

PhishingJS: A Deep Learning Model for JavaScript-Based Phishing Detection

3,381 people reacted

👍 11

7 min. read

SHARE



By Lucas Hu

September 10, 2021 at 6:00 AM

Category: Unit 42

Tags: Cybercrime, deep learning, JavaScript, Machine Learning, Phishing, Scams, URL filtering



This site uses cookies essential to its operation, for analytics, and for

Executive Summary

Many traditional phishing detection systems rely on the presence of login forms within the HTML of the webpage to classify pages as phishing. [Client-side cloaking](#) is an increasingly common technique that attackers use to evade phishing detection systems by requiring complex user interaction before revealing the webpage's phishing content, or by using JavaScript to inject phishing content into the page only after the page renders in the user's browser.

To combat these evasion techniques, we trained a deep learning model to detect phishing webpages based on the JavaScript content contained within the script tags of an HTML page. The model, dubbed PhishingJS, runs in the Palo Alto Networks cloud-delivered [Advanced URL Filtering](#) service. It currently detects an additional 15,000 phishing URLs per week on sophisticated JavaScript-based attacks that many existing phishing detection systems struggle to detect.

Palo Alto Networks customers with the [Next-Generation Firewall](#) and the Advanced URL Filtering security subscription are protected against the sophisticated types of phishing attacks discussed here.

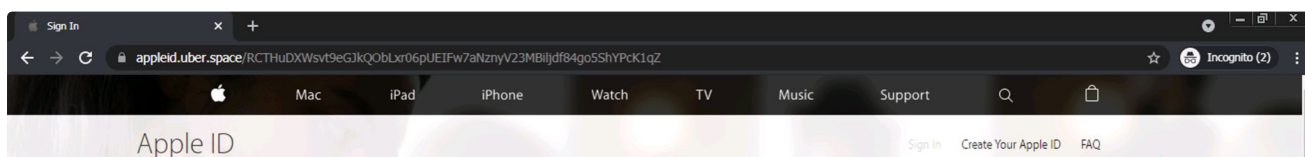
Client-Side Cloaking

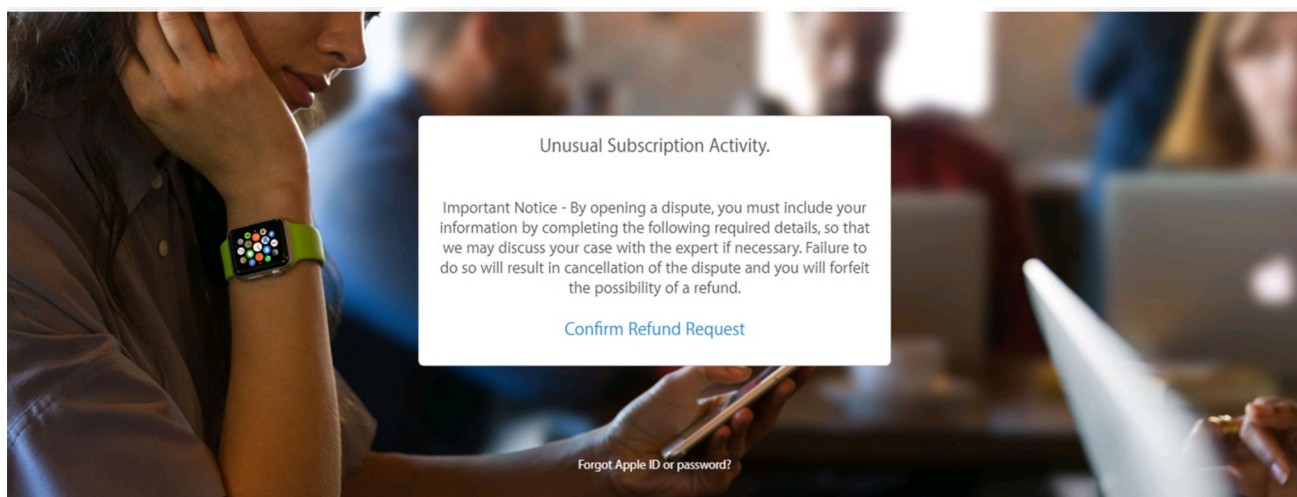
Many existing phishing detection systems rely on the presence of login forms, brand logos and similar signals within the HTML of a webpage to determine whether the page is a phishing page.

[Client-side cloaking](#) is an increasingly common evasion technique used by cybercriminals to evade these phishing detection systems. These attackers often use JavaScript to dynamically render phishing content (the same types of content often picked up by detection systems) on the client side, sometimes requiring user interaction before revealing any signs of attempted credential theft. Since the phishing content does not appear in the HTML document until after the page renders, these phishing pages can be difficult for traditional phishing detection engines to catch.

In Figures 1-2, we can see an example of a phishing webpage employing client-side cloaking techniques. This page claims that there has been some unusual activity on the user's Apple ID account and that the user needs to process a refund related to the activity.

Upon first visiting the webpage, there is no phishing form immediately apparent. Only after the user clicks the "Confirm Refund Request" button is the credential-stealing form revealed. Since many crawler-based phishing detection systems cannot handle these sorts of interactions, these types of phishing pages can often pass undetected.





Your account for everything Apple.

Figure 1. [https://appleid\[.\]uber\[.\]space/LoginFailed.php?sslchannel=true&sessionid=VRqELCgYcqDhDQ2KGKMPYt4FcEXO7Kz1wi2xZyCCQiA5fz9smSqmoWfnsEn9znu4ixcl3RkVzwRW5f](https://appleid[.]uber[.]space/LoginFailed.php?sslchannel=true&sessionid=VRqELCgYcqDhDQ2KGKMPYt4FcEXO7Kz1wi2xZyCCQiA5fz9smSqmoWfnsEn9znu4ixcl3RkVzwRW5f). A phishing page employing client-side cloaking techniques.

