MANDIANT

YOUR CYBERSECURITY ADVANTAGE

Flare-On Challenge 8 Solution

By James T. Bennett

Challenge 4: My Aquatic Life

Challenge Prompt

What would Flare-On do without a healthy amount of nostalgia for the abrasive simplicity of 1990's UI design? Probably do more actual work and less writing fun challenges like this.

Solution

`myaquaticlife.exe` is a 32-bit x86 Windows executable file weighing in at almost 2.3 megabytes; a pretty hefty size for a challenge binary. A quick glance at the PE headers show that it may be packed with UPX, and successfully unpacking it with the UPX utility confirms this. Now we are at nearly 3mb.

Many of the strings contained in the binary refer to something named Multimedia Builder (MMB), the most telling being the string `Created with Multimedia Builder, version 4.9.8.13`. A quick Web search reveals the product's Web page which has the following to say about it:

"With MMB you can develop autorun menus, multimedia apps, games, or front-ends for your CD's without having to spend months learning complex programming languages.*

<…>

MMB creates small stand-alone exe applications and has many bells & whistles you will ever need. Create a cool looking app or small game and send it to all your friends

<…>

*Our first released product under Mediachance label was Multimedia Builder (aka MMB) in **1998**. Since then we created more than 50 different applications sold in various markets around the world."*

CDs?? 1998!? Clearly, we have come across some ancient technology… how exciting! Based on this information, there is a good chance that this binary was generated by a What You See Is What You Get (WYSIWYG) app building tool. And a closer look at the binary reveals, unsurprisingly, that it uses the Microsoft Foundation Classes (MFC) framework. Our next step is attempting to locate the user-created content within this binary.

The Payload

Running the application presents us with a throwback to the wonder that was the Internet in the 90's (Figure 1).

