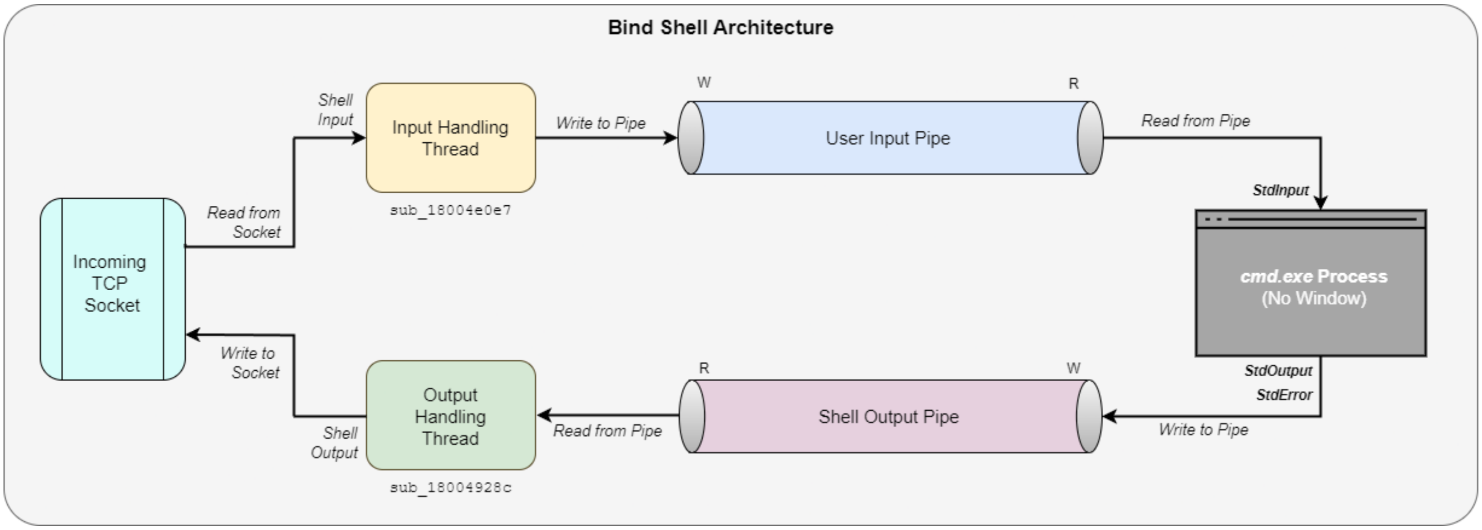

Figure 14: Bind Shell Architecture

Code Obfuscation: Problem Statement

Looking into the disassembly of `y0da.exe` in IDA we can spot an obfuscation pattern wherein the code consists of small basic blocks that consist of two instructions at most. A typical basic block consists of an instruction, followed by an unconditional jump that leads to the next basic block of the shellcode. The exceptions to this rule are basic blocks that consist of a single unconditional jump or `retn` instruction. To demonstrate how this obfuscation works, consider the following simple piece of code:

```
start:
        sub     rsp, 28h
        call    sub_140012610
        add     rsp, 28h
        retn
;...
;<code>
;...
sub_140012610:
        push    rsi
        mov     rsi, rsp
        and     rsp, 0FFFFFFFFFFFFFFF0h
        sub     rsp, 20h
        call    sub_140001010
        mov     rsp, rsi
        pop     rsi
        retn
```

After obfuscation, it will look like this:

```
loc_180004DED:
        mov     rsp, rsi
        jmp     loc_18001DDA2
;...
;<junk\other code>
;...
sub_18000853F:
        push    rsi
        jmp     loc_18005A0B8
;...
;<junk\other code>
;...
loc_180017663:
        retn
;...
;<junk\other code>
;...
loc_18001DDA2:
        pop     rsi
        jmp     loc_180017663
;...
;<junk\other code>
;...
loc_1800251F2:
        and     rsp, 0FFFFFFFFFFFFFFF0h
        jmp     loc_180034E4F
;...
;<junk\other code>
;...
loc_1800136A2:
        call    sub_18000853F
        jmp     loc_1800340C9
;...
;<junk\other code>
;...
loc_180016D82:
        retn
;...
;<junk\other code>
;...
start:
        sub     rsp, 28h
        jmp     loc_1800136A2
;...
;<junk\other code>
;...
loc_1800340C9:
        add     rsp, 28h
        jmp     loc_180016D82
;...
;<junk\other code>
;...
loc_180034E4F
        sub     rsp, 20h
        jmp     loc_18005DFEF
;...
;<junk\other code>
;...
loc_18005A0B8:
        mov     rsi, rsp
        jmp     loc_1800251F2
;...
;<junk\other code>
;...
loc_18005DFEF:
        call    sub_18003CEDE
```

```
        jmp     loc_180004DED
```

This form of obfuscation thwarts the ability to perform proper static analysis. The biggest hurdle is that IDA can't immediately tell the flow of the shellcode's basic blocks apart from other functions that remain in the executable from the original `ApplicationFrame.dll`. Consequently, we don't know the starting addresses of the real functions that comprise the shellcode, instead we see functions with faulty disassembly of what is essentially leftovers of dead code from the original DLL, mixed with shellcode basic blocks. In the absence of information on shellcode related function addresses and their bounds, we need to step through the basic blocks manually, which proves to be infeasible due to the size of the shellcode.

Inspecting API Calls in the TTD Trace

One of the things that we can do right after recording the trace of `y0da.exe` is to triage it for API calls that were invoked throughout its execution. This is made possible due to WinDbg's Debugger Data Model – a hierarchy of objects that provide debugger extensions with the ability to consume and produce information that can be accessed by the debugger or other extensions of it. One notable set of objects is the TTD Calls Objects that hold information about function calls that occurred over the course of the trace.

Before we look at the calls themselves, it's worth checking the modules that were loaded during run-time. We can do so by stepping to the end of the trace with the command `!tt 100` and inspecting the list of modules with the lm command. We'll also make note of the address range in which `y0da.exe` is mapped.

```
0:000> !tt 100
Setting position to the end of the trace
Setting position: 157951:140
(1228.1888): Break instruction exception - code 80000003 (first/seco
Time Travel Position: 157951:140 [Unindexed] Index
ntdll!NtTerminateProcess+0x12:
00007ffd`dd50d2f2 0f05            syscall
0:000> lm
start             end                 module name
00000001`80000000 00000001`800c9000   y0da       (no symbols)
00007ffd`bdef0000 00007ffd`be0ca000   TTDRecordCPU # (pdb symbols)
00007ffd`d83f0000 00007ffd`d8480000   apphelp    (pdb symbols)
00007ffd`da270000 00007ffd`da2da000   mswsock    (pdb symbols)
00007ffd`dab90000 00007ffd`dac2d000   msvcp_win   (pdb symbols)
00007ffd`dac30000 00007ffd`dad30000   ucrtbase   (pdb symbols)
00007ffd`dadc0000 00007ffd`daecb000   gdi32full # (pdb symbols)
00007ffd`daf30000 00007ffd`db1f9000   KERNELBASE # (pdb symbols)
00007ffd`db250000 00007ffd`db272000   win32u   # (pdb symbols)
00007ffd`dbd60000 00007ffd`dbf01000   user32   # (pdb symbols)
00007ffd`dbfe0000 00007ffd`dc04b000   ws2_32     (pdb symbols)
00007ffd`dc130000 00007ffd`dc25a000   RPCRT4   # (pdb symbols)
00007ffd`dc2d0000 00007ffd`dc38e000   KERNEL32 # (pdb symbols)
00007ffd`dccc0000 00007ffd`dcd5b000   sechost    (pdb symbols)
00007ffd`dce00000 00007ffd`dce2b000   GDI32    # (pdb symbols)
00007ffd`dd2e0000 00007ffd`dd310000   IMM32      (pdb symbols)
00007ffd`dd470000 00007ffd`dd665000   ntdll    # (pdb symbols)
```

Figure 15: Modules loaded during the execution of y0da.exe and the address range of the main module

Now we can run the following query that would give us calls to functions that start with the Create keyword in `kernel32.dll` and any function invoked from `ws2_32.dll` by `y0da.exe`'s code:

```
dx -g @$cursession.TTD.Calls("KERNEL32!Create*", "ws2_32!*").Where(c =>
c.ReturnAddress > 0x180000000 && c.ReturnAddress < 0x1800c9000)
```

The result looks as follows:

(+) ThreadId	(+) UniqueThreadId	(+) TimeStart	(+) TimeEnd	(+) Function	(+) FunctionAddress	(+) ReturnAddress	(+) ReturnValue
0x1888	0x2	152F:666	1C77:1E	ws2_32!WSAStartup	0x7ffddbfeeb10	0x180010ef2	0x0
0x1888	0x2	1CE8:9E	1CEF:14	KERNEL32!CreatePipeStub	0x7ffddc2f0250	0x18002c6ab	0x1
0x1888	0x2	1D60:94	1D64:14	KERNEL32!CreatePipeStub	0x7ffddc2f0250	0x180031522	0x1
0x1888	0x2	1E1B:149B	2020:13	ws2_32!socket	0x7ffddbfe55f0	0x180064ac3	0x144
0x1888	0x2	20D7:1006	20E9:C	ws2_32!bind	0x7ffddbff09c0	0x180002b67	0x0
0x1888	0x2	224F:660	2262:B	ws2_32!listen	0x7ffddbff12a0	0x1800656a1	0x0
0x1888	0x2	251A:1B24	28DA:90	KERNEL32!CreateProcessAStub	0x7ffddc2ec760	0x18004722c	0x1
0x1888	0x2	2991:88A	2997:F	ws2_32!send	0x7ffddbfe2320	0x180057ce7	0x7a9
0x1888	0x2	2997:3A	2999:29	KERNEL32!CreateThreadStub	0x7ffddc2eb5a0	0x180057794	0x160
0x1888	0x2	2999:39	299B:29	KERNEL32!CreateThreadStub	0x7ffddc2eb5a0	0x180062fdc	0x15c
0x1a64	0x6	2D90:1B21	37319:10	ws2_32!recv	0x7ffddbff1d90	0x180038673	0x1
0x15a8	0x5	2DF1:E65	2E02:F	ws2_32!send	0x7ffddbfe2320	0x180057ce7	0x6b
0x15a8	0x5	3785D:1CDA	3786E:F	ws2_32!send	0x7ffddbfe2320	0x180057ce7	0x1
0x1a64	0x6	37A72:19BC	3D3FB:10	ws2_32!recv	0x7ffddbff1d90	0x180038673	0x1
0x15a8	0x5	37B84:209C	37B8A:F	ws2_32!send	0x7ffddbfe2320	0x180057ce7	0xb

Figure 16: Partial list of API functions invoked during the execution of y0da.exe

We can immediately observe that the program creates a process which we've already established is `cmd.exe`. In addition, there are two threads and two pipes created during run-time, and that recv and send operations are made from separate threads. All of this suggests that one thread handles the input passed from the socket to the shell and another handles output passed from the shell to the socket, all done via pipes.

As an example, we can step into the first recv call in the above list by navigating to its corresponding index in the trace, which is 2D90:1B21. What we can do then is step back to the previous frame from which this API function was called using the g-u command, and step another two instructions backwards. What we'll see is that the constant 0x5FC8D02 is passed to r15d. This constant is in fact the name hash of recv according to Metasploit's name hashing algorithm, which alludes to sub_180014d24 being a function that dynamically resolves API function addresses given their name hash.

```
0:004> !ttdext.tt 2D90:1B21
Setting position: 2D90:1B21
(1228.1a64): Break instruction exception - code 80000003 (first/second chance not available)
Time Travel Position: 2D90:1B21 [Unindexed] Index
ws2_32!recv:
00007ffd`dbff1d90 48895c2408      mov     qword ptr [rsp+8],rbx ss:00000000`2993fe60=1bf07db05fc8d902
0:004> g-u
Time Travel Position: 2C17:1300 [Unindexed] Index
y0da+0x3866e:
00000001`8003866e e8b1c6fdff      call    y0da+0x14d24 (00000001`80014d24)
0:004> t-
Time Travel Position: 2C17:12FF [Unindexed] Index
y0da+0x436f1:
00000001`800436f1 e9784fffff      jmp     y0da+0x3866e (00000001`8003866e)
0:004> t-
Time Travel Position: 2C17:12FE [Unindexed] Index
y0da+0x436eb:
00000001`800436eb 41bf02d9c85f    mov     r15d,5FC8D902h
```

Figure 17: Finding recv's name hash passed via r15d by stepping backwards from recv's call in the TTD trace

Navigating a few more instructions backwards, we can see that there's another value that is likely passed via r14:

```
Time Travel Position: 2C17:12FD [Unindexed] Index
y0da+0x2865d:
00000001`8002865d e989b00100      jmp     y0da+0x436eb (00000001`800436eb)
0:004> t-
Time Travel Position: 2C17:12FC [Unindexed] Index
y0da+0x2865a:
00000001`8002865a 4d33f6          xor     r14,r14
0:004> t-
Time Travel Position: 2C17:12FB [Unindexed] Index
y0da+0x319fd:
00000001`800319fd e9586cffff      jmp     y0da+0x2865a (00000001`8002865a)
0:004> t-
Time Travel Position: 2C17:12FA [Unindexed] Index
y0da+0x319fb:
00000001`800319fb 4156            push    r14
```

Figure 18: Finding another argument passed via r14 when invoking recv

Also, we can note that the four arguments of recv itself seem to be passed according to the fastcall convention, i.e., via rcx, rdx, r8 and r9:

```
0:004> dx -r1 @$cursession.TTD.Calls("KERNEL32!Create*", "ws2_32!*").Where(c => c.ReturnAddress > 0x180000000 && c.ReturnAddress < 0x1800c9000)[279].Parameters
@$cursession.TTD.Calls("KERNEL32!Create*", "ws2_32!*").Where(c => c.ReturnAddress > 0x180000000 && c.ReturnAddress < 0x1800c9000)[279].Parameters
    [0x0]            : 0x148
    [0x1]            : 0x29890000
    [0x2]            : 0x4000
    [0x3]            : 0x0
0:004> r rcx
rcx=0000000000000148
0:004> r rdx
rdx=0000000029890000
0:004> r r8
r8=0000000000004000
0:004> r r9
r9=0000000000000000
```

Figure 19: Parameters of the recv API call passed via the fastcall convention

Cleaning-up the Control Flow using IDAPython

To further understand the logic of `y0da.exe` we need to have a look at its code, and that requires doing some fixes to make it readable. Our strategy will be to clean up any dead code that isn't part of the challenge's flow, identify function bounds and designate basic blocks to their corresponding functions. Consequently, Hexrays decompiler should be able to present us with clean pseudo-code that matches the actual logic of `y0da.exe`.
Cleaning dead code can be done in several ways. Our way to go would be by iterating over the relevant instructions starting from the entry point of the executable and tainting only those that are part of the actual code flow by writing their addresses to a list. Any other instruction that wasn't tainted in that process can be later patched out with a nop instruction.

The steps to implement this idea resemble a recursive descent disassembly algorithm:

1. Start from a given address that we will denote `entry_point`.
2. Traverse through instructions one after the other and add each instruction's address to the `tainted_addresses` list, until one of the following instructions is reached:
    a. Conditional jump: register the jump's target address to the `conditional_jumps` list and add the address of the jump itself to the tainted_addresses list. Go to step 2 with the next instruction that will be executed if the jump's condition is not met.
    b. Call: register the call's target address into the `function_calls` list and add the address of the call itself to the `tainted_addresses` list. Go to step 2 with the next instruction that will be executed right after returning from the called function.
    c. Return: Add the address of the return instruction to the `tainted_addresses` list, break and move to step 3.
3. If the `conditional_jumps` list is not empty, pop an address from it and go to step 2.
4. If the `function_calls` list is not empty, pop an address from it and go to step 2.

Here are a few points that are worth considering with regards to this algorithm:

- For step 1, we clearly need to pass the executable's main entry point at address 180032701. However, we should also remember that there are two threads executed by the program at addresses 18004928C and 18004E0E7, as we established during the initial analysis phases. Therefore, it's required that we pass each one of them as an `entry_point` to step 1 as well.

- One scenario that we should consider with regards to the above algorithm is infinite loops, in which case the algorithm we'll keep iterating through the same instructions indefinitely. To deal with that, we will use a simple heuristic of counting the number of instances that we are seeing an instruction that has already been tainted. If the counter reaches a high number of our choice, we can infer that we are likely in an infinite loop. This is not a general and robust method of dealing with such cases but will suffice for the purpose of cleaning up the code in question.
- Aside from cleaning up dead code, we need to help IDA determine which function each basic block belongs to. For that purpose, we will rely on the notion of chunked functions and function tails. Basically, chunk functions are ones that are composed of multiple non-contiguous address ranges, just like `y0da.exe`'s code exhibits. To be able to associate a code chunk (or basic block in our case) to a function, we can use append_func_tail in IDAPython.

