Defense Against Content-Sensitive Web Bots

Background:

Web bots are programs that simulate user browsing behavior. They read the HTML code, analyze the content and interact with the web app just like humans. Web bots are commonly used for various purposes such as searching, scraping and impersonation. Bots can be roughly classified into two groups based on the targets. Some do not focus on particular items but grab all contents. Bots targeting search engines are examples of this type. The bots in the other group focus on specific elements. They parse the HTML and locate the targets using predefined patterns. Once found, they either simulate human behaviors (e.g., clicking buttons) or extract valuable data. Data theft by web scraping and human impersonators are typical examples. Since being able to locate the targets is important, we dub them content-sensitive web bots.

A Content-Sensitive Web Bot on Lenovo Website:

Content-sensitive web bots are widely used on Ecommerce websites. Take a bot targeting at the Lenovo outlet store as an example [1]. The Lenovo outlet store offers substantially discounted computers but the quantity is limited. It is usually hard to get one since many people keep refreshing the inventory and grab a deal as soon as it becomes available. While it is tedious for a human to repeat this procedure, a bot was programmed to monitor the store and add deals to the shopping cart automatically.

Fig. 1 shows the HTML code of the “Add to cart” button. It is an element representing a link. There are totally 109 elements on this page.

```
1. <html>
2.   ...
3.   <a class="button-called-out button-full" href=
   "//outlet.lenovo.com/.../?sb=:000001BD:0002F49B:">Add to cart</a>
4.   ...
5. </html>
```

Figure 1: The source of the “Add to cart” button. The page has totally 109 an elements but only one “Add to cart” button.

Fig. 2 shows a snippet of the web bot. The script loads a product page at line 1. It hence tries to locate the “Add to cart” link. Since there are many elements, the script has to distinguish the target from the others. It does so by comparing the style class name and the element id of a candidate link with some patterns. In this case, at line 2, the script uses two style class names “button-called-out” and “button-full” as the signature. If such an is found, at line 4, it further extracts the id from the content after “sb=” in the link. In this example, the itemid is “:000001BD:0002F49B:”. Then it constructs the actual addto-cart link at line 5 and invokes the browser at line 6 to add the item to the shopping cart.

```
1. content = open_url(item_url_full,...)
2. tmp_found = re.findall(r"<a class=\"button-called-out buttonfull\"(.+?)Add to
   cart", content, ...)
3. if len(tmp_found) != 0:
4.   itemid = re.findall(r"\?sb=(.+?)\"", tmp_found[0], ...)
5.   new_addtocart_url = '//outlet.lenovo.com/...AddToCart? addtocart-itemid='+
   itemid[0]
6.   webbrowser.open(new_addtocart_url)
```

Figure 2: A snippet of a web bot for the Lenovo Outlet.

As observed above, a critical precondition for content-sensitive web bots is that they identify important DOM objects by pattern matching. In order to defend against content-sensitive web bots, we randomize the class of the element, i.e., mapping the value of the class to randomized values (e.g., “button-called-out” is randomized to “rand-class-1234”).

Project Description:

The web page of the Lenovo website is located at: https://weihang-wang.github.io/cse705/project/lenovo/Y40_80.html The scripts needed for verifying your result can be downloaded: Config and Adding to cart.

1. Download the web page of the Lenovo website, and host the web page within a subfolder under your personal website, e.g., www.cse.buffalo.edu/~weihangw/cse705/Y40_80.html Information on how to create your own web page with CSE’s server: https://wiki.cse.buffalo.edu/services/content/how-create-your-cse-home-page

2. Randomize the class of all elements on the web page to a randomized value starting with “rand-class-”, followed by 4-digit random integer, e.g., “button-called-out” is randomized to “rand-class-1234”, “button-full” is randomized to “rand-class-7623”.

   (1) In order to locate all elements, you need an HTML parser to parse the HTML page into a DOM tree. We will use htmlparser2 for this purpose (https://www.npmjs.com/package/htmlparser2).

