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:

![Static strings output](image)

**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.d11` (SHA256: **8FA35F1694595AA5B92E67A1105AF4CC04703DFBE06E12088E68828C46F99569**) that is typically found in the `%SYSTEM%` directory. In fact, the string `\System32\ApplicationFrame.d11` 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:
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:

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:
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:

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.
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.

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

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.
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:

```plaintext
start:
  sub    ebp, 28h
  call   sub_140012610
  add    ebp, 28h
  retn

;...
;<code>
;...
sub_140012610:
  push    ebp
  mov     ebp, esp
  and     ebp, 0FFFFFFFFFFFFFFF0h
  sub     ebp, 20h
  call    sub_140001010
  mov     ebp, esp
  pop     ebp
  retn
```

After obfuscation, it will look like this:
loc_180004DED:
    mov   rdi, 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   rdi, 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   rdi, 28h
    jmp   loc_1800136A2
    ;...
    ;<junk\other code>
    ;...
loc_1800340C9:
    add   rdi, 28h
    jmp   loc_180016D82
    ;...
    ;<junk\other code>
    ;...
loc_180034E4F:
    sub   rdi, 20h
    jmp   loc_18005DFEF
    ;...
    ;<junk\other code>
    ;...
loc_18005A0B8:
    mov   rdi, rsi
    jmp   loc_1800251F2
    ;...
    ;<junk\other code>
    ;...
loc_18005DFEF:
    call  sub_18003CEDE
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.
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:

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.

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:

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 thefastcall convention, i.e., via rcx, rdx, r8 and r9:
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 = 0x18004EE07
second_thread_ea = 0x18004928C

# Global data structures
function_calls = []
conditional_jumps = []
conditional_jumps_mnem = ['"jo","jno","ja","jns","je","jz","jne","jn","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