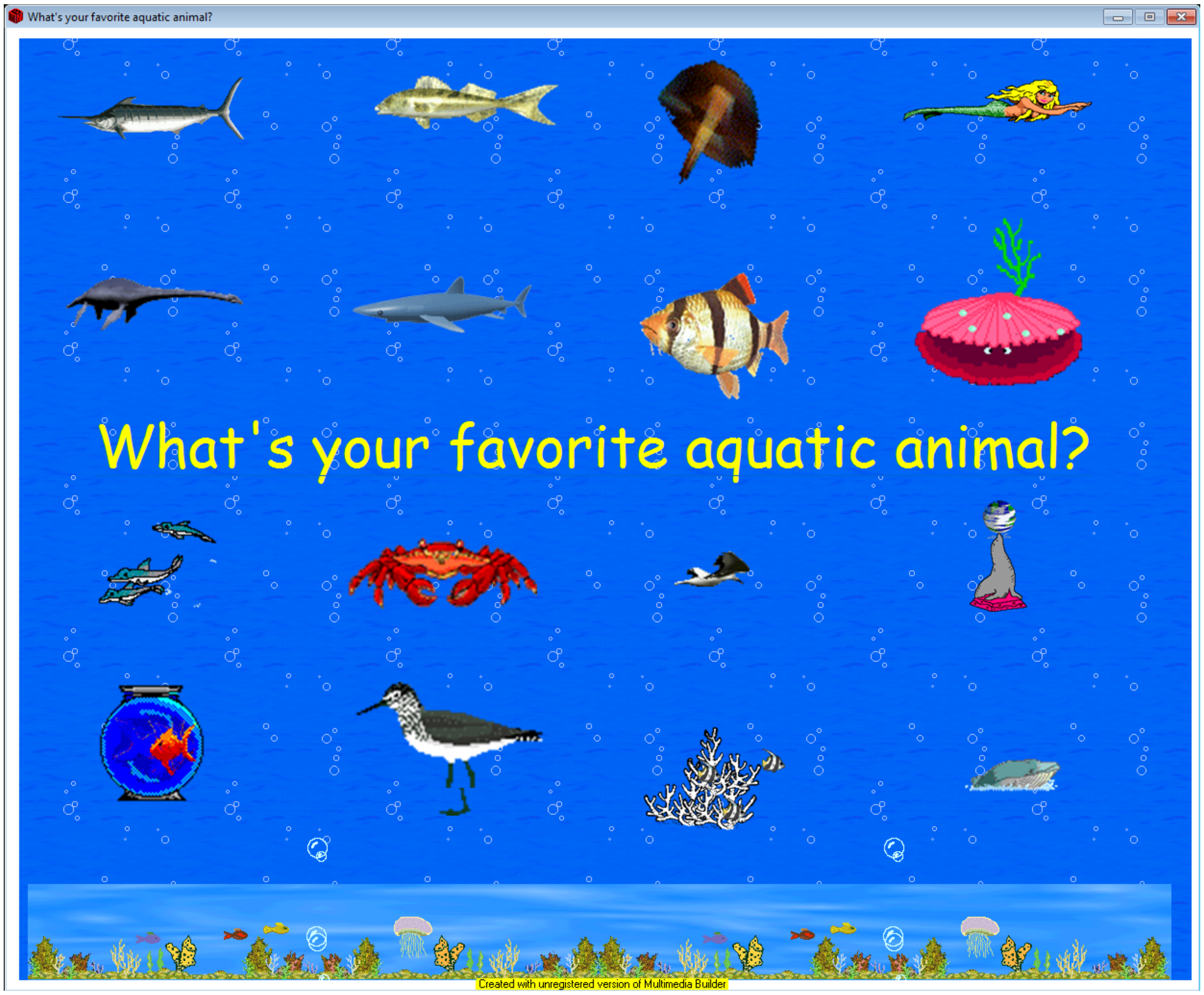
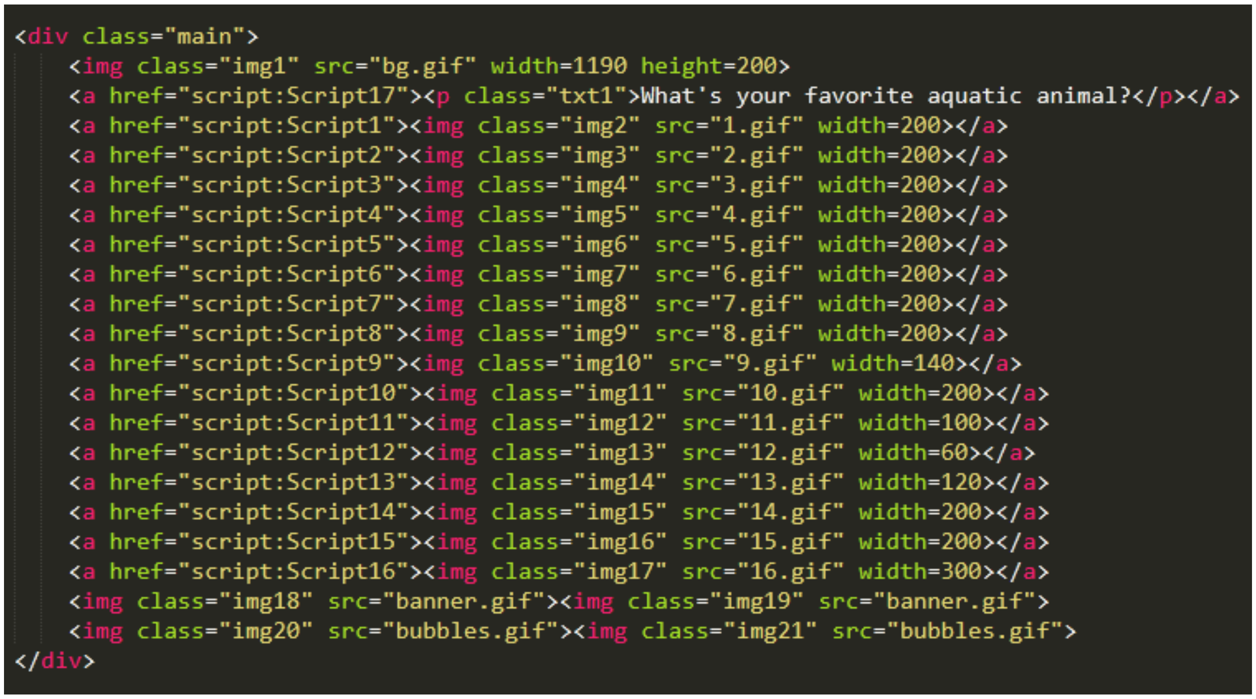

