

2.0

Data Contract Specification for HSN 2.0 Services

Krzysztof Lasota
Paweł Pawliński

Version 0.43 Last edit by: Łukasz Siewierski, 2012-12-05

Summary
This document contains data contracts for all essential HSN 2.0 services, including feeders, web clients and analyzers. Configuration parameters, attributes read and appended to the current object and new objects are specified.

Version	Date	Author	Changes
0.1	2011-06-12	Krzysztof Lasota	Initial version
0.2	2011-06-15	Paweł Pawliński	Reformatted the whole document; partially rewrote the first two chapters (no major conceptual changes)
0.3	2011-06-17	Paweł Pawliński	Rewrote the third chapter, except the section about extensions
0.4	2011-06-20	Paweł Pawliński	Rewrote the section related to JavaScript; reorganized information about "type" attribute; extended introduction; minor fixes
0.5	2011-06-27	Paweł Pawliński	Minor improvements throughout the document; extended description of data structures used by web clients
0.6	2011-07-19	Paweł Pawliński	Changed handling of SWF and PDF files (introduction of "type" attribute)
0.7	2011-07-22	Krzysztof Lasota	New chapter: Flash services
0.8	2011-07-29	Paweł Pawliński	New service: URL normalization, updated other services accordingly
0.9	2011-08-04	Cezary Żukowski	New service: shellcode analyzer
0.10	2011-08-19	Paweł Pawliński	Minor changes and clarifications in the "standard web client" section; added appendix
0.11	2011-09-15	Paweł Pawliński	Major changes: rewrote fifth and sixth chapters (Flash and shellcode analyzers), added a new chapter (URL extraction)
0.12	2011-09-16	Paweł Pawliński	Consistent use of selected attributes in all services: depth, parent, top ancestor; web client additions: timeouts, absolute request URIs

0.13	2011-10-04	Paweł Jacewicz	Update in section about client-side redirects
0.14	2011-11-18	Paweł Pawliński	Reworked shell-scdbg specification
0.15	2011-11-24	Paweł Pawliński	New service: static n-gram JavaScript analyzer; reworked JavaScript extension for web client
0.16	2011-11-30	Paweł Pawliński	New chapter: "Common attributes"; removed automatic attributes from specifications of individual services
0.17	2012-05-16	Wojciech Michalak	New chapter: "Razorback nuggets as services"
0.18	2012-05-17	Łukasz Siewierski	New chapter: "Client honeypots". Added Thug and Cuckoo as client honeypots.
0.19	2012-05-18	Paweł Pawliński	Minor corrections to the Razorback chapter
0.20	2012-06-06	Rafał Kordalski	New attributes added by norm-url
0.21	2012-06-11	Łukasz Siewierski	New chapter: "Yara"
0.22	2012-06-11	Łukasz Siewierski	New chapter: "Analyzing PCAP files"
0.23	2012-06-26	Łukasz Siewierski	Added Cuckoo Classifier service configuration parameters
0.24	2012-06-29	Łukasz Siewierski	Added PCAP analyzing services
0.25	2012-07-04	Paweł Pawliński	Clarified link format for web client child objects
0.26	2012-07-11	Paweł Pawliński	Minor update: extended description of File structure in web client output
0.27	2012-07-17	Łukasz Siewierski	Added appendix about embedded files and links
0.28	2012-07-27	Łukasz Siewierski	Grouping embedded files into four groups: multimedia, images, objects and others
0.29	2012-07-08	Łukasz Siewierski	Embedded objects: adding td/background