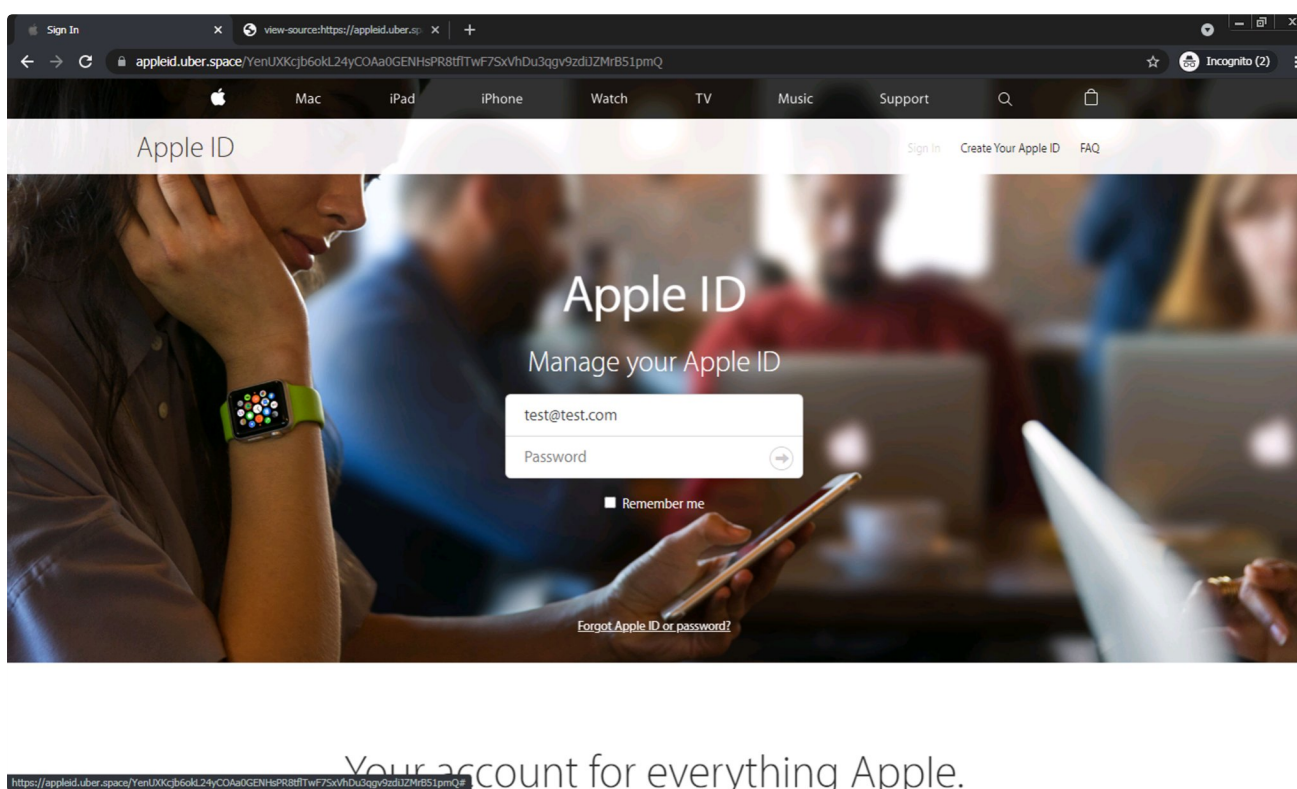
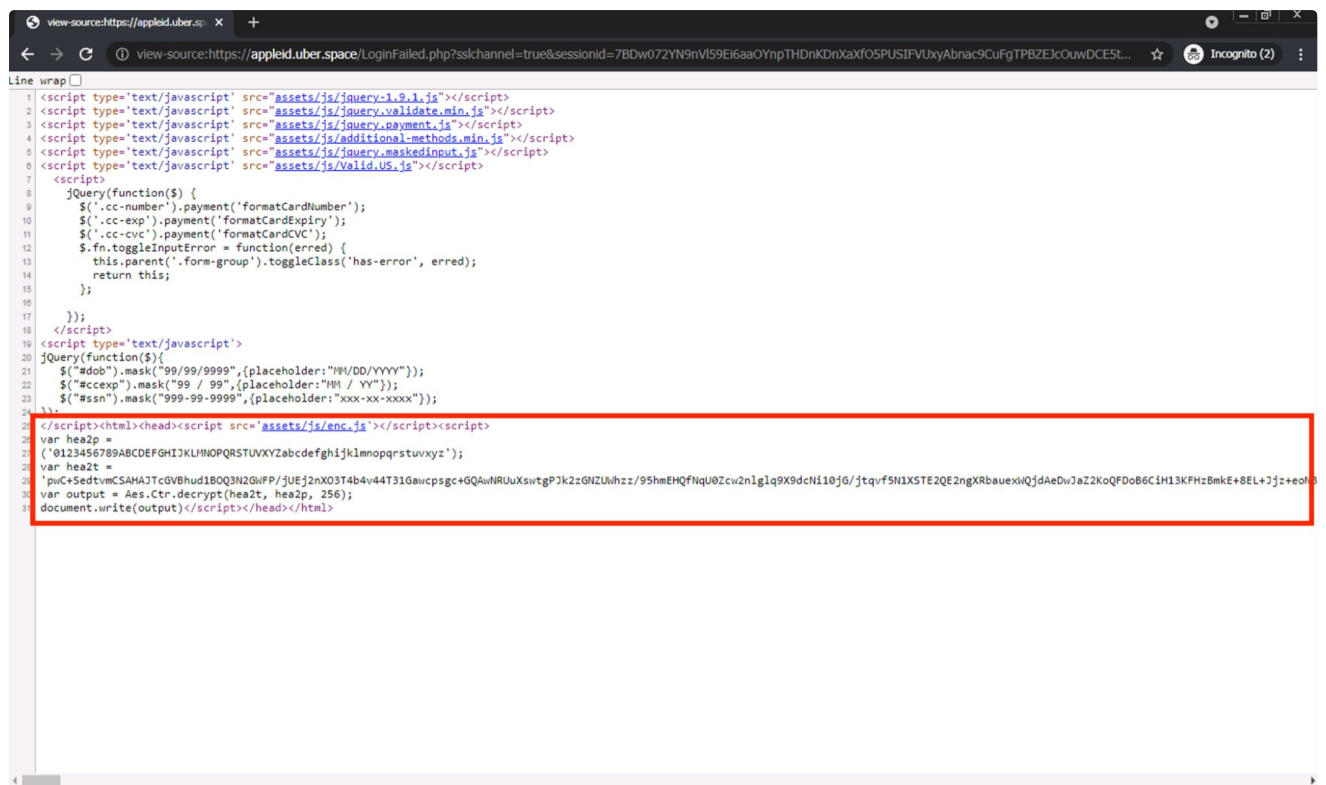