Following is an IDAPython implementation that deobfuscates `y0da.exe`'s code according to the above method:

```
import ida_ua, idautils, idc, idaapi, ida_bytes, ida_funcs

# Entry points in the code
main_entry_point_ea = 0x180032701
first_thread_ea = 0x18004E0E7
second_thread_ea = 0x18004928C

#  Global data structures
function_calls = []
conditional_jumps = []
conditional_jumps_mnem =
["jo","jno","js","jns","je","jz","jne","jnz","jb","jnae","jc","jnb","jae","jnc"
,"jbe","jna","ja","jnbe","jl","jnge","jge","jnl","jle","jng","jg","jnle","jp","
jpe","jnp","jpo","jcxz","jecxz"]
tainted_addresses = []

# Adds addresses to tainted_addresses list starting from the address ea until a
retn instruction is hit or an infinite loop is detected.
# ea: address to start scanning the code from.
# func_ea: the function that is being currently inspected. All iterated basic
blocks will be appended as function tails to it.
def taint(ea, func_ea):

    initial_ea = ea
    curr_insn = ida_ua.insn_t()
    prev_insn_mnem = ""
    seen_count = 0  # counter to detect infinite loops
    basic_block_start_ea = ea
    basic_block_end_ea = ea

    while True:

        ins_len = idc.create_insn(ea)
        ida_ua.decode_insn(curr_insn, ea)

        if curr_insn.get_canon_mnem() == "retn" or seen_count > 1000:
            ida_funcs.append_func_tail(func_ea, ea, ea + ins_len)
            break

        if ea not in tainted_addresses:
            tainted_addresses.append(ea)
        else:
            seen_count += 1 # We have seen this instruction already but it's
being executed again, possibly in a loop

        if curr_insn.get_canon_mnem() in conditional_jumps_mnem and
curr_insn.Op1.addr not in conditional_jumps and curr_insn.Op1.addr !=
initial_ea and curr_insn.Op1.addr not in tainted_addresses:
            conditional_jumps.append(curr_insn.Op1.addr)

        if curr_insn.get_canon_mnem() == "call" and curr_insn.Op1.addr not in
function_calls and curr_insn.Op1.addr != initial_ea and curr_insn.Op1.addr not
in tainted_addresses:
```

```
            function_calls.append(curr_insn.Op1.addr)

        if curr_insn.get_canon_mnem() == "jmp":
            if prev_insn_mnem == "jmp":
                basic_block_start_ea = ea
            basic_block_end_ea = ea + 5
            ea = curr_insn.Op1.addr
            result = ida_funcs.append_func_tail(func_ea, basic_block_start_ea,
basic_block_end_ea)
            print("BB Start EA: 0x%x, BB End EA: 0x%x, Function EA: 0x%x,
Append Tail Result: %s" % (basic_block_start_ea, basic_block_end_ea, func_ea,
result ))
        else:
            basic_block_start_ea = ea
            if ins_len > 0:
                ea += ins_len
            else:
                ea = idc.next_head(ea)
        prev_insn_mnem = curr_insn.get_canon_mnem()

    tainted_addresses.append(ea)


# Go over all the addresses, including conditional jump targets in a function
that starts at address start_ea.
def taint_func(start_ea):
    taint(start_ea,start_ea)
    while len(conditional_jumps) > 0:
        ea = conditional_jumps.pop()
        taint(ea, start_ea)

# Go over all functions that are located when starting to scan from the address
entry_point_ea.
def taint_from_entry_point(entry_point_ea):
    add_func(entry_point_ea)
    taint_func(entry_point_ea)
    while len(function_calls) > 0:
        ea = function_calls.pop()
        add_func(ea)
        taint_func(ea)

# Find start and end ea of a section with a given name.
def get_section_limits(section_name):
    for s in idautils.Segments():
        if idc.get_segm_name(s) == section_name:
            section_start = idc.get_segm_start(s)
            section_end = idc.get_segm_end(s)
    return (section_start, section_end)

def undefine_section(section_name):
    section_start, section_end = get_section_limits(section_name)
    if section_start > 0 and section_end > 0 and section_start < section_end:
        for ea in range(section_start, section_end):
```

```
            ida_bytes.del_items(ea)

# Remove all dead code in a section.
def patch_nop_to_untainted_addresses(section_name):
    section_start, section_end = get_section_limits(section_name)
    if section_start > 0 and section_end > 0 and section_start < section_end:
        ea = section_start
        while ea < section_end:
            if ea not in tainted_addresses:
                idaapi.patch_byte(ea,0x90)
                ea += 1
            else:
                ins_len = idc.create_insn(ea)
                if ins_len > 0:
                    ea += ins_len
                else:
                    ea = idc.next_head(ea)


if __name__ == "__main__":
    # Undefine all existing code in the .text section
    undefine_section(".text")

    # Taint shellcode instructions within the section starting from given entry
points
    taint_from_entry_point(main_entry_point_ea)
    taint_from_entry_point(first_thread_ea)
    taint_from_entry_point(second_thread_ea)

    # Patch out everything in the section other than the tainted code
    patch_nop_to_untainted_addresses(".text")
```

After the script is done running, we can attempt to decompile the code using the Hexrays decompiler. As an example, the beginning of the main function `sub_18003CEDE` will looks as follows (with some fixes of stack variables):

Figure 20: Yoda ASCII banner seen in the decompilation view after cleaning up the disassembly of y0da.exe

API Address Resolution & Invocation

One adjustment that we can immediately apply on the IDB pertains to the definition of the function `sub_180014D24`, which we noted earlier has an unusual calling convention. As we described, this function gets a name hash as an argument in the r15d register, an unknown flag in the r14 register and the rest of the arguments to the underlying API function are passed via rcx, rdx, r8, r9 (and the stack, if need be). Fortunately, IDA allows us to apply [custom calling conventions]() using the `__usercall` keyword. In our case, our function definition will now look like this:

```
PVOID __usercall getProcAddressAndExecute@<rax>(int returnAddressMode@<r14d>, int
metasploitNameHashArg@<r15d>, int apiArg1@<ecx>, int apiArg2@<edx>, int
apiArg3@<r8d>, int apiArg4@<r9d>)
```

The decompiled code of this function after we apply this definition will then look like this:

```
for ( ldrDte = getPEB()->Ldr->InMemoryOrderModuleList.Flink; ; ldrDte = *nextDte )
{
  baseDllName = CONTAINING_RECORD(ldrDte, LDR_DATA_TABLE_ENTRY, InMemoryOrderLinks)->BaseDllName.Buffer;
  baseDllNameMaxLen = CONTAINING_RECORD(ldrDte, LDR_DATA_TABLE_ENTRY, InMemoryOrderLinks)->BaseDllName.MaximumLength;
  moduleNameHash = 0;
  do
  {
    baseDllNameChr = *baseDllName;
    baseDllName = (baseDllName + 1);
    if ( baseDllNameChr >= 'a' )
      LOBYTE(baseDllNameChr) = baseDllNameChr - 0x20;
    moduleNameHash = baseDllNameChr + __ROR4__(__ROR4__(moduleNameHash, 10), 3);
    --baseDllNameMaxLen;
  }
  while ( baseDllNameMaxLen );
  nextDte = ldrDte;
  c_moduleNameHash = moduleNameHash;
  dllBase = CONTAINING_RECORD(ldrDte, LDR_DATA_TABLE_ENTRY, InMemoryOrderLinks)->DllBase;
  dllNtHeaders = &dllBase[*(dllBase + 15)];
  if ( dllNtHeaders->OptionalHeader.Magic == 0x20B )
  {
    exportDirRva = dllNtHeaders->OptionalHeader.DataDirectory[0].VirtualAddress;
    if ( exportDirRva )
      break;
  }
ABEL_5:
  ;
}
exportDir = exportDirRva + dllBase;
numberOfNames = *(exportDirRva + dllBase + 24);
addressOfNames = &dllBase[*(exportDirRva + dllBase + 32)];
do
{
  if ( !numberOfNames )
    goto LABEL_5;
  --numberOfNames;
  exportName = &dllBase[*&addressOfNames[4 * numberOfNames]];
  exportNameHash = 0;
  do
  {
    exportNameChr = *exportName++;
    exportNameHash = exportNameChr + __ROR4__(__ROR4__(exportNameHash, 8), 5);
  }
  while ( exportNameChr );
}
while ( c_moduleNameHash + exportNameHash != metasploitNameHashArg );
LOWORD(numberOfNames) = *&dllBase[2 * numberOfNames + *(exportDir + 9)];
functionAddr = &dllBase[*&dllBase[4 * numberOfNames + *(exportDir + 7)]];
if ( returnAddressMode != 1 )
  __asm { jmp     rax }
return functionAddr;
```

Figure 21: Decompiled code of the function used for resolving and calling API functions

We can infer that the flag passed in r14 indicates if `sub_180014D24` will return the resolved API address or call it directly by passing the control to the API function with the instruction `jmp rax`.
In addition, we can now cross reference all the calls to this function, collect their hashes and mark-up functions according to them where applicable. These are the relevant Metasploit name hashes and their associated API functions:

Metasploit Name Hash	Associated API Function
0xE553A458	kernel32!VirtualAlloc
0x300F2F0B	kernel32!VirtualFree
0x528796C6	kernel32!CloseHandle
0x56A2B5F0	kernel32!ExitProcess
0x863FCC79	kernel32!CreateProcessA
0x160D6838	kernel32!CreateThread
0xBB5F9EAD	kernel32!ReadFile
0x5BAE572D	kernel32!WriteFile
0xDDCEADE7	kernel32!GetEnvironmentVariableA
0x601D8708	kernel32!WaitForSingleObject
0x726774C	kernel32!LoadLibraryA
0xEAFCF3E	kernel32!CreatePipe
0x6558F55E	kernel32!FindResourceA
0x8E8BB14A	kernel32!LoadResource
0xE8BE94B	kernel32!LockResource
0x42F9102E	kernel32!SizeOfResource
0xB33CB718	kernel32!PeekNamedPipe