What's your favorite aquatic animal?
What's your favorite aquatic animal?
Created with unregistered version of Multimedia Builder

Figure 1: Screenshot of the initial challenge page

Thankfully, it is quite easy to locate the app's content. When the app is executed, file monitoring tools reveal a directory named `%TEMP%\MMBPlayer` with many files being created within it. Most of these files are images used by the challenge; at first glance, the interesting files appear to be `index.html` and `fathom.dll.`

The HTML File

The HTML file is quite small and contains our first clue towards finding the flag for the challenge. The HTML aligns with what we see in the application, and the images presented in the app are all wrapped in anchor tags. As shown in Figure 2, these tags contain peculiar hrefs that specify different scripts following the naming scheme `Script<number>`.



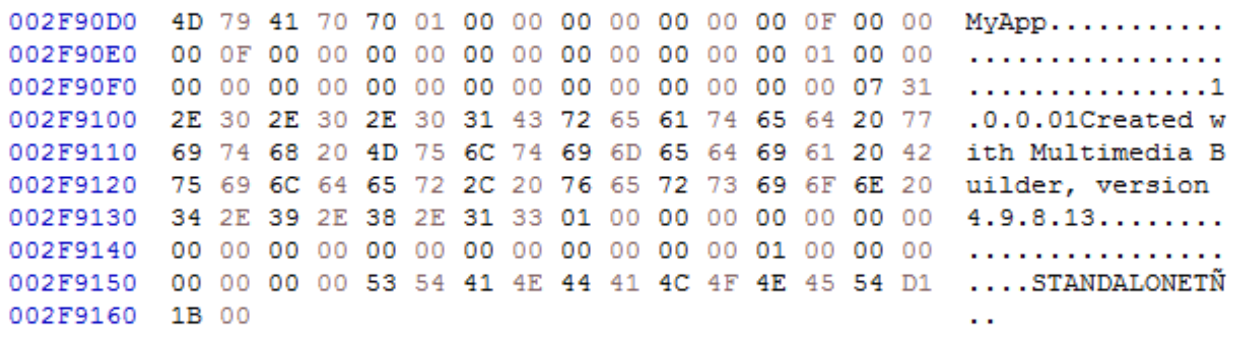
```
<div class="main">
    <img class="img1" src="bg.gif" width=1190 height=200>
    <a href="script:Script17"><p class="txt1">What's your favorite aquatic animal?</p></a>
    <a href="script:Script1"><img class="img2" src="1.gif" width=200></a>
    <a href="script:Script2"><img class="img3" src="2.gif" width=200></a>
    <a href="script:Script3"><img class="img4" src="3.gif" width=200></a>
    <a href="script:Script4"><img class="img5" src="4.gif" width=200></a>
    <a href="script:Script5"><img class="img6" src="5.gif" width=200></a>
    <a href="script:Script6"><img class="img7" src="6.gif" width=200></a>
    <a href="script:Script7"><img class="img8" src="7.gif" width=200></a>
    <a href="script:Script8"><img class="img9" src="8.gif" width=200></a>
    <a href="script:Script9"><img class="img10" src="9.gif" width=140></a>
    <a href="script:Script10"><img class="img11" src="10.gif" width=200></a>
    <a href="script:Script11"><img class="img12" src="11.gif" width=100></a>
    <a href="script:Script12"><img class="img13" src="12.gif" width=60></a>
    <a href="script:Script13"><img class="img14" src="13.gif" width=120></a>
    <a href="script:Script14"><img class="img15" src="14.gif" width=200></a>
    <a href="script:Script15"><img class="img16" src="15.gif" width=200></a>
    <a href="script:Script16"><img class="img17" src="16.gif" width=300></a>
    <img class="img18" src="banner.gif"><img class="img19" src="banner.gif">
    <img class="img20" src="bubbles.gif"><img class="img21" src="bubbles.gif">
</div>
```

Figure 2: Contents of index.html

MMB's documentation states that these hrefs are used to execute Script objects created by the user and contained within the packaged app. A cursory look at the `fathom.dll` file does not reveal any traces of these Script objects, so we turn back to the challenge binary.