Figure 2. Same URL as Figure 1, with the credential-stealing form revealed after clicking "Confirm Refund Request."

After investigating the source code behind the page, we see that most of the page content does not exist directly within the main body of the HTML. Rather, there is a large script tag at the bottom of the HTML source that uses the `document.write(...)` API to inject the bulk of the page content into the HTML document; this happens only *after* the page is rendered in the browser.

Note that this script is also highly obfuscated, likely in an attempt to avoid phishing detection

engines. The obfuscated code runs through AES decryption before being passed into the `document.write(...)` call.



```
1 <script type='text/javascript' src='assets/js/jquery-1.9.1.js'></script>
2 <script type='text/javascript' src='assets/js/jquery.validate.min.js'></script>
3 <script type='text/javascript' src='assets/js/jquery.payment.js'></script>
4 <script type='text/javascript' src='assets/js/additional-methods.min.js'></script>
5 <script type='text/javascript' src='assets/js/jquery.maskedinput.js'></script>
6 <script type='text/javascript' src='assets/js/Valid.US.js'></script>
7 <script>
8   jQuery(function($) {
9     $('.cc-number').payment('formatCardNumber');
10    $('.cc-exp').payment('formatCardExpiry');
11    $('.cc-cvc').payment('formatCardCVC');
12    $.fn.toggleInputError = function(errored) {
13      this.parent('.form-group').toggleClass('has-error', errored);
14    };
15    return this;
16  });
17 </script>
18 <script type='text/javascript'>
19   jQuery(function($) {
20     $('#dob').mask("99/99/9999", {placeholder: "MM/DD/YYYY"});
21     $('#acexp').mask("99 / 99", {placeholder: "MM / YY"});
22     $('#ssn').mask("999-99-9999", {placeholder: "xxx-xx-xxxx"});
23   });
24 </script><html><head><script src='assets/js/enc.js'></script><script>
25   var head2 =
26     ('0123456789ABCDEFGHIJKLMNPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz');
27   var head2t =
28     'pWc+SedtmCSAHAJTcGVbhud180Q3N2GwFP/jUEj2nX03T4b4v44T31Gawcpsgc+GQ4wNRUuXsvtgp3k2zGNIZUihzz/95hmEHQfNqU0Zcv2n1g1q9X9dchi10j6/jtqv75N1XSTE2QE2ngXRbauexiuQjdAeDu7aZ2KoQFD0b6CiH13KFH:8mKE+8EL+Jjz+eol9
29   var output = Aes.Ctr.decrypt(head2t, head2, 256);
30   document.write(output)</script></head></html>
```

Figure 3. HTML source code for the page in Figures 1-2.

Sophisticated phishing pages like these may pose problems to traditional phishing detection engines. Therefore, we need to investigate additional machine learning (ML) techniques to classify pages like these as phishing.

How We Trained Our Model to Detect JavaScript-Based Phishing

We first created a training data collection pipeline to continuously gather phishing JavaScript samples from recent phishing URLs. We collected these samples from phishing URLs discovered from third-party sources and our phishing detection systems.

Once enough samples were collected, we trained a deep learning model on ~120,000 phishing and ~300,000 benign JavaScript samples. We validated the model in a staging environment before promoting it to production.

Traditional ML relies on subject matter experts to design a set of handcrafted features for the model to use as indicators of phishing or benign activity. On the other hand, our deep learning model iterates through the dataset to *learn* what patterns in the JavaScript code may indicate phishing (or otherwise suspicious) behavior. This ML-based pattern-matching makes the model more flexible than purely signature-based detection techniques, and also more robust to newer or lesser-known

JavaScript-based phishing tactics that some researchers may not have seen before.

For each script tag, the model produces a score from 0 to 1, where 1 indicates high confidence in the script being related to some sort of phishing behavior, and 0 indicates high confidence in the script being benign. For example, highly obfuscated JavaScript code, or code that injects credential-stealing forms into the page, may cause the model to output a high phishing score. Benign JavaScript code will pass through the model without raising any red flags.