0x614D6E75	ws2_32!closesocket
0x6B8029	ws2_32!WSAStartup
0xF44A6E2B	ws2_32!WSACleanup
0xED83E9BA	ws2_32!socket
0xFF38E9B7	ws2_32.dll!listen
0xE13BEC74	ws2_32.dll!accept
0x6737DBC2	ws2_32!bind
0x5FC8D902	ws2_32!recv
0x5F38EBC2	ws2_32!send
0xD0EB608D	user32!wsprintf

Hidden Commands

Let's revisit the FLOSS output containing stack strings and observe an interesting one that looks like a possible shell command. It is found in the function `sub_18004e0e7` which matches the input processing thread that we located earlier:

```
0x18004e0e7   0x1800365f0          0x38              gimmie_advic3
```

Figure 22: `gimmie_advic3` hidden command found in the code of the thread sub_18004e0e7

Looking at the decompilation of this function we spot two similar strings being initialized:

```
void __fastcall sub_18004E0E7(__int64 *a1)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  v13 = a1;
  strcpy(s_gimmie_advice, "gimmie_advic3");
  strcpy(&s_gimmie_secret, "gimmie_s3cr3t");
```

Figure 23: Hidden shell commands found in the decompiled view of sub_18004e0e7

If we enter gimmie_advic3 as an input to the shell, we'll get a different Yoda advice each time:

```
C:\Windows>gimmie_advic3
Y0da's life tip #0x10d4:
"No! Try not. Do. Or do not. There is no try."

C:\Windows>gimmie_advic3
Y0da's life tip #0x10d4:
"Size of code matters not. Look at me. Judge me by my size, do you?"

C:\Windows>gimmie_advic3
Y0da's life tip #0x10d4:
"A Jedi uses the Force for knowledge and defense, never for attack."

C:\Windows>gimmie_advic3
Y0da's life tip #0x10d4:
"A Jedi's strength flows from their knowledge of assembly."

C:\Windows>gimmie_advic3
Y0da's life tip #0x10d4:
"Fear of malware is the path to the dark side."

C:\Windows>gimmie_advic3
Y0da's life tip #0x10d4:
"Obfuscation leads to anger. Anger leads to hate. Hate leads to suffering."

C:\Windows>gimmie_advic3
Y0da's life tip #0x10d4:
"Packers, crypters, shellcode. The dark side are they."
```

Figure 24: Yoda advice presented in the shell in response to a gimmie_advic3 command

This doesn't bring us closer to the flag, so let's try the other command which presents us with the following password prompt:

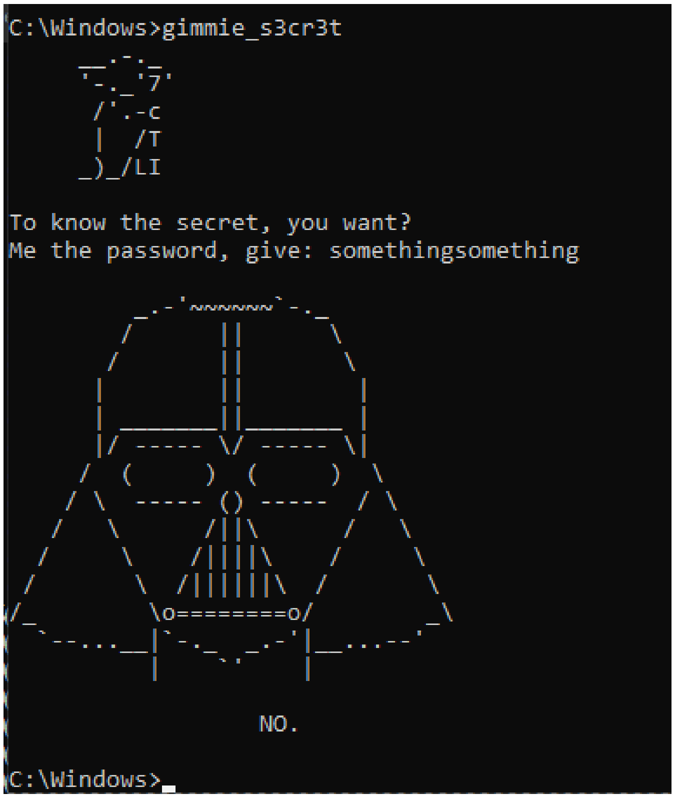

Figure 25: Password prompt presented in the shell as a response to the gimmie_s3cr3t command

Password Prompt

It is clear at this point that we would like to find out the password that we should enter in the above prompt. If we revisit the FLOSS output again, we'll find some of the above prompt's strings and the function `sub_1800216F6` in which they were found. If we look at the decompiled view of this function while using the information that we already have about the function `sub_180014D24` (which we'll name getProcAddressAndExecute at this point) as well as do some basic analysis on the functions it invokes, we'll get the following code:

```
int __stdcall sub_1800216F6(void *a1)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  v8 = 0;
  strcpy(passwordReqMsg, "'.-c\n      |  /T\n     _)_/LI\n \nTo know the secret, you want? \nMe the password, give:");
  strcpy(
    darthBanner,
    "\n"
    "         _.-'~~~~~~`-._\n"
    "        /      ||      \\\n"
    "       /       ||       \\\n"
    "      |        ||        |\n"
    "      | _______||_______ |\n"
    "      |/ ----- \\/ ----- \\|\n"
    "     /  (     )  (     )  \\\n"
    "    / \\  ----- () -----  / \\\n"
    "   /   \\      /||\\      /   \\\n"
    "  /     \\    /||||\\    /     \\\n"
    " /       \\  /||||||\\  /       \\\n"
    "/_        \\o========o/        _\\\n"
    "  `--...__|`-._  _.-'|__...--'\n"
    "          |    `'    |\n"
    "\t\t  \n"
    "\t\t  NO.\n");
  rawInput = getProcAddressAndExecute(0, 0xE553A458, 0, 0x4000, 4096, 4);// VirtualAlloc
  for ( i = 0; i < 0x75; i += v2 )
  {
    v2 = pSend(ps->inSock, &passwordReqMsg[117 * i], 117 - i, 0);
    if ( !v2 )
      cleanupAndExit(ps);
  }
  if ( !pRecv(ps->inSock, ps->netBufferIn, 0x4000, 0) )
    cleanupAndExit(ps);
  strcat(rawInput, ps->netBufferIn, 0x4000ui64);
  removeChar(rawInput, ' ');
  removeChar(rawInput, '\r');
  removeChar(rawInput, '\n');
  result = sub_18001BB76(rawInput, ps);
  if ( !result )
  {
    for ( j = 0; ; j += v3 )
    {
      result = j;
      if ( j >= 0x198 )
        break;
      v3 = pSend(ps->inSock, &darthBanner[408 * j + 32], 408 - j, 0);
      if ( !v3 )
        cleanupAndExit(ps);
    }
  }
  if ( rawInput )
    return (getProcAddressAndExecute)(0, 0x300F2F0B, rawInput, 0, 0x8000);// VirtualFree
  return result;
}
```

Figure 26: Decompiled code of the function sub_1800216F6

We'll note that the result of `sub_18001BB76` is being checked, and if it's null the `darthBanner` buffer is sent over the socket, indicating that we should further seek for the password checking logic in this function.
If we consider our previous CAPA results, we can recall the function `sub_1800126AB` which is part of the MD5 hash calculation function `sub_18002483`.This is being used by the function `sub_18004EBC7` invoked from `sub_18001BB76`. It's only if `sub_18004EBC7` returns a non-null value that `sub_18001BB76` performs any additional actions, which warrants a deeper inspection of the former.

```
__int64 __fastcall sub_1800126AB(md5Ctx *ctx)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD

  ctx->size = 0i64;
  ctx->buffer[0] = 0x67452301;
  ctx->buffer[1] = 0xEFCDAB89;
  ctx->buffer[2] = 0x98BADCFE;
  result = 12i64;
  ctx->buffer[3] = 0x10325476;
  return result;
}
```

Figure 27: sub_1800126AB the initialization of an MD5 context struct.