The Script Objects

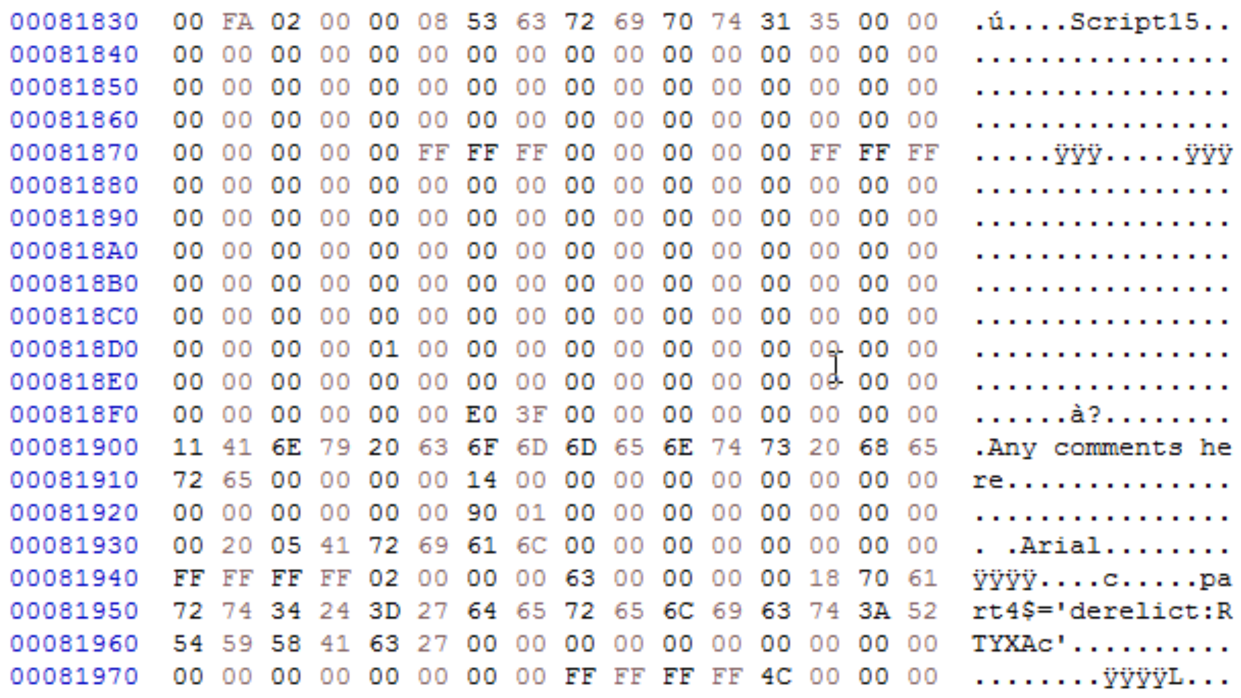
There are several ways in which one could locate the Script objects in the binary. One method could be to set breakpoints on file parsing APIs such as `SetFilePointer` and `ReadFile` and discover that the binary reads some values from the end of its file to locate the embedded project files and objects within itself. <> shows the string `STANDALONE` located towards the end of the file, followed by a 32-bit value that specifies a negative offset from the end of the file to seek to in order to find the start of the embedded objects and files.



```
002F90D0  4D 79 41 70 70 01 00 00 00 00 00 00 00 0F 00 00  MyApp...........
002F90E0  00 0F 00 00 00 00 00 00 00 00 00 00 00 01 00 00  ................
002F90F0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 07 31  ...............1
002F9100  2E 30 2E 30 2E 30 31 43 72 65 61 74 65 64 20 77  .0.0.01Created w
002F9110  69 74 68 20 4D 75 6C 74 69 6D 65 64 69 61 20 42  ith Multimedia B
002F9120  75 69 6C 64 65 72 2C 20 76 65 72 73 69 6F 6E 20  uilder, version 
002F9130  34 2E 39 2E 38 2E 31 33 01 00 00 00 00 00 00 00  4.9.8.13........
002F9140  00 00 00 00 00 00 00 00 00 00 00 00 01 00 00 00  ................
002F9150  00 00 00 00 53 54 41 4E 44 41 4C 4F 4E 45 54 D1  ....STANDALONETÑ
002F9160  1B 00                                            ..
```