Once we have a phishing score from the model, we apply various thresholds to make our final verdict regarding whether to publish the given URL as phishing or not. The PhishingJS model contributes roughly 15,000 new phishing detections weekly, meaning that Palo Alto Networks customers – specifically those who subscribe to [Advanced URL Filtering](#) – will be protected from these sophisticated phishing attacks in the real world.

PhishingJS in the Real World

Here are some sample detections that our model has generated.

First, we can see that the model can detect phishing pages employing client-side cloaking, such as when the page requires the user to click a button before actually revealing the credential-stealing form. In Figures 4-5, we see an example of a phishing webpage impersonating a Dropbox login page. The user must first click a “Sign in with Gmail” or “Sign in with Outlook” button before being presented with a modal asking for their login information. This particular URL was detected as phishing with a score of 0.99998, meaning that the model was very confident in marking this page as phishing.

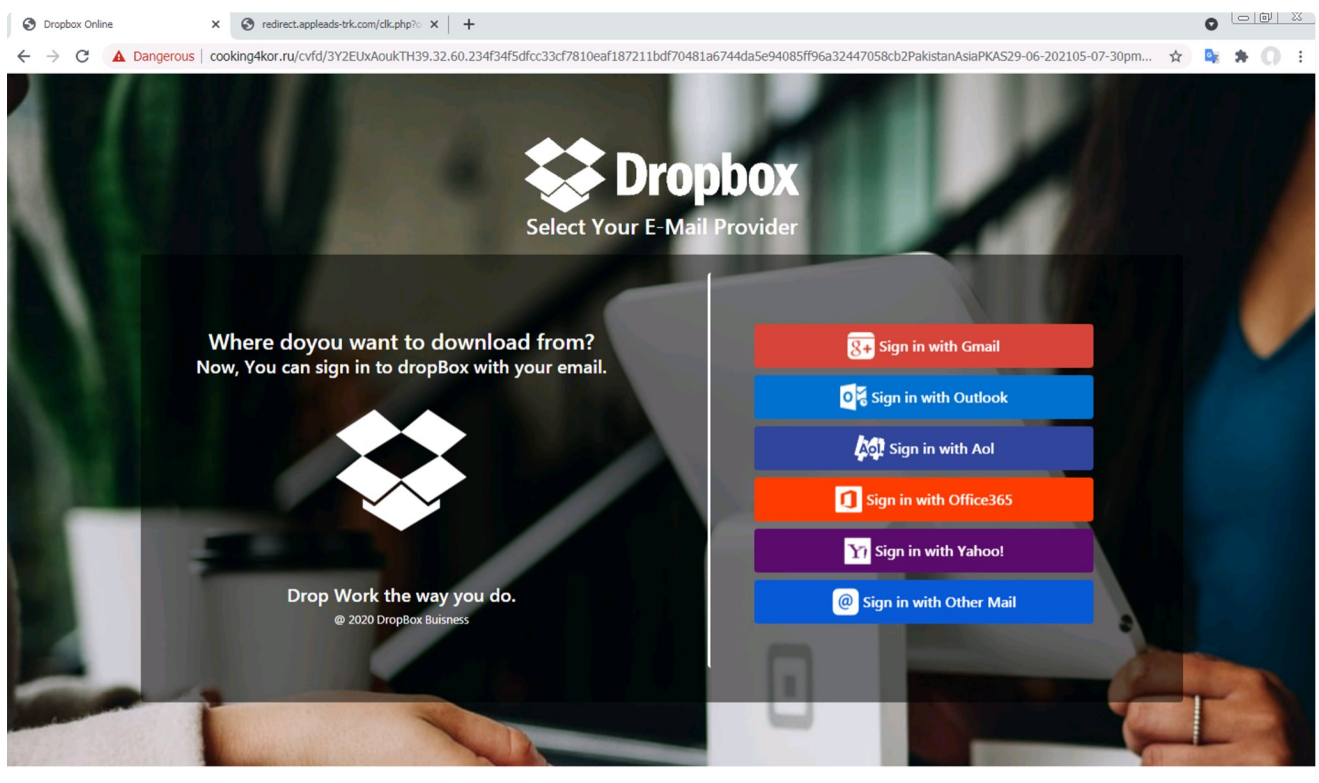


Figure 4. *cooking4kor[.]ru/cvfd/?onstoreid=40uti89763*. A phishing page employing client-side cloaking that our PhishingJS model

detected.

