Lab 05 – Exploit Kits and Landing pages

**Note: This lab contains malicious code, do not open in your browser unless in a controlled lab environment.**

Your goal for this lab is to analyze the artifacts provided to understand how Angler works. I recommend the following tools to assist in your analysis:

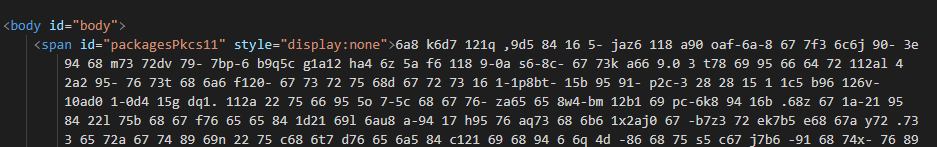
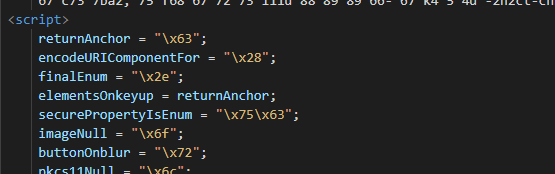
* JSDetox (REMnux)
  + <https://docs.remnux.org/run-tools-in-containers/remnux-containers#jsdetox>
* De4Js (REMnux)
  + <https://docs.remnux.org/run-tools-in-containers/remnux-containers#de-4-js>
* JS Beautifier or similar
* Developer Tools from the browser
* Text editor (e.g. Sublime)
* Wireshark
* Network Miner

Apologies for the length – I thin the screencaps really make it seem longer than it is.

**Task 1**

**Step 1 - Finding the Injected Script**

This task requires the use of *original\_page.txt.* Analyze this page and identify the injected script. Deobfuscate this first stage and explain how it works. Explain this in detail, using textual descriptions alongside relevant screenshots.

After reformatting, around line 465 that you’ll find a curious span with a “display:none” style tag – followed immediately by what appears to be an elaborate display of string manipulation, which is followed by a homebrewed function call.

The definition of letElements() traces back:

elementsOnKeyUp is a constructor, at the line :”imageNew = [][elementsOnKeyUp],

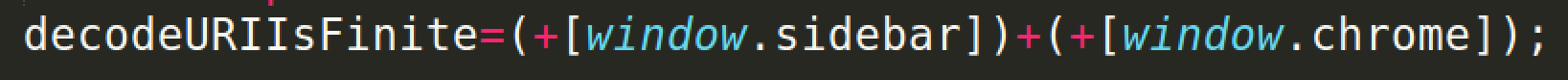


here, lastElements is assigned the function type, by some magic using the constructor in the array.

And here, throwReturn contains the codeblock presumably executed by this syntax from letElements.

**Step 2 - The Next Stage**

The next round of JavaScript performs more fingerprinting and, of course, more deobfuscation. You’ll know when you have this script when you see something that looks **similar** to but is **slightly different** to the following code:



Deobfuscate this code and explain how it works.



the first row of this chunk of code is pulling the Dom for the contents of the span above the script tag, and performs a replace and split on that block of gibberish.

The next bit is a for loop.

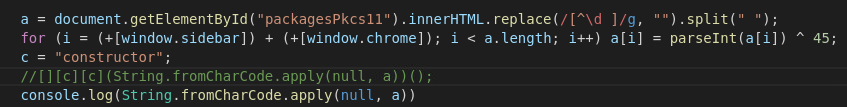
The + operator turns whatever follows it into a number, so it is turning the expressions +[window.sidebar] and +[window.chrome] into integer values used to initialize the iterator, controlling the loop. Window.sidebar checks if the browser is firefox (though it is completely removed in modern versions of firefox.) and window.chrome checks if the browser is chrome (microsoft edge also has this window.chrome object).

Each should return a nonzero value (the object) when invoked this way - if the browser matches and the object can be fetched. I can’t predict how exactly those objects would pan out during the conversion to integers, but If the iterator is nonzero then there will be an offset to the contents being pulled from the span element.

This control mechanism gives no clue as to the target system of the malware, but it will only run on the targeted browser (unless the window.sidebar and window.chrome objects are converted to the same integer value (which I doubt).

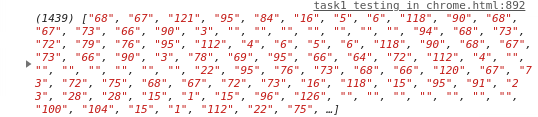
The loop then iterates over the data block and converts it to this

The last part of this builds another function, as in the previous task, builds another function prototype using that array constructor technique. It (inline) converts the interpreted content from their charcodes to a string, and passes it to this brand new function-type object, and executes it with apply().