Figure 3: EOF contents with payload length indicator

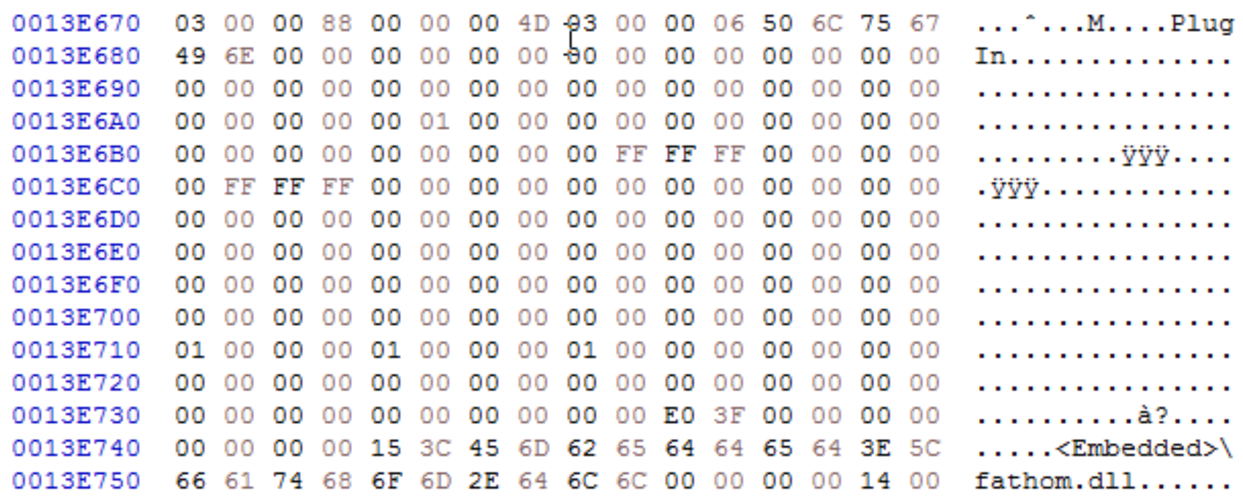
Another way could be to simply search for the term `Script` and find the compiled Script objects. In any case, each "object" in a project is compiled into the binary as a mixture of binary data and strings, as shown in Figure 4. Writing a decompiler is not necessary, however; some simple guesswork will be enough here.



```
00081830  00 FA 02 00 00 08 53 63 72 69 70 74 31 35 00 00  .ú....Script15..
00081840  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00081850  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00081860  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00081870  00 00 00 00 00 FF FF FF 00 00 00 00 00 FF FF FF  .....ÿÿÿ.....ÿÿÿ
00081880  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00081890  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000818A0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000818B0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000818C0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000818D0  00 00 00 00 01 00 00 00 00 00 00 00 00 00 00 00  ................
000818E0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000818F0  00 00 00 00 00 00 E0 3F 00 00 00 00 00 00 00 00  ......à?........
00081900  11 41 6E 79 20 63 6F 6D 6D 65 6E 74 73 20 68 65  .Any comments he
00081910  72 65 00 00 00 00 14 00 00 00 00 00 00 00 00 00  re..............
00081920  00 00 00 00 00 00 90 01 00 00 00 00 00 00 00 00  ................
00081930  00 20 05 41 72 69 61 6C 00 00 00 00 00 00 00 00  . .Arial........
00081940  FF FF FF FF 02 00 00 00 63 00 00 00 00 18 70 61  ÿÿÿÿ....c.....pa
00081950  72 74 34 24 3D 27 64 65 72 65 6C 69 63 74 3A 52  rt4$='derelict:R
00081960  54 59 58 41 63 27 00 00 00 00 00 00 00 00 00 00  TYXAc'..........
00081970  00 00 00 00 00 00 00 00 FF FF FF FF 4C 00 00 00  ........ÿÿÿÿL...
```