0.30	2012-08-09	Paweł Pawliński	Major change of web client behavior: redirects are followed automatically now
0.31	2012-08-20	Łukasz Siewierski	Updating embedded objects appendix with new information about object/applet tags
0.32	2012-08-21	Łukasz Siewierski	Updating webclient: adding mime type and updating new objects description
0.33	2012-08-23	Łukasz Siewierski	Updating Appendix B: changed "applet/data" to "applet/code"
0.34	2012-08-30	Łukasz Siewierski	Updating Appendix B: CSS handling information
0.35	2012-09-04	Łukasz Siewierski	Changing norm-url protocol information
0.36	2012-09-19	Paweł Pawliński	Web client parameters update: limits and timeouts
0.37	2012-09-28	Paweł Pawliński	Web client parameters update: recursion limit, page timeout
0.38	2012-10-05	Łukasz Siewierski	Moving DnsQueries definition to common attributes definition
0.39	2012-10-11	Paweł Pawliński	Changed attributes appended by norm-url
0.40	2012-10-16	Paweł Pawliński	Added whitelisting to js-sta
0.41	2012-10-26	Paweł Pawliński	Web client update: minor change of parameters related to JavaScript, different handling of new objects corresponding to files
0.42	2012-12-04	Łukasz Siewierski	Added webclient timeout information and removes extract parameters
0.43	2012-12-05	Łukasz Siewierski	Old webclient timeout removed

Contents

1. Introduction **9**

2. Common attributes **11**
2.1. Attributes representing relationships 11
2.1.1. `parent` 11
2.1.2. `top_ancestor` 11
2.1.3. `depth` 11
2.2. Attributes added automatically 12
2.2.1. `creation_time` 12
2.2.2. `top_ancestor` 12
2.2.3. `depth` 12

3. Common data structures **13**
3.1. `JSContextList` 13
3.2. `DnsQueries` 13

4. Normalization services **14**
4.1. URL normalization service (norm-url) 14
4.1.1. Service configuration 14
4.1.2. Input data 14
4.1.3. Appended attributes 14

5. Feeder services **16**
5.1. URL list feeder [SF-1] (feeder-list) 16
5.1.1. Service configuration 16
5.1.2. Input data 17
5.1.3. Appended attributes 17
5.1.4. New objects 17

6. Web client services **19**
6.1. Standard web client (webclient) 19
6.1.1. Service configuration 19
6.1.2. Input data 22
6.1.3. Attributes added for visited pages 23

6.1.4. New objects 27
6.2. Dumping JavaScript sources 28

7. Flash services **30**
7.1. Flash CVE analyzer (swf-cve) 30
7.1.1. Service configuration 30
7.1.2. Input data 30
7.1.3. Appended attributes 30
7.2. Flash shellcode scanner (swf-scs) 31
7.2.1. Service configuration 31
7.2.2. Input data 31
7.2.3. Appended attributes 31
7.2.4. New objects 32

8. Shellcode analyzers **33**
8.1. Libscizzle-based shellcode analyzer (shell-scizzle) 33
8.1.1. Service configuration 33
8.1.2. Input data 33
8.1.3. Appended Attributes 34
8.2. Scdbg-based shellcode analyzer (shell-scdbg) 34
8.2.1. Service configuration 35
8.2.2. Input data 35
8.2.3. Appended Attributes 35
8.2.4. New objects 36

9. URL extraction services **37**
9.1. Generic URL scanner (url-scan) 37
9.1.1. Service configuration 37
9.1.2. Input data 37
9.1.3. Appended attributes 38
9.1.4. New objects 38

10. JavaScript analyzers **40**
10.1. Static n-gram JavaScript analyzer (js-sta) 40
10.1.1. Service configuration 40
10.1.2. Input data 40
10.1.3. Appended attributes 41