Other than MD5 calculation, there is a call to sub_1800382E1 from sub_18004EBC7. This function is an implementation of a strtok_r method used to tokenize a string according to a given character.

```
char *__fastcall sub_1800382E1(char *ptr, char *sep, char **end)
{
  char *start; // [rsp+20h] [rbp+8h]

  start = ptr;
  if ( !ptr )
    start = *end;
  while ( *start && strchr(sep, *start) )
    ++start;
  if ( !*start )
    return 0i64;
  for ( *end = start + 1; **end && !strchr(sep, **end); ++*end )
    ;
  if ( **end )
    *(*end)++ = 0;
  return start;
}
```

Figure 28: Implementation of strtok_r

If we go on and look further into what's happening in `sub_18004EBC7` we'll see that it tokenizes an input string according to a given character and then computes the MD5 hash of each token, comparing it to hard-coded MD5 hash values on the stack. At this point, we can assess that this is the logic that underlies the password check in the gimmie_s3cr3t prompt.

```
md5ToString(passInputBuff, inputHash);
if ( !memcmp(inputHash, passHash, 0x10ui64) )
  v24 = 1;
inputToken = strtok_r(passInputBuff, &chrUnderscore, &passInput);
do
{
  md5ToString(inputToken, inputHash);
  if ( !tokenCount )
  {
    if ( memcmp(inputHash, tokenOneHash, 0x10ui64) )
      break;
  }
  if ( tokenCount == 1 )
  {
    if ( memcmp(inputHash, tokenTwoHash, 0x10ui64) )
      break;
  }
  if ( tokenCount == 2 )
  {
    if ( memcmp(inputHash, tokenThreeHash, 0x10ui64) )
      break;
  }
  if ( tokenCount == 3 )
  {
    if ( memcmp(inputHash, tokenFourHash, 0x10ui64) )
      break;
  }
  inputToken = strtok_r(0i64, &chrUnderscore, &passInput);
  ++tokenCount;
  if ( !inputToken )
    break;
}
while ( tokenCount < 4 );
00045417 sub_18004EBC7:88 (180046017) (Synchronized with IDA View-A)
```

Figure 29: Decompiled code outlining password checking logic.

Unfortunately, we may not be able to fully rely on the decompilation results due to some stack analysis issues, but as the figure above indicates, we can find the address of each call in the disassembly view by setting the cursor on the line of interest in the Hexrays pseudocode and looking at the current location marked in the bar at the bottom of the window.

With that information, we can simply attach a debugger to `y0da.exe`'s process and break on the `strtok_r` call at 180046017 to see what character is used to tokenize the password. We'll find out that this is an underscore (or 0x5F in ASCII).

```
0:005> bp 180046017
0:005> g
Breakpoint 0 hit
y0da+0x46017:
00000001`80046017 e8c522ffff      call    y0da+0x382e1 (00000001`800382e1)
0:004> da /c 1 rdx
00000000`01d9fb50  "_"
```

Figure 30: The character used to tokenize the input to the password prompt

Then, we can break on each call to `memcmp` and check the MD5 hash that the token hashes will be compared to, as pointed to by the argument in the rdx register.

```
0:004> bp 180009079
0:004> g
Breakpoint 1 hit
y0da+0x9079:
00000001`80009079 e8a5540000      call    y0da+0xe523 (00000001`8000e523)
0:004> db /c 0x10 rdx
00000000`01d9fb68  4c 84 76 db 19 7a 10 39-15 3c a7 24 67 4f 7e 13  L.v..z.9.<.$gO~.
```

Figure 31: Example of an MD5 hash that the passwords token hashes are compared to

After we collect all the compared hashes, we can use a simple [online reverse MD5 tool] to give us the keywords that would correspond to it. As outlined in the table below, we can infer that the password should be `patience_y0u_must_h4v3`.

MD5	Reverse MD5
4C8476DB197A1039153CA724674F7E13	patience
627FE11EEEF8994B7254FC1DA4A0A3C7	y0u
D0E6EF34E76C41B0FAC84F608289D013	must
48367C670F6189CF3F413BE394F4F335	h4v3

We can now rename `sub_18004EBC7` to `checkPassword` and go back to `sub_18001BB76`. After some analysis of the other functions in `sub_18001B76` we can learn that if the checked password is correct, it is being used as the RC4 key that decrypts the `Y0D4` resource we found in the basic static analysis phase. The beginning of the decrypted result is checked against the magic 0xFFD8FFE0 of a JPEG image.

```
__int64 __fastcall sub_18001BB76(BYTE *passwordInput, stPipeShell *ps)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  rsrcSize[0] = 0;
  jpegMagic[0] = 0xFF;
  jpegMagic[1] = 0xD8;
  jpegMagic[2] = 0xFF;
  jpegMagic[3] = 0xE0;
  if ( !checkPassword(passwordInput) )
    return 0i64;
  if ( !ps->rsrcBuffer )
  {
    ps->rsrcBuffer = readResource(rsrcSize);
    if ( !ps->rsrcBuffer )
      return 0i64;
  }
  passwordInputLen = strlen(passwordInput);
  rc4(passwordInput, passwordInputLen, ps->rsrcBuffer, rsrcSize[0]);
  if ( memcmp(ps->rsrcBuffer, jpegMagic, 4ui64) )
    return 0i64;
  LODWORD(ps->rsrcBufferLen) = rsrcSize[0];
  return 1i64;
}
```

Figure 32: Code that decrypts the Y0D4 resource using RC4 and the password prompt's input as the key

Decrypting the resource and opening it as an image will yield the following false flag:

Figure 33: The false flag image that gets decrypted from the Y0D4 resource

Since this flag does not end with the flare-on.com domain and it explicitly states that it isn't the flag, we need to keep looking for the real one. Let's see what happens when we enter the password that we just found into the gimmie_s3cr3t password prompt:

```
C:\Windows>gimmie_s3cr3t
     __.-._
     '-._'7'
      /'.-c
      |  /T
     _)_/LI

To know the secret, you want?
Me the password, give: patience_y0u_must_h4v3

              .--.
 ::\`--._,'.::.`._.--'/::
 ::::.  `  __::__  '  .::::
 ::::::-:.`'..`'.:-::::::
 ::::::::\ `--' /::::::::

M4st3r Y0d4 says OIZC4eMC/UnTPfDDMMaHeQXUHMPZy4LfSgg/HnB5SXVOIyKOBIHMe45B2KBCe5T/HRfRHZ4SKJe3eLJHeMe5IM5QQJ======

C:\Windows>


              .--.
 ::\`--._,'.::.`._.--'/::
 ::::.  `  __::__  '  .::::
 ::::::-:.`'..`'.:-::::::
 ::::::::\ `--' /::::::::

M4st3r Y0d4 says /SeOTX+LJS4/32+ynPBMPOe/I5MXw4HXyJRD4+QwP5KVafngPMgw+y+y/ne+/OBRMXaCnZI3JIa/D/ZeBJTVUnyCCC======
 C:\Windows>
```

Figure 34: The result of entering the correct password into the prompt

We see a recurring message that is attached to the shell outputs sent to us. In each instance M4st3r Y0d4 says something that appears encoded to us, wherein the encoded string keeps changing all the time. Our next mission will be to decode Y0d4's words.

Flag Construction

To get to the part of the code in which the flag is constructed we can revisit our FLOSS results and find the beginning of the string that presents the encoded flag. The function in which it appears is `sub_18004928c` which corresponds to another thread that we still didn't cover. This thread is responsible for sending output from the `cmd.exe` console over the TCP socket and appending the Y0da says message to it after the correct password has been entered into the gimmie_s3cr3t prompt.

```
0x18004928c   0x180014d24        0x3d          M4st3r Y0d4 says
```

Figure 35: The string of interest that pertains to the correct password output, as presented by FLOSS

As you may have noted, each Y0d4 says print contains a different encoded message. As an example, we'll take the first one that appears after entering the correct password:


```
OIZC4eMC/UnTPfDDMMaHeQXUHMPZy4LfSgg/HnB5SXVOIyKOBIHMe45B2KBCe5T/HRfRHZ4SKJe3eLJHeMe5
IM5QQJ======
```

One thing we can already get rid of is the final layer of encoding. The above string is in fact Base32 encoded with the custom index `Q4T23aSwLnUgHPOIfyKBJVM5+DXZC/Re=` that we have already seen in FLOSS results (note the length of the index, it cannot be used by Base64 – if that was your first guess). When we decode the first Y0d4 says related string from Base32, we get the following sequence of 56 bytes:

```
7F F7 C0 FE DC EA 92 26 C3 39 B5 8A CF 83 4A 65 9B B8 85 10 32 D7 D6 26 77 36 AA
E7 C6 4E 9B D9 6F 86 F3 1C A7 CF DC 5D 67 A1 E6 6C 26 95 3E 4F A2 8C FD BF 77 DA
E0 05
```