Figure 4: Compiled script object

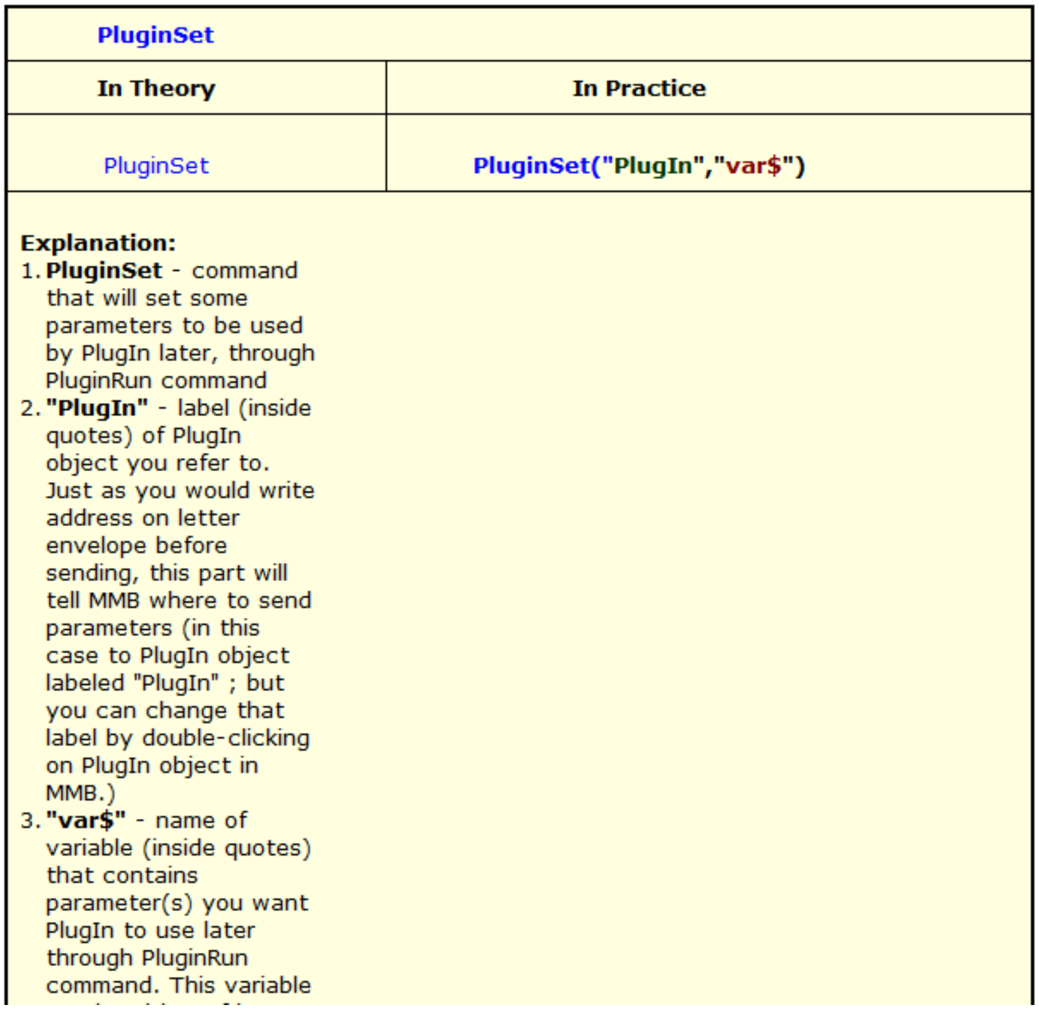
The names of the scripts can be seen in this area of the binary, each followed by some strings that appear to be a part of the script. Some variables with names like `part1$, part2$,` etc., are being assigned strings that are colon-delimited into two parts. The first part is always one of the following strings: `flotsam, jetsam, lagan, derelict`. The second part is some string data. After each of these variable assignments is always the string `PlugIn` followed by `<var name>,` where `<var name>` is the name of the previously assigned variable. These look to be arguments to a function, but which function? `PlugIn` is an interesting string, and we did observe the `fathom.dll` file as part of the unpacked payload in the temp directory earlier. A little further digging confirms this theory, as shown in Figure 5. The object labeled `PlugIn` assigns the embedded file `fathom.dll`.



```
0013E670  03 00 00 88 00 00 00 4D 03 00 00 06 50 6C 75 67  ...ˆ...M....Plug
0013E680  49 6E 00 00 00 00 00 00 00 00 00 00 00 00 00 00  In..............
0013E690  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0013E6A0  00 00 00 00 00 01 00 00 00 00 00 00 00 00 00 00  ................
0013E6B0  00 00 00 00 00 00 00 00 00 FF FF FF 00 00 00 00  .........ÿÿÿ....
0013E6C0  00 FF FF FF 00 00 00 00 00 00 00 00 00 00 00 00  .ÿÿÿ............
0013E6D0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0013E6E0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0013E6F0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0013E700  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0013E710  01 00 00 00 01 00 00 00 01 00 00 00 00 00 00 00  ................
0013E720  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0013E730  00 00 00 00 00 00 00 00 00 00 E0 3F 00 00 00 00  ..........à?....
0013E740  00 00 00 00 15 3C 45 6D 62 65 64 64 65 64 3E 5C  .....<Embedded>\
0013E750  66 61 74 68 6F 6D 2E 64 6C 6C 00 00 00 00 14 00  fathom.dll......
```