11. Razorback nuggets as services **43**
11.1. Information regarding all nuggets 43

11.2. Flash analyzer nugget - swfScanner (rb-swfscanner) 43
11.2.1. Service configuration 43
11.2.2. Input data 43
11.2.3. Appended attributes 44
11.3. PDF analyzer nugget - pdfFox (rb-pdffox) 44
11.3.1. Service configuration 44
11.3.2. Input data 45
11.3.3. Appended attributes 45
11.4. Microsoft Office analyzer nugget - officeCat (rb-officecat) 45
11.4.1. Service configuration 45
11.4.2. Input data 46
11.4.3. Appended attributes 46
11.5. Clam AntiVirus nugget - clamavNugget (rb-clamavnugget) . . 46
11.5.1. Service configuration 46
11.5.2. Input data 47
11.5.3. Appended attributes 47
11.6. VirusTotal nugget - virusTotal (rb-virustotal) 47
11.6.1. Service configuration 47
11.6.2. Input data 48
11.6.3. Appended attributes 48
11.7. Archive Inflation nugget - archiveInflate (rb-archiveinflate) . . . 48
11.7.1. Service configuration 48
11.7.2. Input data 49
11.7.3. Appended attributes 49
11.7.4. New objects 49

12. Client honeypots **51**
12.1. Thug (thug) 51
12.1.1. Service configuration 51
12.1.2. Input data 52
12.1.3. Appended Attributes 52
12.1.4. New objects 54
12.2. Cuckoo (cuckoo) 54
12.2.1. Service configuration 54
12.2.2. Input data 56
12.2.3. Appended Attributes 57
12.2.4. New objects 58
12.2.5. Exceptions 58

13. Yara **59**

13.1. Introduction 59
13.2. Performance 59
13.3. Rule sources 59
13.4. Yara analyzer (yara) 59
13.4.1. Service configuration 60
13.4.2. Input data 60
13.4.3. Appended Attributes 60
13.4.4. New objects 61

14. Analyzing PCAP files **62**
14.1. PCAP file extractor (pcap-extract) 62
14.1.1. Service configuration 63
14.1.2. Input data 63
14.1.3. Appended Attributes 63
14.1.4. New objects 63
14.2. PCAP analyzer (pcap-analyze) 63
14.2.1. Looking for unexpected traffic 63
14.2.2. Looking for traffic with malicious domains / IPs 65
14.2.3. Extracting known protocols 65
14.2.4. Service configuration 66
14.2.5. Input data 66
14.2.6. Appended Attributes 67
14.2.7. New objects 67
14.3. Checking for malicious domains (malicious-domains) 67
14.3.1. Service configuration 68
14.3.2. Input data 69
14.3.3. Appended Attributes 69
14.3.4. New objects 69

A. Redirects in web pages **70**
A.1. Server-side redirects 70
A.2. Client-side redirects 70

B. Embedded resources and links **72**
B.1. Embedded resources 72
B.2. URL child objects 73
B.3. Not present in `webclient` output 74

1. Introduction

This document contains specification of feeder and web client services, which are necessary for basic functionality of HSN 2.0. The specification is focused on the attributes which are appended by services to the currently processed or newly created objects.

Attribute types are selected in accordance with the following rules:

- For attributes that may contain large amount of data (e.g. long text, document, image) we use the *bytes* type. Contents are saved in a data store and only a reference is kept in the attribute itself.
- We also use the *bytes* type for attributes that hold structured data, especially lists. However in such cases, a data store entry holds a serialized protocol buffers message. This document contains definition of all protocol buffers structures that are referred from attributes.
- In all other cases we use 'flat' attribute types, that store data directly within an object, e.g. *boolean*, *integer*, *string*.

A list of all types with information how they are represented in protocol buffers is contained in the HSN 2.0 Communication Protocol [3].

Definitions of some terms used in this document:

current object The current object for the instance of the workflow process that invoked the service; input of a service. The current object consists of an identifier (reference) within an object store and a set of attributes.

append an attribute For services, the most common way to return results is to append a new attribute to the current object. Depending on the workflow, this attribute can be read and interpreted by other components.

create an object An alternative way of returning results — a service can create a new object that, depending on the workflow, can trigger a new instance of a process in the workflow. After triggering, the newly created object becomes the current object of the new process.

p(*param***)** Reference to the value of the configuration parameter *param*.

a(*attr***)** Reference to the value of the attribute *attr* of the current object.

a^n**(***attr***)** Reference to the value of the attribute *attr* of a new object.