            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):
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:

```c
strcpy(_wo2_32_dll, "wo2_32.dll");
strcpy(_user32_dll, "user32.dll");
strcpy(_SYSTEMROOT, "SYSTEMROOT");
strcpy(_yodaAscii,
    &yodaAsci,
```
The decompiled code of this function after we apply this definition will then look like this:

```
for ( 1d Ort = getPEB()->Ldr->InMemoryOrderModuleList.Flink; 1d Ort = *next(1d Ort) )
{
    base01Name = CONTAINING_RECORD(1d Ort, LDR_DATA_TABLE_ENTRY, InMemoryOrderLinks)->BaseDllName.Buffer;
    base01NameMaxlen = CONTAINING_RECORD(1d Ort, LDR_DATA_TABLE_ENTRY, InMemoryOrderLinks)->BaseDllName.MaxLength;
    module0NameHash = 0;
    do
        base01NameChr = base01Name;
        base01Name = (base01Name + 1);  
        if ( base01NameChr == NULL )
            module0NameHash = base01NameChr + __ROR4(__ROR4(module0NameHash, 19), 3);  
            base01NameMaxlen;
    while ( base01NameMaxlen );
    nextDte = 1d Ort;
    c_moduleNameHash = module0NameHash;
    dllBase = CONTAINING_RECORD(1d Ort, LDR_DATA_TABLE_ENTRY, InMemoryOrderLinks)->DllBase;
    dllNIHeaders = &dllBase[*((dllBase + 15)];
    if ( dllNIHeaders->OptionalHeader.Magic == 0x2008 )
        {
            exportDirRva = dllNIHeaders->OptionalHeader.DataDirectory[0].VirtualAddress;
            if ( exportDirRva )
                break;
        }
ABEL 5:
;
exportDir = exportDirRva + dllBase;
numberOfNames = (exportDirRva + dllBase + 24);
address0fNames = &dllBase[*((exportDirRva + dllBase + 32)];
    do
        {  
            if ( !numberOfNames )
                goto LABEL 5;
            numberOfNames;
            exportName = &dllBase[*address0fNames[4 * numberOfNames]];
            exportNameHash = 0;
            do
                {
                    exportNameChr = exportName++;
                    exportNameHash = exportNameHash + __ROR4(__ROR4(exportNameHash, 0), 5);
            }
        while ( exportNameHash );
    while ( c_moduleNameHash + exportNameHash != metasploitNameHashArg )
        {  
            LOWORD(numberOfNames) = &dllBase[2 * numberOfNames + *(exportDir + 9)];
            functionAddr = &dllBase[4 * numberOfNames + *(exportDir + 7)];
            if ( returnAddressMode == 1 )
                
            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:
<table>
<thead>
<tr>
<th>Metasploit Name Hash</th>
<th>Associated API Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xE553A458</td>
<td>kernel32!VirtualAlloc</td>
</tr>
<tr>
<td>0x300F2F0B</td>
<td>kernel32!VirtualFree</td>
</tr>
<tr>
<td>0x528796C6</td>
<td>kernel32!CloseHandle</td>
</tr>
<tr>
<td>0x56A2B5F0</td>
<td>kernel32!ExitProcess</td>
</tr>
<tr>
<td>0x863FCC79</td>
<td>kernel32!CreateProcessA</td>
</tr>
<tr>
<td>0x160D6838</td>
<td>kernel32!CreateThread</td>
</tr>
<tr>
<td>0xBB5F9EAD</td>
<td>kernel32!ReadFile</td>
</tr>
<tr>
<td>0x5B5E572D</td>
<td>kernel32!WriteFile</td>
</tr>
<tr>
<td>0xDDCEADE7</td>
<td>kernel32!GetEnvironmentVariableA</td>
</tr>
<tr>
<td>0x601D8708</td>
<td>kernel32!WaitForSingleObject</td>
</tr>
<tr>
<td>0x726774C</td>
<td>kernel32!LoadLibraryA</td>
</tr>
<tr>
<td>0xEAFCF3E</td>
<td>kernel32!CreatePipe</td>
</tr>
<tr>
<td>0x6558F55E</td>
<td>kernel32!FindResourceA</td>
</tr>
<tr>
<td>0x8E8B14A</td>
<td>kernel32!LoadResource</td>
</tr>
<tr>
<td>0xE8BE94B</td>
<td>kernel32!LockResource</td>
</tr>
<tr>
<td>0x42F9102E</td>
<td>kernel32!SizeOfResource</td>
</tr>
<tr>
<td>0xB33CB718</td>
<td>kernel32!PeekNamedPipe</td>
</tr>
<tr>
<td>Address</td>
<td>Function</td>
</tr>
<tr>
<td>------------</td>
<td>-------------------</td>
</tr>
<tr>
<td>0x614D6E75</td>
<td>ws2_32!closesocket</td>
</tr>
<tr>
<td>0x6B8029</td>
<td>ws2_32!WSAStartup</td>
</tr>
<tr>
<td>0xF4A4E2B</td>
<td>ws2_32!WSACleanup</td>
</tr>
<tr>
<td>0xED83E9BA</td>
<td>ws2_32!socket</td>
</tr>
<tr>
<td>0xFF38E9B7</td>
<td>ws2_32.dll!listen</td>
</tr>
<tr>
<td>0xE13BEC74</td>
<td>ws2_32.dll!accept</td>
</tr>
<tr>
<td>0x6737DBC2</td>
<td>ws2_32!bind</td>
</tr>
<tr>
<td>0x5FC8D902</td>
<td>ws2_32!recv</td>
</tr>
<tr>
<td>0x5F38EBC2</td>
<td>ws2_32!send</td>
</tr>
<tr>
<td>0xD0EB608D</td>
<td>user32!wsprintf</td>
</tr>
</tbody>
</table>

### 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:

![gimmie_advic3 hidden command found in the code of the thread sub_18004e0e7](image)

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:
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:

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:
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.

Figure 26: Decompiled code of the function sub_1800216F6
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.

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.
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).
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.

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`.

<table>
<thead>
<tr>
<th>MD5</th>
<th>Reverse MD5</th>
</tr>
</thead>
<tbody>
<tr>
<td>4C8476DB197A1039153CA724674F7E13</td>
<td>patience</td>
</tr>
<tr>
<td>627FE11EEEF8994B7254FC1DA4A0A3C7</td>
<td>y0u</td>
</tr>
<tr>
<td>D0E6EF34E76C41B0FAC84F608289D013</td>
<td>must</td>
</tr>
<tr>
<td>48367C670F6189CF3F413BE394F4F335</td>
<td>h4v3</td>
</tr>
</tbody>
</table>

We can now rename `sub_18004EBC7` to `checkPassword` and go back to `sub_18001BB76`. After some analysis of the other functions in `sub_18001BB76` 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.
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:
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.

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/UnTPfDMMaHeQXUHMzyp34LfsSgg/HnB5SxV0IyK0BIHeMe45B2KBCeS5T/HRfRHz4SKJe3eLJHeMe5IM5QQJ=======
```
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 Q4T2aSwLnUgHP0IyKBjVm5+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 B3 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.