Figure 5: PlugIn object

It appears these variables are being fed to the plugin via some function, but unfortunately the function name appears to be encoded as binary data here. Some study of the MMB documentation (Figure 6) strengthens this theory and provides us with some more helpful information: the name of the function is `PluginSet`, the first argument is the label of the plugin you wish to interact with, and the second argument is a variable containing the data you wish to send to the plugin for later interaction using the function named `PluginRun`.



PluginSet	
In Theory	**In Practice**
PluginSet	**PluginSet("PlugIn","var$")**

Explanation:
1. **PluginSet** - command that will set some parameters to be used by PlugIn later, through PluginRun command
2. **"PlugIn"** - label (inside quotes) of PlugIn object you refer to. Just as you would write address on letter envelope before sending, this part will tell MMB where to send parameters (in this case to PlugIn object labeled "PlugIn" ; but you can change that label by double-clicking on PlugIn object in MMB.)
3. **"var$"** - name of variable (inside quotes) that contains parameter(s) you want PlugIn to use later through PluginRun command. This variable

Figure 6: MMB PlugIn documentation

The Plugin – fathom.dll

A quick look at the strings for `fathom.dll` reveals some familiar friends: `flotsam, jetsam, lagan, and derelict`. They are all referenced in the same exported function: `SetFile`. This function splits the received string by the colon delimiter and appends the second part of the split string to a specific global string variable, depending on the value of the first part of the split string. Essentially, the first part of the colon delimited string is the name of a variable to be appended with the string passed as the second part. The strings are of the MFC type `CString`, which are quite messy to reverse engineer, but some debugging helps with this.

Checking the cross-references to these global variables, we find that only `flotsam` and `jetsam` are referenced in another export. The export `PluginFunc19` uses these two strings as part of its decryption routine. There does not seem to be anything else interesting in the other exports. Could this be our flag? `PluginFunc19` is passed as an argument to a function within the Script object `Script17`, which is set to be executed when the user clicks on the text `What's your favorite aquatic animal?`.