2. Common attributes

This chapter describes attributes that have special semantics in the whole HSN 2.0 system.

2.1. Attributes representing relationships

Attributes listed in this section implement various relationships that can occur between different objects in the system.

2.1.1. `parent`

The most important relationship in the HSN 2.0 data model is the parent-child relationship. When a service creates new objects based on information present in its current object, all these new objects are considered children of the current object.

Generally, every child has a `parent` attribute that points to its parent. However, this rule does not hold for situations where contents of the parent object does not have any relation to contents of its children, e.g. URLs generated by a feeder do not have a(`parent`) reference.

Type: object

2.1.2. `top_ancestor`

Top ancestor is an object without a parent. Usually such objects correspond to initial URLs that were used to start a crawl. Every object has this attribute pointing to the initial object in the current crawl tree. Top ancestors also have this property but in their case it is a self-reference.

Type: object

2.1.3. `depth`

Depth of the object in relation to a(`top_ancestor`). Objects which are top ancestor themselves have depth equal to zero. Usually value of `depth` corresponds to the number of link and redirects followed in a particular branch of the crawl process.

Type: integer

2.2. Attributes added automatically

This section lists all attributes that are appended automatically to new objects by the object store and defines how their values are computed. Most of these attributes are described in more detail in the previous section. They should never be explicitly appended to objects by services or other components.

2.2.1. `creation_time`

Time when the object was created. Format: seconds in UNIX time (UTC+0 timezone).

Type: timestamp

2.2.2. `top_ancestor`

If the object being added to the store has a(`parent`) defined, attribute `top_ancestor` from parent is copied here. Otherwise, this reference is set to identifier of the object being added (self-reference).

Type: object

2.2.3. `depth`

If the object being added to the store has a(`parent`) defined, take value of a(`depth`) from the parent object and increment it by one. Otherwise, assign zero.

Type: integer

3. Common data structures

3.1. JSContextList

Structure `JSContextList` contains list of all contexts encountered on a website.

```
message JSContextList {
  repeated JSContext contexts = 1;
}
```

Each context consists of plain, unparsed JavaScript source code and a flag that signals whether it was executed through an `eval` call, or it was present directly in the web page. Additionally, each context has an identifier, which is unique for the given object — this method allows to refer to individual contexts without relying on list ordering.

```
message JSContext {
  required uint32 id = 1;
  required string source = 2;
  required bool eval = 3;
}
```

3.2. DnsQueries

Structure `DnsQueries` contains list of DNS queries performed.

```
message DnsQueries {
  repeated string dns = 1;
}
```

4. Normalization services

This chapter contains specification of services dedicated to normalization of data to various canonical forms. The most important reason for using separate services for this purpose is to guarantee consistent data formats across the whole system — there is no risk that a subtle difference in implementation of normalization logic in a single component will lead to unexpected behavior.

4.1. URL normalization service (norm-url)

The only purpose of this service is to normalize URLs to a single canonical form. Since URLs can be represented in multiple ways, or even deliberately obfuscated, it only makes sense to compare their normalized forms.

4.1.1. Service configuration

This service does not have any configuration parameters.

4.1.2. Input data

Only a single attribute is read by the service:

`url_original` mandatory: yes type: string
The URL in its original form, e.g. as appeared in the HTML source.

4.1.3. Appended attributes

The service transforms the `url_original` attribute and adds the following one:

`url_normalized` type: string condition: always
URL in its canonical form. Normalization requires the use of punny-code for IDN domains and percent-encoding for whole URL. Normalized URL must start with a proper URI scheme (e.g. `http://`) and comply with its format. Further details are defined in a separate document [5].

`protocol` type: string condition: always
URL scheme name, e.g. `http`, `mailto`, `about`.