![Figure 36: Binary patterns in the decrypted resource image.](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.
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.

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

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.
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 first buffer from the JPEG and will invoke sub_18001D361 for each byte:
Figure 40: Loop that processes encoded bytes, likely used to build the flag

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

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:
After the `ret` instruction is invoked, we see the execution of other instructions, wherein each instruction is followed by a `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
movzx    eax, dword ptr [rbp+24h] ; ss:00000000'01d5dff4=00000000
movzx    eax, dword ptr [rbp+50h] ; ss:00000000'01d5e020=0000000001df0000
; 0x9D is the first character in the Mersenne Twister sequence buffer
movzx    eax, byte ptr [rbp+30h] ; 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, eax
movzx    eax, dword ptr [rbp+24h] ; ss:00000000'01d5dff4=00000000
; Next index value (1) is put into ecx
inc      ecx
mov      rax, rax
mov      rax, rax
movzx    eax, dword ptr [rbp+50h] ; ss:00000000'01d5e020=0000000001df0000
; Next character in the Mersenne Twister sequence is 0xB5
movzx    eax, byte ptr [rbp+30h] ; ds:00000000'01df0000=9d
; The character gets shifted left by one bit
shl      eax, 1
and      eax, 0FFh
movzx    eax, dword ptr [rbp+24h] ; ss:00000000'01d5dff4=00000000
; Next index value (2) is put into edx
add      edx, 2
mov      rax, rax
mov      rax, rax
movzx    eax, dword ptr [rbp+50h] ; ss:00000000'01d5e020=0000000001df0000
; Next character in the Mersenne Twister sequence is 0xDF
movzx    eax, byte ptr [rbp+30h] ; ds:00000000'01df0000=9d
; The character gets shifted right by 1
sar      eax, 1
and      eax, 0FFh
and      eax, eax
; The result is XORed with what we calculated thus far
xor      eax, eax
movzx    eax, dword ptr [rbp+24h] ; ss:00000000'01d5dff4=00000000
; Next index value (3) is put into ecx
add      ecx, 3
mov      rax, rax
mov      rax, rax
movzx    eax, dword ptr [rbp+50h] ; ss:00000000'01d5e020=0000000001df0000
; Next character in the Mersenne Twister sequence is 0x75
movzx    eax, byte ptr [rbp+30h] ; ds:00000000'01df0000=9d
; The character gets shifted left by 2
shl      eax, 2
and      eax, 0FFh
; The result is XORed with what we calculated thus far
xor      eax, eax
movzx    eax, dword ptr [rbp+24h] ; ss:00000000'01d5dff4=00000000
; [rbp + 40h] contains the target address to which we write the encoded flag byte
mov      rax, rax
mov      rax, rax
movzx    eax, dword ptr [rbp+40h] ; ss:00000000'01d5e010=0000000001e00000
; The encoded flag byte 0x7F is written to memory
mov      byte ptr [rax+40h], 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:

```python
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
```

**Yoda’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:
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.

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