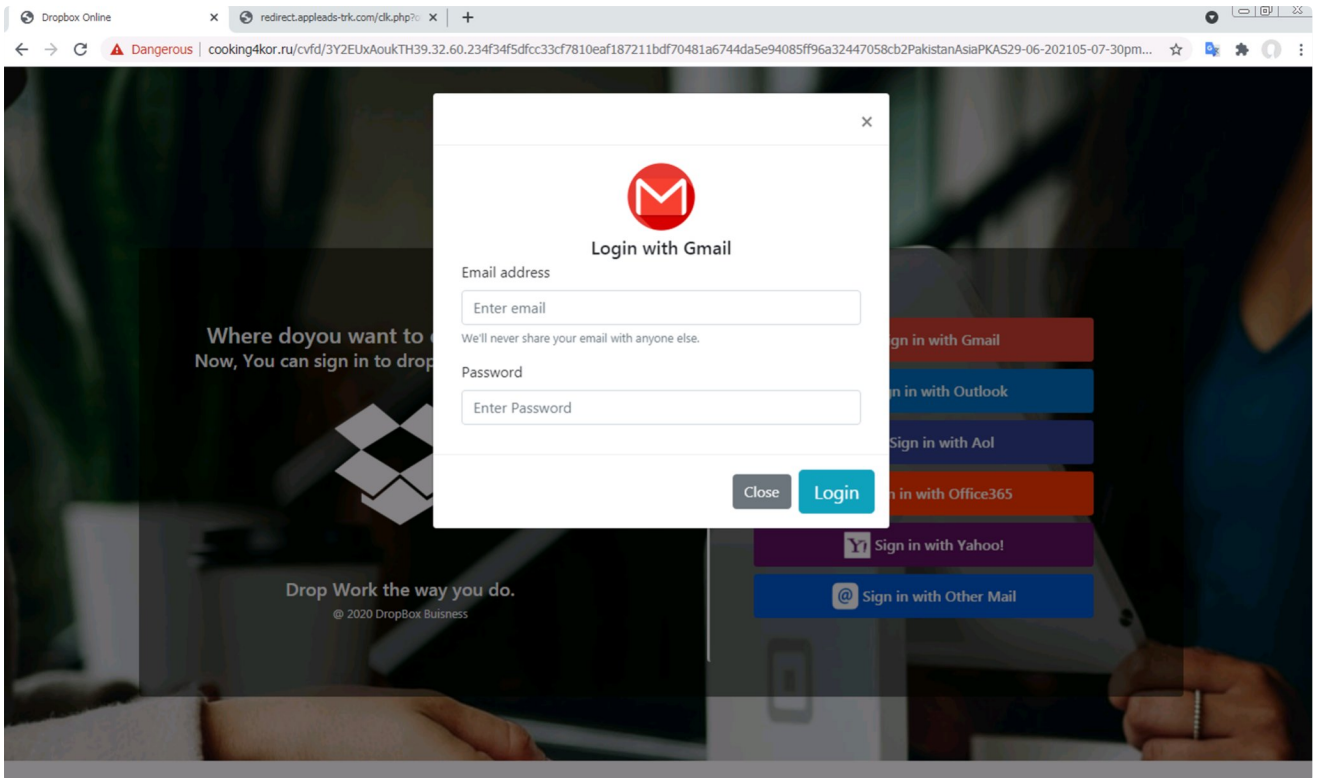


Figure 5. *cooking4kor[.]ru/cvfd/?onstoreid=40uti89763*. After user interaction.

Next, we show that the model can also detect highly obfuscated JavaScript code responsible for generating phishing webpages. In Figure 6, we present a phishing webpage that is attempting to impersonate a SharePoint login site. This may look like a relatively straightforward phishing page upon first glance. However, upon inspecting the page's source code, we see that the entire page is generated using a single JavaScript script tag. Similar to the example in Figure 3, this JavaScript snippet takes a highly obfuscated string, decodes it and then calls `document.write(...)` to inject the output into the webpage.

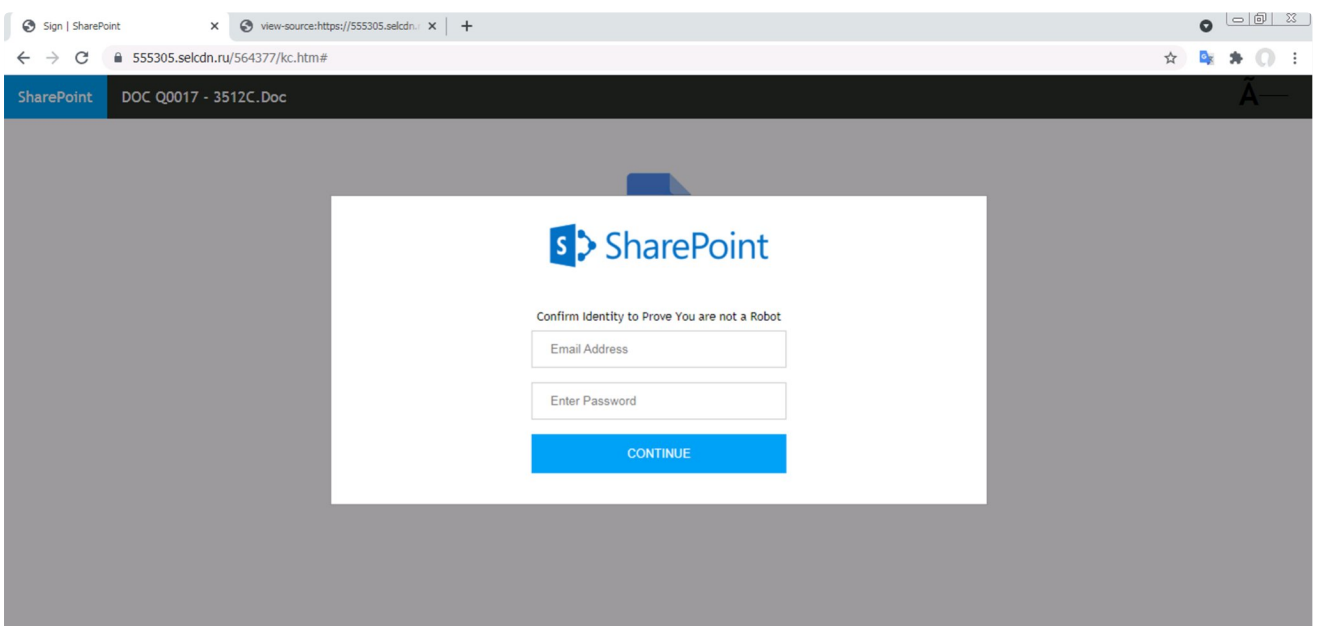


Figure 6. [https://555305\[.\]selcdn\[.\]ru/564377/kc.htm#](https://555305[.]selcdn[.]ru/564377/kc.htm#). A phishing webpage that uses JavaScript to dynamically render the content of the page.