To better understand those, we need to take one step back and see how they are generated. Within `sub_18004928C`, the function of interest for the flag construction is `sub_180050E82`. In that function we'll find a couple of binary patterns that are sought within the decrypted resource image – 0xFFE1AA3B and 0xFFE2A1C5. Those are in fact two hidden markers that are used to store data in the underlying JPEG image. The search for those markers is being done via calls to a `memmem` function in the addresses 18005C570 and 180002737.

```
0:004> bp 18005C570
0:004> bp 180002737
0:004> g
Breakpoint 0 hit
y0da+0x5c570:
00000001`8005c570 e85d93fdff      call    y0da+0x358d2 (00000001`800358d2)
0:003> r r8
r8=000000001d5e038
0:003> db r8
00000000`01d5e038  ff e1 aa 3b ff e2 a1 c5-00 00 00 00 00 00 00 00  ...;............
00000000`01d5e048  00 00 df 01 00 00 00 00-00 00 e0 01 00 00 00 00  ................
00000000`01d5e058  00 00 e1 01 00 00 00 00-00 00 dc 01 00 00 00 00  ................
00000000`01d5e068  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
00000000`01d5e078  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
00000000`01d5e088  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
00000000`01d5e098  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
00000000`01d5e0a8  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
0:003> g
Breakpoint 1 hit
y0da+0x2737:
00000001`80002737 e896310300      call    y0da+0x358d2 (00000001`800358d2)
0:003> r r8
r8=000000001d5e03c
0:003> db r8
00000000`01d5e03c  ff e2 a1 c5 00 00 00 00-00 00 00 00 00 00 df 01  ................
00000000`01d5e04c  00 00 00 00 00 00 e0 01-00 00 00 00 00 00 e1 01  ................
00000000`01d5e05c  00 00 00 00 00 00 dc 01-00 00 00 00 00 00 00 00  ................
00000000`01d5e06c  00 00 00 00 f1 b0 de 01-00 00 00 00 00 00 00 00  ................
00000000`01d5e07c  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
00000000`01d5e08c  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
00000000`01d5e09c  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
00000000`01d5e0ac  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
```

Figure 36: Binary patterns in the decrypted resource image.

We can look for these markers in the decrypted image and note that they are followed by buffers with binary blobs of lengths 0x3B and 0x1C5.

```
1:B0F0h  D9 FF E1 AA 3B 7F 2B D8 F5 C3 44 6D B7 75 95 89
1:B100h  A7 B9 C3 2C 3F 9E 91 B8 DC 6E 55 A7 51 E6 2C 59
1:B110h  BC 9C 12 98 06 8B A0 50 79 18 AA 29 4E 84 96 5F
1:B120h  A6 37 9F ED 9A 33 3C ED 34 2D 63 7F 6C 5A FF E2
1:B130h  A1 C5 05 AC 00 00 00 C3 05 E4 00 00 00 C3 05 E8
1:B140h  00 00 00 C3 83 C0 5A C3 83 C0 60 C3 83 C0 70 C3
1:B150h  83 C0 7B C3 05 8F 00 00 00 C3 05 96 00 00 00 C3
1:B160h  03 45 24 C3 48 83 C5 38 C3 39 45 24 C3 FF C0 C3
1:B170h  88 04 0A C3 89 55 10 C3 4C 89 45 18 C3 88 45 20
1:B180h  C3 89 45 24 C3 48 89 4D 08 C3 C7 45 24 00 00 00
1:B190h  00 C3 8B 45 24 C3 8B 45 48 C3 8B 4D 24 C3 48 8B
1:B1A0h  4D 40 C3 48 8B 55 40 C3 48 8B 55 50 C3 0F B6 04
1:B1B0h  01 C3 0F B6 45 20 C3 0F B6 0C 0A C3 0F B6 4D 20
1:B1C0h  C3 F7 D8 C3 F7 D0 C3 0B C1 C3 D1 F8 C3 C1 F8 02
1:B1D0h  C3 C1 F8 03 C3 C1 F8 05 C3 C1 F8 06 C3 C1 F8 07
1:B1E0h  C3 D1 E1 C3 C1 E1 02 C3 C1 E1 03 C3 C1 E1 05 C3
1:B1F0h  C1 E1 06 C3 C1 E1 07 C3 2D B1 00 00 00 C3 2D B2
1:B200h  00 00 00 C3 2D C3 00 00 00 C3 2D C5 00 00 00 C3
1:B210h  2D DC 00 00 00 C3 2D F3 00 00 00 C3 2D FF 00 00
1:B220h  00 C3 83 E8 18 C3 83 E8 1A C3 83 E8 1E C3 83 E8
1:B230h  28 C3 83 E8 36 C3 83 E8 04 C3 83 E8 49 C3 83 E8
1:B240h  56 C3 83 E8 58 C3 2D 81 00 00 00 C3 2D 90 00 00
1:B250h  00 C3 2D 9A 00 00 00 C3 2B 45 24 C3 48 83 ED 38
1:B260h  C3 35 A3 00 00 00 C3 35 B6 00 00 00 C3 35 BF 00
1:B270h  00 00 C3 35 C2 00 00 00 C3 35 C9 00 00 00 C3 35
1:B280h  CB 00 00 00 C3 83 F0 0D C3 35 E1 00 00 00 C3 35
1:B290h  EB 00 00 00 C3 83 F0 16 C3 83 F0 20 C3 83 F0 22
1:B2A0h  C3 83 F0 25 C3 83 F0 40 C3 83 F0 78 C3 83 F0 7C
1:B2B0h  C3 35 8F 00 00 00 C3 33 45 24 C3 33 C0 C3 33 C1
1:B2C0h  C3 FF C1 C3 8B C9 C3 81 E1 FF 00 00 00 C3 8B 55
1:B2D0h  24 C3 83 C2 02 C3 8B D2 C3 4C 8B 45 50 C3 41 0F
1:B2E0h  B6 14 10 C3 D1 FA C3 81 E2 FF 00 00 00 C3 23 CA
1:B2F0h  C3 83 C1 03 C3
```

Figure 37: The sought patterns and the blobs that follow them in the image

After that we'll see a loop that populates a buffer of 60 bytes generated by the function `sub_180063054`. If we revisit our CAPA results from the basic static analysis phase, we'll see that it's one of few functions that pertain to a Mersenne Twister PRNG implementation. This function generates a pseudo-random integer in each iteration of the loop, which is then assigned as 4 bytes into the buffer in question, until all 60 bytes are filled.

```
while ( i < 60 )
{
  merseneTwisterLong = getRandLongMT(mtRand);
  mersenneTwisterSequence[i] = HIBYTE(merseneTwisterLong);
  mersenneTwisterSequence[i + 1] = BYTE2(merseneTwisterLong);
  mersenneTwisterSequence[i + 2] = BYTE1(merseneTwisterLong);
  mersenneTwisterSequence[i + 3] = merseneTwisterLong;
  LODWORD(i) = i + 4;
}
```

Figure 38: Calculation of a Mersenne Twister generated sequence

After the loop, we'll see a call to `sub_180015EC1`. If we inspect the arguments passed to this function, we'll find that it receives the buffer that follows the 0xFFE1AA3B marker in the decrypted resource, the size of this buffer which is 0x3B, the 60-byte Mersenne Twister sequence that was generated in the loop and the second buffer from the decrypted resource that follows the 0xFFE2A1C5 marker.

```
0:003> bp 18004936E
0:003> g
Breakpoint 2 hit
y0da+0x4936e:
00000001`8004936e 4c894c2420      mov     qword ptr [rsp+20h],r9 ss:00000000`01d5e028=0000000000000000
0:003> r rcx
rcx=0000000001e00000
                                  First Buffer from JPEG
0:003> db rcx
00000000`01e00000  7f 2b d8 f5 c3 44 6d b7-75 95 89 a7 b9 c3 2c 3f  .+...Dm.u.....,?
00000000`01e00010  9e 91 b8 dc 6e 55 a7 51-e6 2c 59 bc 9c 12 98 06  ....nU.Q.,Y.....
00000000`01e00020  8b a0 50 79 18 aa 29 4e-84 96 5f a6 37 9f ed 9a  ..Py..)N.._.7...
00000000`01e00030  33 3c ed 34 2d 63 7f 6c-5a 00 00 00 00 00 00 00  3<.4-c.lZ.......
00000000`01e00040  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
00000000`01e00050  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
00000000`01e00060  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
00000000`01e00070  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
0:003> r rdx
rdx=0000000000000039  Length of the Blob in the First Buffer from JPEG
0:003> r r8
r8=0000000001df0000
                                  Mersenne Twister Sequence
0:003> db r8
00000000`01df0000  9d b5 df 75 92 c8 67 0b-50 60 0f b3 4e eb d6 67  ...u..g.P`..N..g
00000000`01df0010  08 eb 59 e9 cf 7f f5 39-a4 07 cb a2 d3 16 c6 93  ..Y....9........
00000000`01df0020  18 4b 01 04 64 a5 4d a8-42 7d 24 d0 a8 2b fb af  .K..d.M.B}$..+..
00000000`01df0030  a1 7d 24 5d 35 eb 3b de-4d 64 69 a4 00 00 00 00  .}$]5.;.Mdi.....
00000000`01df0040  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
00000000`01df0050  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
00000000`01df0060  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
00000000`01df0070  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
0:003> r r9
r9=0000000001e10000
                                  Second Buffer from JPEG
0:003> db r9
00000000`01e10000  05 ac 00 00 00 c3 05 e4-00 00 00 c3 05 e8 00 00  ................
00000000`01e10010  00 c3 83 c0 5a c3 83 c0-60 c3 83 c0 70 c3 83 c0  ....Z...`...p...
00000000`01e10020  7b c3 05 8f 00 00 00 c3-05 96 00 00 00 c3 03 45  {..............E
00000000`01e10030  24 c3 48 83 c5 38 c3 39-45 24 c3 ff c0 c3 88 04  $.H..8.9E$......
00000000`01e10040  0a c3 89 55 10 c3 4c 89-45 18 c3 88 45 20 c3 89  ...U..L.E...E ..
00000000`01e10050  45 24 c3 48 89 4d 08 c3-c7 45 24 00 00 00 00 c3  E$.H.M...E$.....
00000000`01e10060  8b 45 24 c3 8b 45 48 c3-8b 4d 24 c3 48 8b 4d 40  .E$..EH..M$.H.M@
00000000`01e10070  c3 48 8b 55 40 c3 48 8b-55 50 c3 0f b6 04 01 c3  .H.U@.H.UP......
```