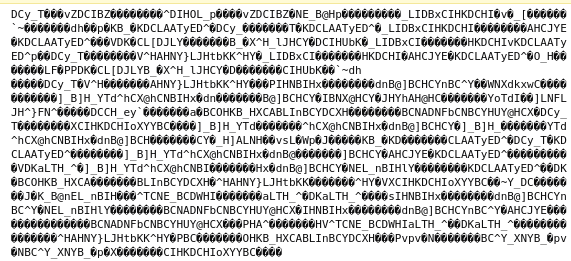
**as a note:** I installed these browsers on the remnux – not using in my personal environments

This prints gibberish in both versions of updated firefox and chrome

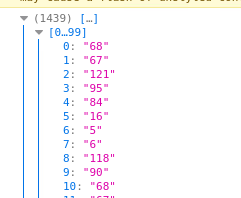
🡨 Chrome



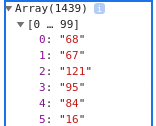
🡨 value of a on chrome



🡨 Firefox

****

🡨 value of a on firefox …looks the same :shrug:

****

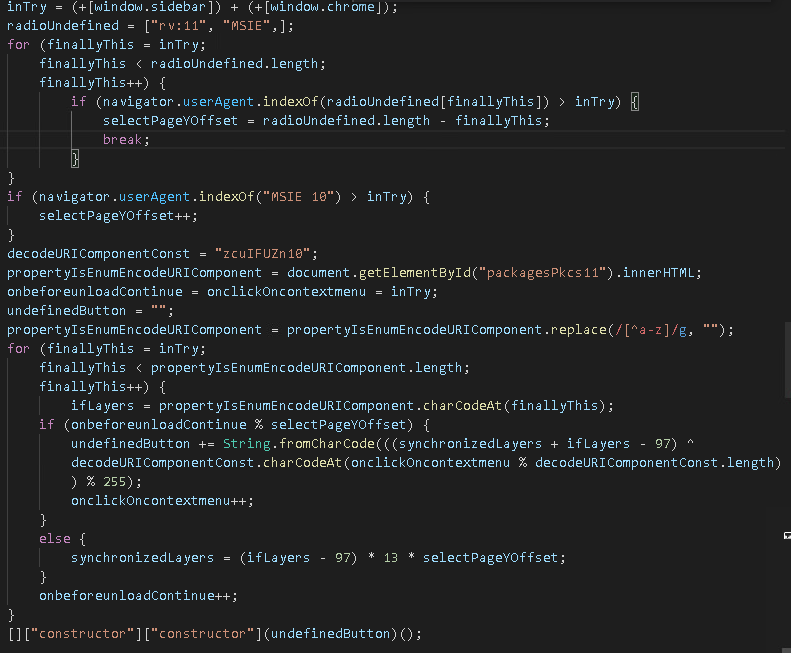
🡨 value of a on opera.hmmm.

I may need to rethink my theory.  
about the browser checks affecting the offset….

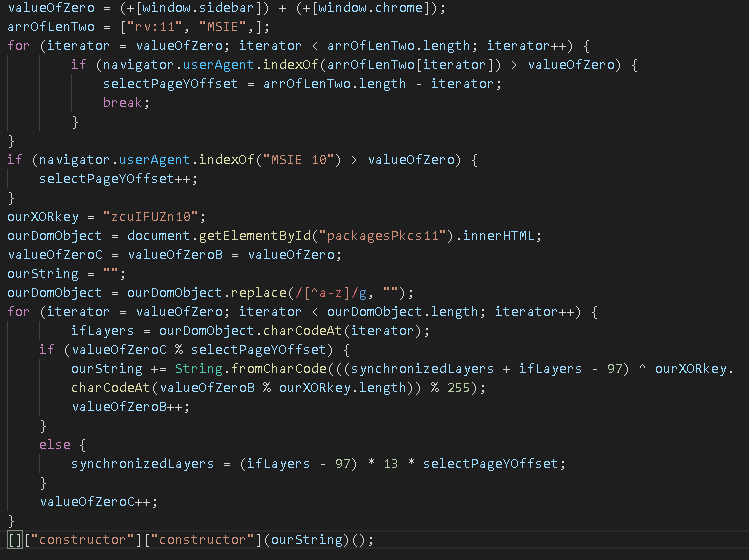
Perhaps the chrome.window object has become more ubiquidous, defeating the purpose of the gate? Hmmm. Not sure what to make of this -> moving on.

Old School internet explorer came through.