Since the HTML of the page lacks any immediately apparent body text or input forms, this page might evade many traditional phishing detection engines. However, by analyzing the JavaScript snippet itself, our PhishingJS model detected this model as phishing with a high-confidence score of 1.00000.

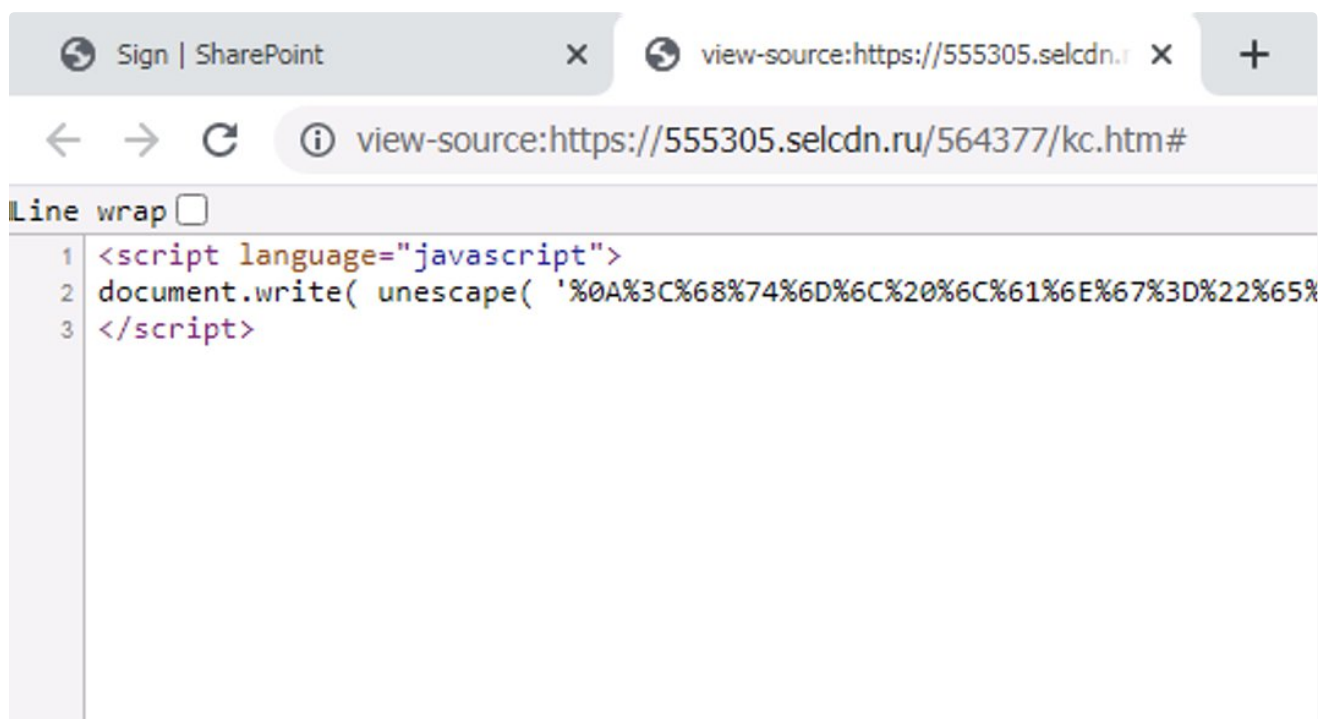
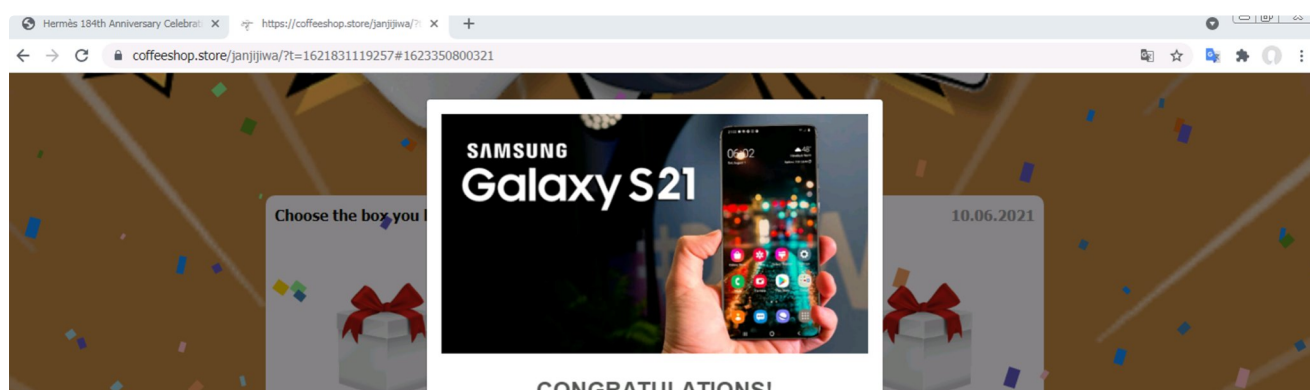


Figure 7. HTML source code for the phishing page in Figure 6.

We find that the model is capable of detecting highly interactive scam pages as well. In Figures 8-10, we show a scam page claiming that the user has won a free Samsung Galaxy S2; all the user needs to do to claim the prize is share the page with five groups or 20 friends on WhatsApp.