Figure 39: Arguments passed to `sub_180015EC1`

Looking deeper into the function `sub_180015EC1`, we'll see a loop that will iterate over the bytes of the of the first buffer from the JPEG and will invoke `sub_18001D361` for each byte:

```
__int64 __fastcall buildEncodedFlag(
        unsigned __int8 *bufferFromJpeg1,
        unsigned int bufferFromJpeg1Len,
        unsigned __int8 *mersenneTwisterSequence,
        unsigned __int8 *bufferFromJpeg2)
{
  __int64 result; // rax
  unsigned int m; // [rsp+4h] [rbp-14h]

  for ( m = 0; ; ++m )
  {
    result = bufferFromJpeg1Len;
    if ( m >= bufferFromJpeg1Len )
      break;
    sub_18001D361(bufferFromJpeg2);
  }
  return result;
}
```

Figure 40: Loop that processes encoded bytes, likely used to build the flag

Unfortunately, we don't get a proper decompilation of `sub_18001D361`:

```
void __stdcall sub_18001D361(char *bufferFromJpeg2)
{
  ;
}
```

Figure 41: Function called from `sub_180015EC1`, which fails to decompile.

To understand why that happens, let's trace the instructions of this function with WinDbg by setting a breakpoint on it and running the command `pa 18001D361`. You can split the output in two. The first are the actual instructions of the function that consist of a set of addresses constructed and pushed on the stack. Those addresses are all within the range of the second buffer of the decrypted resource:

```
y0da+0x1d364:
00000001`8001d364 e92f7f0300      jmp     y0da+0x55298 ;(00000001`80055298)
y0da+0x55298:
00000001`80055298 4883c508        add     rbp,8
y0da+0x5529c:
00000001`8005529c e9b032feff      jmp     y0da+0x38551 ;(00000001`80038551)
y0da+0x38551:
00000001`80038551 498bf1          mov     rsi,r9
y0da+0x38554:
00000001`80038554 e9b324feff      jmp     y0da+0x1aa0c ;(00000001`8001aa0c)
y0da+0x1aa0c:
00000001`8001aa0c 4883c63e        add     rsi,3Eh
y0da+0x1aa10:
00000001`8001aa10 e927e90000      jmp     y0da+0x2933c ;(00000001`8002933c)
y0da+0x2933c:
00000001`8002933c 56              push    rsi
y0da+0x2933d:
00000001`8002933d e9c4290100      jmp     y0da+0x3bd06 ;(00000001`8003bd06)
y0da+0x3bd06:
00000001`8003bd06 498bf1          mov     rsi,r9
y0da+0x3bd09:
00000001`8003bd09 e908eafdff      jmp     y0da+0x1a716 ;(00000001`8001a716)
y0da+0x1a716:
00000001`8001a716 4883c671        add     rsi,71h
y0da+0x1a71a:
00000001`8001a71a e997e20100      jmp     y0da+0x389b6 ;(00000001`800389b6)
y0da+0x389b6:
00000001`800389b6 56              push    rsi
...
...
y0da+0x3a4ac:
00000001`8003a4ac e945290200      jmp     y0da+0x5cdf6 ;(00000001`8005cdf6)
y0da+0x5cdf6:
00000001`8005cdf6 498bf1          mov     rsi,r9
y0da+0x5cdf9:
00000001`8005cdf9 e9a2c7feff      jmp     y0da+0x495a0 ;(00000001`800495a0)
y0da+0x495a0:
00000001`800495a0 4883c660        add     rsi,60h
y0da+0x495a4:
00000001`800495a4 e98cf4feff      jmp     y0da+0x38a35 ;(00000001`80038a35)
y0da+0x38a35:
00000001`80038a35 56              push    rsi
y0da+0x38a36:
00000001`80038a36 e9e01dfeff      jmp     y0da+0x1a81b ;(00000001`8001a81b)
y0da+0x1a81b:
00000001`8001a81b c3              ret
```

After the `ret` instruction is invoked, we see the execution of other instructions, wherein each instruction is followed by a `ret`:

```
00000000`01e10060 8b4524          mov     eax,dword ptr [rbp+24h] ;ss:00000000`01d5dff4=00000000
00000000`01e10063 c3              ret
00000000`01e1006c 488b4d40        mov     rcx,qword ptr [rbp+40h]
;ss:00000000`01d5e010=000000001e00000
00000000`01e10070 c3              ret
00000000`01e1007b 0fb60401        movzx   eax,byte ptr [rcx+rax] ;ds:00000000`01e00000=7f
00000000`01e1007f c3              ret
00000000`01e1004b 884520          mov     byte ptr [rbp+20h],al ;ss:00000000`01d5dff0=00
00000000`01e1004e c3              ret
00000000`01e10080 0fb64520        movzx   eax,byte ptr [rbp+20h] ;ss:00000000`01d5dff0=7f
00000000`01e10084 c3              ret
00000000`01e1009f c1f803          sar     eax,3
00000000`01e100a2 c3              ret
00000000`01e1008a 0fb64d20        movzx   ecx,byte ptr [rbp+20h] ;ss:00000000`01d5dff0=7f
00000000`01e1008e c3              ret
00000000`01e100ba c1e105          shl     ecx,5
00000000`01e100bd c3              ret
00000000`01e10095 0bc1            or      eax,ecx
00000000`01e10097 c3              ret
00000000`01e1004b 884520          mov     byte ptr [rbp+20h],al ;ss:00000000`01d5dff0=7f
00000000`01e1004e c3              ret
00000000`01e10080 0fb64520        movzx   eax,byte ptr [rbp+20h] ;ss:00000000`01d5dff0=ef
00000000`01e10084 c3              ret
00000000`01e10000 05ac000000      add     eax,0ACh
00000000`01e10005 c3              ret
00000000`01e1004b 884520          mov     byte ptr [rbp+20h],al ;ss:00000000`01d5dff0=ef
00000000`01e1004e c3              ret

...
```

What we can infer from this is that `sub_18001D361` is responsible for constructing a ROP chain such that the gadgets are taken from the second buffer of the decrypted resource. Each execution of the ROP chain constructs one encrypted flag byte (the final encrypted flag is what we decoded from Base32 earlier). To decipher the flag, we don't need the whole ROP chain, but only the last ~32 instructions (excluding the `ret` instructions). If we take those instructions that construct the first encrypted flag byte from the trace and clean up the ret instructions, we'll get the following code:

```
...
; [rbp + 20h] contains the first character of the flag - 0x50 ('P')
movzx    eax,byte ptr [rbp+20h] ;ss:00000000`01d5dff0=50
; [rbp + 24h] contains the index of the flag character we are processing, in this
case 0
mov      ecx,dword ptr [rbp+24h] ;ss:00000000`01d5dff4=00000000
; [rbp+50h] points to the  Mersenne Twister sequence buffer
mov      rdx,qword ptr [rbp+50h] ;ss:00000000`01d5e020=000000001df0000
; 0x9D is the first character in the Mersenne Twister sequence buffer
movzx    ecx,byte ptr [rdx+rcx] ;ds:00000000`01df0000=9d
; The byte at index 0 of the flag (i.e., 'P') is XORed with the first character of
the Mersenne Twister sequence (i.e., 0x9D)
xor      eax,ecx
mov      ecx,dword ptr [rbp+24h] ;ss:00000000`01d5dff4=00000000
; Next index value (1) is put into ecx
inc      ecx
mov      ecx,ecx
mov      rdx,qword ptr [rbp+50h] ;ss:00000000`01d5e020=000000001df0000
; Next character in the Mersenne Twister sequence is 0xB5
movzx    ecx,byte ptr [rdx+rcx] ;ds:00000000`01df0001=b5
; The character gets shifted left by one bit
shl      ecx,1
and      ecx,0FFh
mov      edx,dword ptr [rbp+24h] ;ss:00000000`01d5dff4=00000000
; Next index value (2) is put into edx
add      edx,2
mov      edx,edx
mov      r8,qword ptr [rbp+50h] ;ss:00000000`01d5e020=000000001df0000
; Next character in the Mersenne Twister sequence is 0xDF
movzx    edx,byte ptr [r8+rdx] ;ds:00000000`01df0002=df
; The character gets shifted right by 1
sar      edx,1
and      edx,0FFh
and      ecx,edx
; The result is XORed with what we calculated thus far
xor      eax,ecx
mov      ecx,dword ptr [rbp+24h] ;ss:00000000`01d5dff4=00000000
; Next index value (3) is put into ecx
add      ecx,3
mov      ecx,ecx
mov      rdx,qword ptr [rbp+50h] ;ss:00000000`01d5e020=000000001df0000
; Next character in the Mersenne Twister sequence is 0x75
movzx    ecx,byte ptr [rdx+rcx] ;ds:00000000`01df0003=75
; The character gets shifted left by 2
shl      ecx,2
and      ecx,0FFh
; The result is XORed with what we calculated thus far
xor      eax,ecx
mov      ecx,dword ptr [rbp+24h] ;ss:00000000`01d5dff4=00000000
; [rbp + 40h] contains the target address to which we write the encoded flag byte
mov      rdx,qword ptr [rbp+40h] ;ss:00000000`01d5e010=000000001e00000
; The encoded flag byte 0x7F is written to memory
mov      byte ptr [rdx+rcx],al ;ds:00000000`01e00000=7f
retn
```