`host` type: string condition: if present
IP address or FQDN in the authority part of the URL.

`port` type: integer condition: if present
Port number in the authority part of the URL.

`path` type: string condition: always
URL path, without query part. If the authority part is present in the URL, an empty path must be normalized to a single slash (`/`), otherwise it is left unchanged.

`tld` type: string condition: if present
Top-level domain in the authority part of the URL.

`sld` type: string condition: if present
Second-level domain in the authority part of the URL.

`query` type: string condition: if present
Query part of the URL (without the fragment component).

5. Feeder services

This chapter contains a complete feeder data contract, based on the Feeder List document [1].

5.1. URL list feeder [SF-1] (feeder-list)

This feeder provides functionality to access a local or remote text file containing a list of URLs that should be processed. The text file must contain single URL per line (either `LF` or `CR LF` terminated). For each URL the feeder creates a new object.

5.1.1. Service configuration

We can distinguish three groups of feeder configuration parameters. The first one contains parameters responsible for the proper operation of the service.

`uri` mandatory: yes type: string default value: N/A
URI to the file containing the URL list. The feeder should provide the functionality to process file, HTTP, HTTPS, and FTP URI schemes. E.g. `file:///var/local/urls`

The second group of parameters specifies if any additional information is appended to the current object.

`statistics` mandatory: no type: boolean default value: true
If set to true, the service will append statistical information about processed URL list.

The third group of parameters specifies what data is appended to the newly created objects:

`domain_info` mandatory: no type: boolean default value: true
If set to true, the service will append additional information about URL domain, if present.

5.1.2. Input data

Feeders' actions are controlled entirely by their configuration parameters, they do not read any attributes of the current object.

5.1.3. Appended attributes

Two kinds of appended attributes can be distinguished for the feeder — ones that are added unconditionally and ones that are optional (depend on the configuration parameters).

`feeder_time_begin` type: timestamp condition: always
Time when the service started to process its current task. Format: seconds in UNIX time.

`feeder_time_end` type: timestamp condition: always
Time when the service finished processing its current task. Format: seconds in UNIX time.

`url_count` type: integer condition: always
Number of URLs read from the text file.

`domain_count` type: integer condition: p(`statistic`)
Number of URLs read from the file that use domain names, including IDN (internationalized domain name).

`domain_idn_count` type: integer condition: p(`statistic`)
Number of URLs read from the file that use IDNs.

`ipv4_count` type: integer condition: p(`statistic`)
Number of URLs read from the file that use IPv4.

`ipv6_count` type: integer condition: p(`statistic`)
Number of URLs read from the file that use IPv6.

5.1.4. New objects

Presence of some attributes appended to newly created objects depend on configuration parameters. Others (e.g. `url_original`) are added unconditionally.

`type` type: string condition: always
Type of the new object. All objects created by feeders are URLs, hence they always set this attribute to `url`.

`url_original` type: string condition: always
Original URL string, as read from the text file.

`origin` type: string condition: always
Name of a service that created the object. For the URL list feeder the value is always set to the character string `file`.

`origin` needs to be described in more detail.

`url_domain` type: string condition: p(`domain_info`) ∧ *URL contains a domain name*
Canonical form of the domain name in ASCII. Normalization requires the use of punny-code for IDN domains.

`url_ip` type: string condition: p(`domain_info`) ∧ *URL contains an IP address*
IP address in ASCII form. IPv4 addresses are written in the dot-decimal notation. IPv6 addresses are written as 8 groups of 16-bit hexadecimal values separated by colons, without any abbreviations.

6. Web client services

This chapter contains web client data contract, based on HSN 2.0 Crawler: Design Notes [2] and HSN 2.0 Initial Data Contract Specification [4].

6.1. Standard web client (webclient)

The HSN 2.0 web client provide functionality to visit a requested URL and download contents of a page (e.g. HTML, JavaScript, images). From the point of view of the server it emulates behavior of a normal web browser.