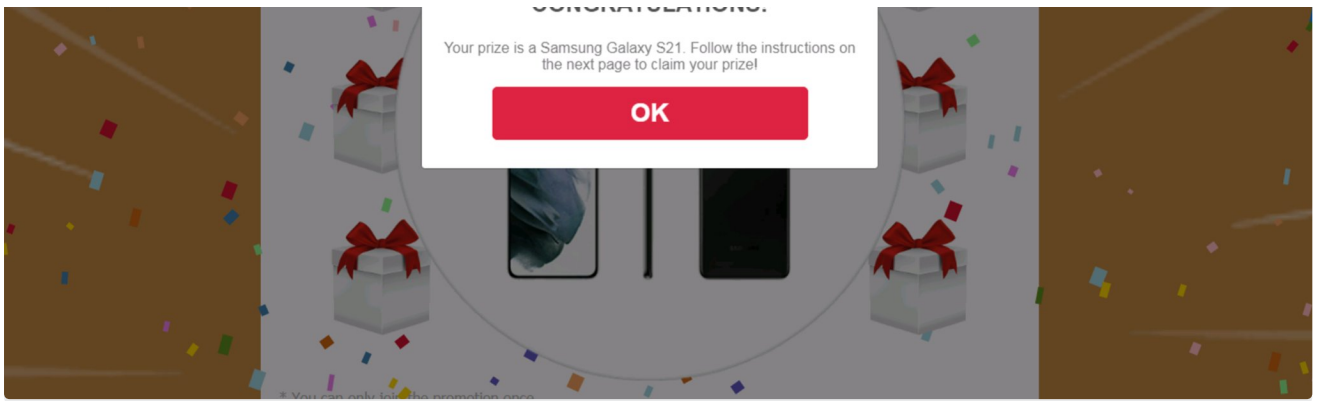


Figure 8. [http://coffeeshop\[.\]store/janjiwa?t=1621831119257](http://coffeeshop[.]store/janjiwa?t=1621831119257). A highly interactive scam page that was detected by our PhishingJS model.

Each time the user clicks the “Share” button, the page opens the user’s WhatsApp app and asks the user to share the page with another WhatsApp number. After each time the page is “shared,” the blue bar is incremented farther toward the right.

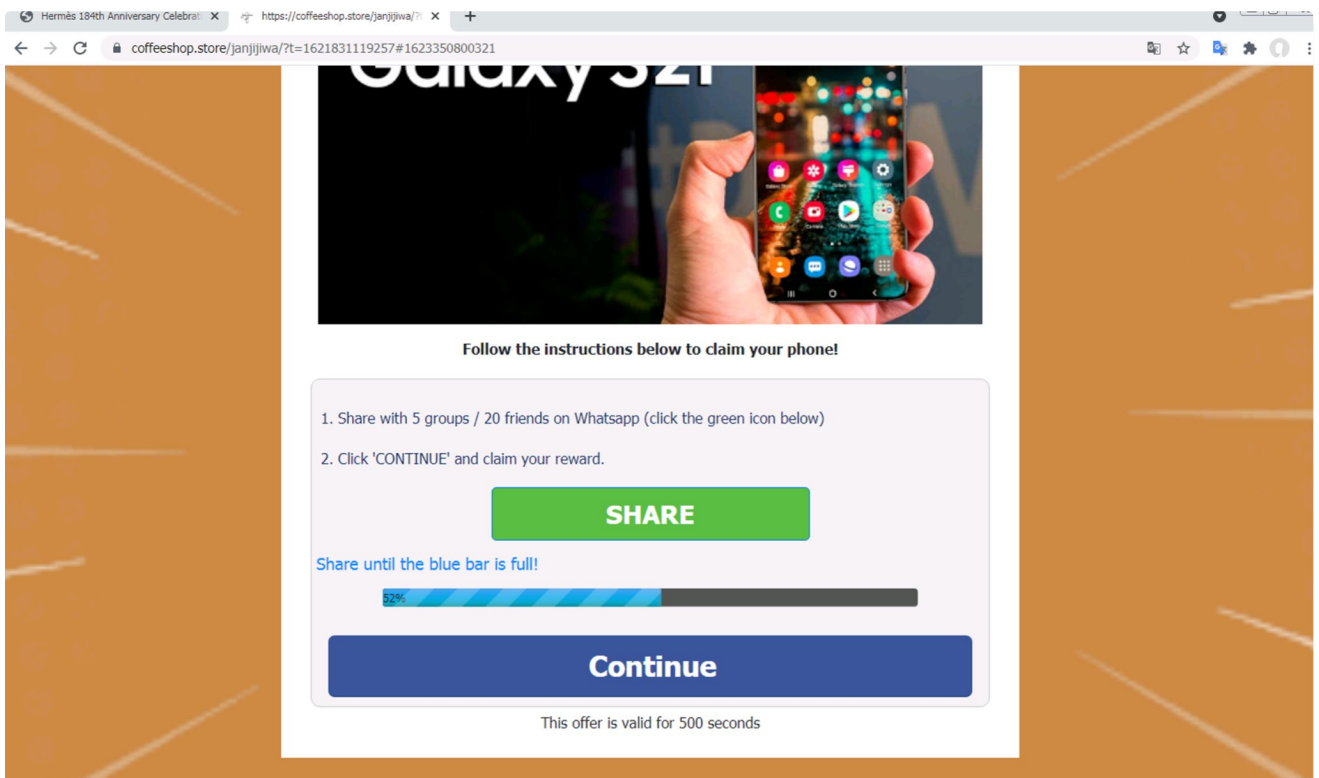


Figure 9. Same URL as Figure 8, after clicking the “Share” button.