   // Read input html page as string
   var html_str = fs.readFileSync(input_file_with_path, 'utf-8');

   // Transform input html page to DOM tree, the DOM tree is stored in handler.dom
   var handler = new htmlparser2.DomHandler();
   var HTMLparser = new htmlparser2.Parser(handler);
   HTMLparser.parseComplete(html_str);

   // Write a recursive function to walk the DOM tree (visit each element and its children):
   // whenever an element is , randomize its class attribute.
   walkDOM(handler.dom);

   (2) Once the class attributes are randomized, you need to convert the DOM tree back to the HTML page, so that a randomized HTML page can be rendered by the browser. For this purpose, we use the nodejs package htmlparse-to-html (https://www.npmjs.com/package/htmlparser-to-html).

```
var html = require('htmlparser-to-html');

// Transform DOM tree back to HTML
var randomized = html(handler.dom);
try {
fs.writeFileSync(output_path, randomized);
}
catch (err) {;}
```

3. After the html page is randomized, the CSS styling formats for each element are broken. To solve this problem, in each CSS file, change the class of to the same randomized string used in the HTML.
   To achieve this, recursively process all CSS files under the folder “Y40_80_files/”. For each CSS file, use a CSS parser/stringifier (https://github.com/reworkcss/css) to parse a CSS string to an AST object, change the class name of elements, and then stringify the AST object to a CSS string.

4. Test your results.
   ```
   ========================================================================
   (1) Configuration:
   ========================================================================
   In "config.py":
   * variable "useOriginal":
   -- Set value to 1: original version of the web page
   -- Otherwise: randomized version of the web page

   * variable "baseURL":
   -- The baseURL to the original and randomized versions
   Once set, page baseURL + "/lenovo/ori/Y40_80.html" should be
   accessiable

   ========================================================================
   (2) How to run:
   ========================================================================
   (a) Set "baseURL" in "config.py" and make it point to the server
   (b) Set "useOriginal = 1" in "config.py" to use the original web
   page
   (c) run "python ./lenovo_mod.py" (please see the sample output
   below)
   (d) Set "useOriginal = 0" in "config.py" to use the randomized
   web page
   (e) run "python ./lenovo_mod.py" again (please see the sample
   output below)

   ========================================================================
   (3) Sample output:
   ========================================================================

   ************************
   (3.1) Original version
   ************************
   1.lenovo$ python ./lenovo_mod.py
   URL adding the laptop to the cart:
   ```

```
http://outlet.lenovo.com/outlet_us/builder/?
sb=:000001BD:0002DA7A:
URL extraction done

************************
(3.2) Randomized version
************************
1.lenovo$ python ./lenovo_mod.py
The deal was found but lost before adding to shopping cart...
```

Note: After the web page is randomized, the functionality of the website may be broken. This is because the JS code cannot access original elements directly. Moreover, even after you match the CSS styles with HTML elements, the web page appearance may be still broken because some styling format are added dynamically. To keep the web page appearance persistent, you need to overwrite the JavaScript APIs that locate HTML elements and the APIs that create new HTML elements.

For example, Figure 3 shows the code snippet to override the JavaScript API that locate elements by class name. Whenever the web page program looks up elements by class name, this override version will be invoked first. It checks the mapping between the original class and the randomized class. If a mapping exist, that means the class was randomized, then the randomized string will be used to invoke the native implementation. Otherwise, the original class is used for the function call.

```
8.  var byClass = document.__proto__.getElementsByClassName;
9.  document.__proto__.getElementsByClassName = function(class) {
10.   if (classMap[class]) {
11.     return byClass.call(document, classMap[class]);
12.   }
13.   else {return byClass.call(document, class);}
14. };
```

Figure 3: override the JavaScript API that locate elements by class name.

Due to the heavy load of implementation (override every function and attribute getter/setter that could possibly affect the behavior of the program), the dynamic loading part is not required for this project. i.e., you only need to finish the 4 steps in "Project Description" section.

[1] A simple python crawler for Lenovo outlet website.
https://github.com/agwlm/lenovo_crawler