Downloaded data is stored in attributes of the current object. Standard web client creates new objects for external links in the web page and for redirections such as (a) server-side redirections for the initial URL (b) client-side redirection triggered by the web page content (c) frames (d) iframes. New objects corresponding to links are not automatically processed by the same instance of the service — depending on the workflow, they can be passed to a web client (crawling) or abandoned. Redirections, on the contrary, are followed automatically in the same manner as normal web browsers visit these sub-pages. Content downloaded from sub-pages is saved in attributes of their objects, so in the result a whole complex page is represented as a tree of objects corresponding to individual sub-pages (main page and frames or other types of redirections). Cookies and referrer information is shared among current URL and new (descendant) objects in order to keep a valid context.

6.1.1. Service configuration

Since the web client is responsible for collecting data from examined pages, its behavior is highly configurable. It is possible to specify details of interaction with remote servers and interpretation of the downloaded content. All parameters of the web client are optional, they must be set only if one needs to customize behavior of the service.

Parameters with names starting with `save` specify what kind of data is downloaded from the page. Every file downloaded by the web client is always reported as a new object, regardless of attributes in objects representing pages (see section 6.1.4). Parameters starting with `add` specify what kind of context information is saved in attributes of the descendant objects.

Since the web client is capable of following redirects automatically there exist parameters that limit such behavior. If some redirects could not be followed due to these limits, an appropriate warning should be generated.

The last group of parameters controls execution of JavaScript present in the visited web pages.

`link_click_policy` mandatory: no type: integer default value: 1
Controls how external links (e.g. `<a href=...>`) should be processed. Possible values:

- **0** no new child[1] objects are created
- **1** new objects are created for links in FIFO order; number of objects is limited by the `link_limit` parameter
- **≥2** *reserved for future versions:* new objects are created for links in some other order — this parameter will control which heuristic will be used to prioritize and select interesting links (e.g. depending on the target domain).

`link_limit` mandatory: no type: integer default value: 100
Maximum number of new objects originating from external links.

`redirect_total_limit` mandatory: no type: integer default value: 50
Maximum number of redirects to follow in total (see description of redirects on page 19).

`redirect_depth_limit` mandatory: no type: integer default value: 10
Maximum depth of redirect chains (object tree height).

`save_html` mandatory: no type: boolean default value: true
If set to true, the HTML content received after the initial GET request will be saved in the `html_source` attribute.

`save_images` mandatory: no type: boolean default value: false
If set to true, contents of all images embedded in a web page (see appendix B) will be put as separate entries in a data store and their metadata with references to the content will be saved in a structured format in the `image_list` attribute. Some elements in the `image_list` may overlap with other, more specific lists specified previously, however file contents will be stored only once.

[1] *Child objects* are direct descendants of some other object.

`save_objects` mandatory: no type: boolean default value: false
If set to true, contents of all objects embedded in a web page (see appendix B) will be put as separate entries in a data store and their metadata with references to the content will be saved in a structured format in the `object_list` attribute. Also, all downloaded PDF and SWF files will have a new object of type `file` created for them. Some elements in the `object_list` may overlap with other, more specific lists specified previously, however file contents will be stored only once.

`save_multimedia` mandatory: no type: boolean default value: false
If set to true, all multimedia content embedded in a web page (see appendix B) will be put as separate entries in a data store and their metadata with references to the content will be saved in a structured format in the `multimedia_list` attribute. Some elements in the `multimedia_list` may overlap with other, more specific lists specified previously, however file contents will be stored only once.

`save_others` mandatory: no type: boolean default value: false
If set to true, all content not described above, but embedded in a web page (see appendix B) will be put as separate entries in a data store and their metadata with references to the content will be saved in a structured format in the `other_list` attribute. Some elements in the `other_list` may overlap with other, more specific lists specified previously, however file contents will be stored only once.