Once the user has shared the page with the requisite number of friends, the user is prompted to complete the final registration step. Presumably, after clicking “Complete registration,” the user will be shown a form asking for some sensitive information so that the scammers can “ship the free phone” (which, to be clear, does not exist) to the user.



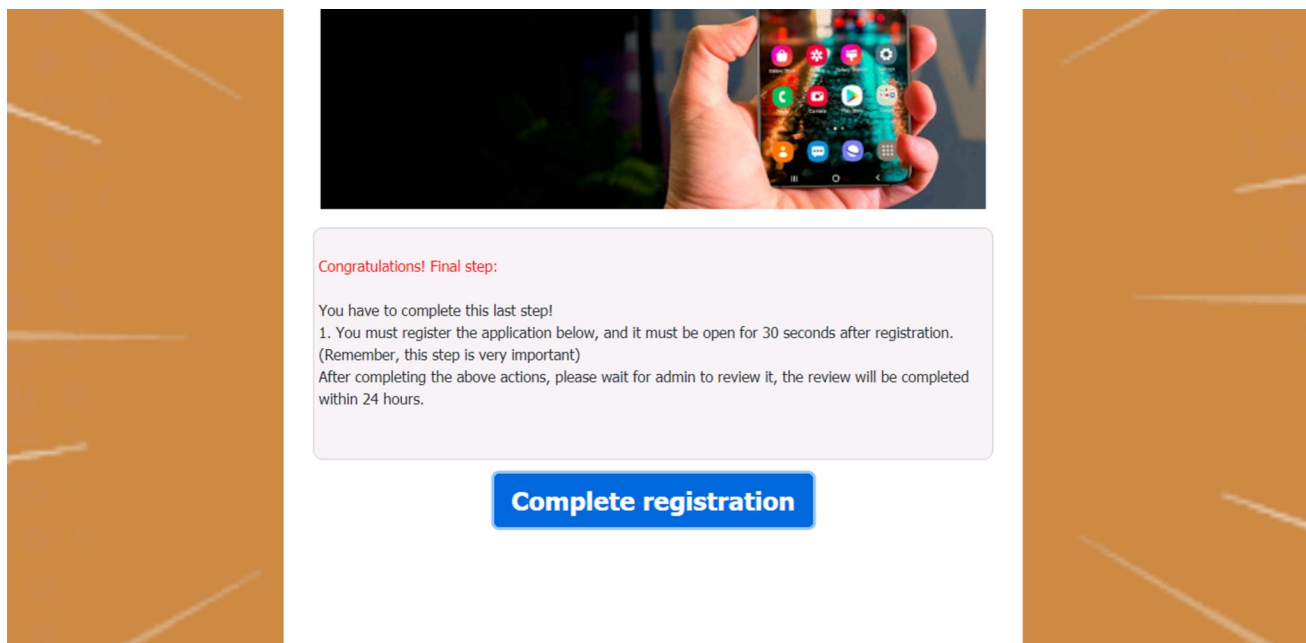


Figure 10. Same URL as Figure 8. After sharing the page with enough WhatsApp users, the user is prompted to "Complete registration."

As seen in the figures above, this scam page requires very complex user interactions before explicitly revealing any credential-stealing forms. By analyzing the JavaScript of the page and using deep learning to look for suspicious patterns that may be indicative of phishing or credential-stealing activity, we can detect these pages as phishing and prevent our customers from falling prey to these sorts of attacks in the real world.

Conclusion

Traditional phishing detection engines can often struggle to detect the increasingly sophisticated phishing webpages that cybercriminals are now crafting. Specifically, they are often unable to detect instances of client-side cloaking, wherein the phishing page may require some user interaction before revealing the actual phishing content, or wait until the page is rendered in the browser before injecting phishing content into the HTML document.

By training a deep learning model to directly analyze the JavaScript contained within the webpage, we can catch these sophisticated JavaScript-based phishing attacks and prevent them from reaching our customers.

The Palo Alto Networks ML-powered [Advanced URL Filtering](#) service for the Next-Generation Firewall, which now has this JavaScript-based phishing detection capability, can help protect against these attacks.

Indicators of Compromise

```
hxxps://appleid[.]uber[.]space  
cooking4kor[.]ru/cvfd/?onstoreid=40uti89763  
hxxps://555305[.]selcdn[.]ru
```

Additional Resources

- [The Innocent Until Proven Guilty Learning Framework Helps Overcome Benign Append Attacks](#)
- [Worldwide Phishing Attacks Ramped Up at the Peak of Working From Home](#)

Acknowledgments

The author would like to thank Wei Wang for helping to guide the PhishingJS project from start to finish; Wayne Xin and Jingwei Fan for helping to get the model into production; Brody Kutt for the original model architecture; and Seokkyung Chung, Yu Zhang, Zeyu You and Ziqi Dong for helping to review the model detections.

Get updates from Palo Alto Networks!

Sign up to receive the latest news, cyber threat intelligence and research from us

Subscribe



I'm not a robot



reCAPTCHA
Privacy - Terms

By submitting this form, you agree to our [Terms of Use](#) and acknowledge our [Privacy Statement](#).



Popular Resources

Resource Center

[Blog](#)

[Communities](#)

[Tech Docs](#)

[Unit 42](#)

[Sitemap](#)

Legal Notices

[Privacy](#)

[Terms of Use](#)

[Documents](#)

Account

[Manage Subscriptions](#)

[Report a Vulnerability](#)

© 2021 Palo Alto Networks, Inc. All rights reserved.