As we can see, the encryption is a mere XOR between the flag bytes and slightly mutated Mersenne Twister bytes that were calculated formerly. Based on this logic we can write a simple Python script that will decode the flag:

```
encodedFlag = [0x7F, 0xF7, 0xC0, 0xFE, 0xDC, 0xEA, 0x92, 0x26, 0xC3, 0x39, 0xB5, 0x8A, 0xCF,
0x83, 0x4A, 0x65, 0x9B, 0xB8, 0x85, 0x10, 0x32, 0xD7, 0xD6, 0x26, 0x77, 0x36, 0xAA, 0xE7,
0xC6, 0x4E, 0x9B, 0xD9, 0x6F, 0x86, 0xF3, 0x1C, 0xA7, 0xCF, 0xDC, 0x5D, 0x67, 0xA1, 0xE6,
0x6C, 0x26, 0x95, 0x3E, 0x4F, 0xA2, 0x8C, 0xFD, 0xBF, 0x77, 0xDA, 0xE0, 0x05]
mersenneTwisterSequence = [0x9D, 0xB5, 0xDF, 0x75, 0x92, 0xC8, 0x67, 0x0B, 0x50, 0x60, 0x0F,
0xB3, 0x4E, 0xEB, 0xD6, 0x67, 0x08, 0xEB, 0x59, 0xE9, 0xCF, 0x7F, 0xF5, 0x39, 0xA4, 0x07,
0xCB, 0xA2, 0xD3, 0x16, 0xC6, 0x93, 0x18, 0x4B, 0x01, 0x04, 0x64, 0xA5, 0x4D, 0xA8, 0x42,
0x7D, 0x24, 0xD0, 0xA8, 0x2B, 0xFB, 0xAF, 0xA1, 0x7D, 0x24, 0x5D, 0x35, 0xEB, 0x3B, 0xDE,
0x4D, 0x64, 0x69, 0xA4]


def decodeFlag(encodedFlag, mersenneTwisterSequence):
    decodedFlag = []
    m = 0
    for e in encodedFlag:
        decodedFlag += [chr(e ^ mersenneTwisterSequence[m] ^
        (((mersenneTwisterSequence[m+1] << 1 ) & 0xff) &
        ((mersenneTwisterSequence[m+2] >> 1) & 0xff)) ^
        (( mersenneTwisterSequence[m+3] << 2) & 0xff) )]
        m += 1
    return "".join(decodedFlag)


print(decodeFlag(encodedFlag,mersenneTwisterSequence))
```

The resulting flag would be:

P0w3rfu1_y0u_h4v3_b3c0m3_my_y0ung_flareaw4n@flare-on.com

Y0da's Advice Revisited

This part is not necessary for getting the flag but serves merely to show a small easter egg in the challenge. When we enter the gimmie_advic3 command **after** entering the correct password into the gimmie_s3cr3t prompt, we'll note that the number of the tip in each Yoda advice varies:

```
C:\Windows>gimmie_advic3
Y0da's life tip #0x5bb76640:
"If no mistake you have made, losing you are. A different game you should play."

            .--.
 ::\`--._,'.::.`._.--'/::
 ::::.  ` __::__ '  .::::
 ::::::-:.`'..`'.:-::::::
 ::::::::\ `--' /::::::::

M4st3r Y0d4 says CSKPfMIDQ5n33Xn4H/Q/aDLHLPD3a2+4weUSXHe+BOTJfIfDng5yCwSUIKZCZaDJCX2H2fnTfw5M5+SPP/LwMMaVJ3======

C:\Windows>gimmie_advic3
Y0da's life tip #0x43be3979:
"No! Try not. Do. Or do not. There is no try."


            .--.
 ::\`--._,'.::.`._.--'/::
 ::::.  ` __::__ '  .::::
 ::::::-:.`'..`'.:-::::::
 ::::::::\ `--' /::::::::

M4st3r Y0d4 says M2wgLBRT54aUXSQMOfTBJeQVyawwRH5gQwLyC/MaD4nUIBg+SUXOMDMwUVRfC53Og4MKQ5UyUZJ4QSVC2yU4Cny+TC======
 C:\Windows>gimmie_advic3
Y0da's life tip #0x7a1786be:
"A Jedi uses the Force for knowledge and defense, never for attack."
```

Figure 42: Varying values that appear in association with Yoda advice after entering the correct gimmie_s3cr3t password

This is no coincidence – these numbers are in fact the result of an LCG stream that is seeded with the value 0x10D4. Each such value is then used as a seed for the Mersenne Twister algorithm when calculating the 60 bytes that are used as a key to encrypt the flag.

```
seed = rand();
qmemcpy(&_mtRand, seedRand(&_seed), 0x9C4ui64);
qmemcpy(mtRand, &_mtRand, 0x9C4ui64);
LODWORD(i) = 0;
while ( i < 60 )
{
  randNum = getRandLong(mtRand);
  flagXorKey[i] = HIBYTE(randNum);
  flagXorKey[i + 1] = BYTE2(randNum);
  flagXorKey[i + 2] = BYTE1(randNum);
  flagXorKey[i + 3] = randNum;
  LODWORD(i) = i + 4;
}
```

Figure 43: LCG result used to seed the Mersenne Twister sequence generation prior to flag encryption

```
  " \\ \\   / / _ \\    | |          / ____| |       | | |\r\n"
  "  \\ \\_/ / | | | __| | __ _ _____| (___ | |__   ___| | |\r\n"
  "   \\   /| | | |/ _` |/ _` |______\\___ \\| '_ \\ / _ \\ | |\r\n"
  "    | | | |_| | (_| | (_| |      ____) | | | |  __/ | |\r\n"
  "    |_|  \\___/ \\__,_|\\__,_|     |_____/|_| |_|\\___|_|_|\r\n"
  "                                                       \r\n"
  "                                                       \r\n");
v38 = 0i64;
v39 = 0i64;
hModuleWs2_32 = (getProcAddressAndExecute)(0, 0x726774C, s_ws2_32_dll);
v43 = (getProcAddressAndExecute)(0, 0x726774C, &v12);
if ( !v42 || !v43 )
  return 0xFFFFFFFFi64;
_ps = getProcAddressAndExecute(0, 0xE553A458, 0, 88, 4096, 4);
if ( !_ps )
  cleanupAndExit(0i64);
memset(_ps, 0, 0x58ui64);
srand(0x10D4);
```

Figure 44: LCG seed that appears in the main function of the challenge

References

- Time Travel Debugging is now available in WinDbg Preview - Windows Developer Blog
- Deep dive into the TTD ecosystem | Elastic
- WinDbg — the Fun Way: Part 1. A while ago WinDbg added support for a… | by Yarden Shafir | Medium
- WinDbg — the Fun Way: Part 2. Welcome to part 2 of me trying to make… | by Yarden Shafir | Medium
- GitHub - mandiant/capa: The FLARE team's open-source tool to identify capabilities in executable files.
- GitHub - mandiant/flare-vm: A collection of software installations scripts for Windows systems that allows you to easily setup and maintain a reverse engineering environment on a VM.
- VirusTotal - File - 8fa35f1694595aa5b92e67a1105af4cc04703dfbe06e12088e68828c46f99569
- Igor's tip of the week #86: Function chunks – Hex Rays (hex-rays.com)
- ida_funcs API documentation (hex-rays.com)
- Igor's tip of the week #51: Custom calling conventions – Hex Rays (hex-rays.com)
- GitHub - snus-b/Metasploit_Function_Hashes