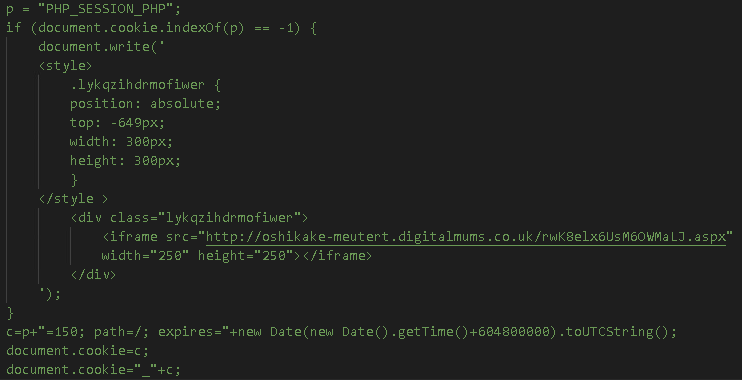
Cleaned up the output looks like this

****This output from internet explorer was cut off towrads the end. I replaced the fingerprinting gates with their numerical equivalents and ran it again in chrome to get the full code snippet:

Cleaned up:



Comment out the function call at the end, and then print ourString which should contain the next code block.  
we get this:

Presumably this iframe is pulling from our landing page addressed in the next step.

**Step 3 - The Landing Page**

The next stage is the landing page, this is primarily responsible for delivering the exploit. Analyze the landing page, can you identify patterns, trends or functionality? Do you see anything going on with the DOM? You can inspect changes made with inner html by using Inspector. If so, yes or no, justify your response with evidence. You do NOT need to attempt to deobfuscate this page. Do respond to each of the questions here.

Do not spend too much time on this landing page (we will work more on task 3 landing page). We are just trying to get a feel for how it works, but not trying to deobfuscate it.

can you identify patterns, trends or functionality?

Just skimming through, I see 5 script tags, doing a lot of string manipulations.

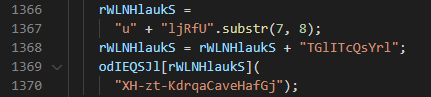
 Inside the strings I see things that are probably part of a script to be executed, so it’s obfuscated away from us.

In some of the later script tags there apperas to be simple string manipulation, but then some odd structures that at first glance wouldn’t stand out. From this class I recognize those as an obscure way of building and executing functions built at runtime”

I’m also noticing that the loewr script tags recycle this “rWLNH” string variable, modifying it ,and then calling it again.







Do you see anything going on with the DOM?

I can see this page pulling data from this page’s less coherent elements, processing and building code strings from them. There’s also a number of encrypted prepared strings being used.

Hard to make out much sense from all the splitting in the first tag, but one interesting string stood out to me that just straight said “push” – curious if that is pushing data to a server, pushing something to a stack structure – maybe it’s nothing.

Note: sometimes with complex code, JSDetox can “break” the code, by introducing syntax or other errors, so keep that in mind as you perform your analysis.

**Task 2 – Landing Page**

* Look at the large *chunk* of obsfuscated code at the top. Can you find below where it is deobfuscated? You can log it to console to get to the next intermediate stage.
* This newly deobfuscated intermediate stage is difficult to understand due to automated obfuscation. However, we should be able to identify perhaps the type of vulnerability it is looking for, by seeing that it is trying to determine a particular version of a plugin. After all, would it be looking for something, if it were not vulnerable?
* You need not look at the landing page beyond this!
* Open the PCAP in Wireshark and Network Miner. What is the malicious payload? Was it what you suspected?

**Task 3 – Compromised Site**

* Not much is done or required to load the evil landing page. Show the excerpt of injected code responsible for this.
* What is the URL of the landing page?

**Task 3 ­­-- Deobfuscating the Landing Page**

**Do this in Remnux**

* The malware is doing something evil with innerHTML by playing with the DOM. Focus on line 97 with ptYabkqBDC and start to deobfuscate there. Notice the array and what is going on below that. Can you log any of that to console? What do you see?
* Note that if you output it too early, you may encounter something that is still in an obfuscated state. That is okay! Just continue to try to get what is going on with this array into a fully deobfuscated state. Show it deobfuscated.
* We will consider the result of the above an intermediate next stage of the malware. You will know you have reached it when you get a var that is equal to **“"cyGwivu8DpAPrvOZT4B61S0xD” (**truncated – it is a long text). Now try to deobfuscate this intermediate stage, by logging something to console. If successful, it will give you an icon, and you hover over it, it will tell you that you can view this in Inspector. Click on it, and you will see a **<script type=”text/javascript”>…</script>.** Click on the …and, you will then be able to determine what its malicious payload is.
* It links to two different payloads – what are they?

**Deliverables**

* Turn in your lab report as a Word document or PDF.
* The document need not to be lengthy, but should fully answer any questions posed.