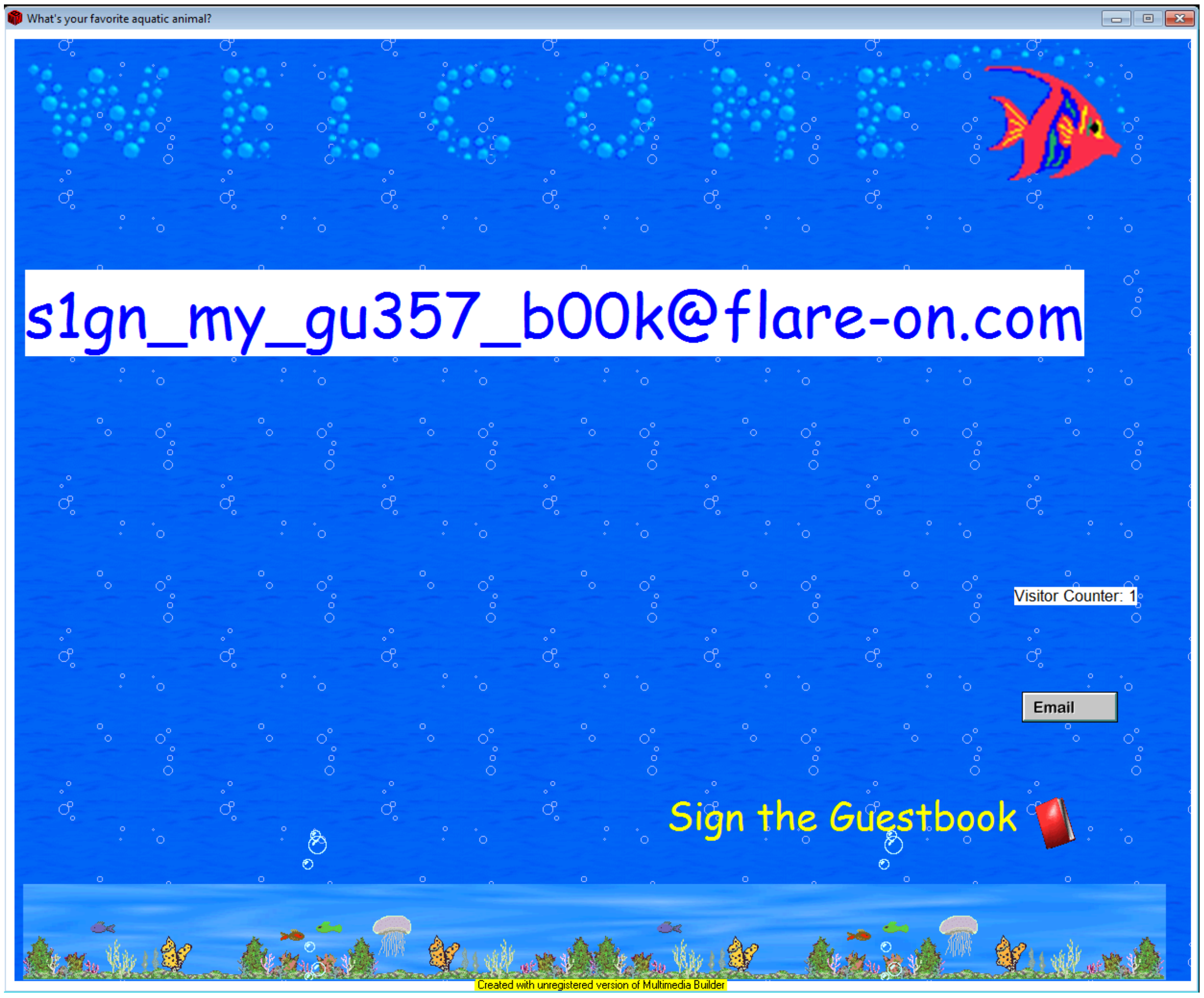
Putting It All Together

The clues we have so far:

1. Each image in the app executes a specific script when clicked.
2. Each script appends a string to a global string variable in the plugin, and have names like `part1$`, `part2$`, etc.
3. Only the `flotsam` and `jetsam` variables are used by the `PluginFunc19` export, which is called when clicking on the `What's your favorite aquatic animal?` link.

With these clues, one could deduce that clicking on the images in the right order, as specified by the Script object variable names (`part1$`, `part2$`, etc.), would build the proper strings in the plugin required to decrypt the flag when the text link is clicked.

In fact, the ordering is not entirely strict. Either `flotsam` or `jetsam` can be built first. Clicking the images in the proper order yields us victory as shown in Figure 7.


What's your favorite aquatic animal?
WELCOME
Visitor Counter: 1
Email
Sign the Guestbook
Created with unregistered version of Multimedia Builder

s1gn_my_gu357_b00k@flare-on.com

Figure 7: Victory!