`save_failed` mandatory: no type: boolean default value: true
If set to true, all failed requests (e.g. to nonexistent images) are saved in a structured format in the `failed_list` attribute.

`save_cookies` mandatory: no type: boolean default value: true
Controls whether cookies set by the server are saved in a structured format in the `cookie_list` attribute of the current object.

`add_referrer` mandatory: no type: boolean default value: true
Adds the referrer's URL to child objects in the `referrer` attribute.

`add_referrer_cookie` mandatory: no type: boolean default value: true
Adds the `referrer_cookies` attribute to child object with a reference to cookies from the parent page.

`single_js_timeout` mandatory: no type: integer default value: 3000
The amount of time in milliseconds that a single script can execute. When it elapses, the execution of the given script is aborted.

`js_enable` mandatory: no type: boolean default value: true
Should JavaScript on the page be executed by the web client.

`background_js_timeout` mandatory: no type: integer default value: 10000
The maximum amount of time (in milliseconds) that the web client can wait for all scripts scheduled for execution in background. When it elapses no further scripts will not be executed and the currently processed ones are aborted. This time is measured from the moment the given page is loaded by the client, i.e. all embedded elements have been downloaded.

`page_timeout` mandatory: no type: integer default value: 30000
The maximum amount of time (in milliseconds) that the web client can wait for a resource to be downloaded. This is per resource timeout.

`processing_timeout` mandatory: no type: integer default value: 90000
The maximum amount of time (in milliseconds) that the web client can wait for all processing. This is a hard limit, after this time all processing is stopped, none of the information is saved and `TaskError` is send.

`js_recursion_limit` mandatory: no type: integer default value: 80
In order to limit the amount of memory consumed by the JavaScript engine, recursive execution of identical script contexts (see section 6.2) is detected at runtime. If the same context is executed more than p(`js_recursion`) times, the current script is aborted. A number equal or less than zero implies that the limit is disabled.

6.1.2. Input data

In order to visit a web page, the web client requires that the current object carries information about the target URL. Additional context data will be used

if present.

`url_original` — mandatory: if a(`url_normalized`) missing — type: string
The URL in its original form. If present, it will be used for requests to the remote server instead of the normalized form, in order to avoid detection.

`url_normalized` — mandatory: if a(`url_original`) missing — type: string
A canonical form of the URL to visit. It will be used for the initial HTTP request only if `url_original` attribute is not supplied.

`referrer` — mandatory: no — type: string
Referrer string that will be sent to the remote server.

`referrer_cookie` — mandatory: no — type: bytes
Cookies to be loaded by the service and sent to the remote server. Referrer's cookies can be overwritten by the web page.

6.1.3. Attributes added for visited pages

The service creates new objects (with attributes defined in section 6.1.4) for any web page that was visited in the result of an automatic redirection. Attributes specified in this section are appended to these objects and the current object, which corresponds to the main page.

Standard web client always appends several attributes related to the request that was sent to the remote server. Saving of downloaded content however, is entirely optional and depends on the configuration.

Attributes described in this section use some common data structures, which are presented below:

```
1 import "Object.proto"; // contains definition of Reference
2 message File {
3   repeated Request requests  = 1;
4   required string type       = 2;
5   required Reference content = 3;
6   optional uint64 object     = 4;
7 }
```

```
1 message FileList {
2   repeated File files = 1;
3 }
```

When the web client attempts to download a remote resource, its initial request may be redirected to another URL. A complete chain of requests is saved in the `requests` field of the `File` structure. If the resource was finally fetched, it is saved in the data store (`content` field) and its type in the MIME format is saved in the `type` field. The type of the downloaded file is determined by web client's internal mechanisms, in particular it is not rewritten from the HTTP header, based on the extension of the filename, or deduced from the origin HTML element. If the service created a new object corresponding to the downloaded file, it should put a reference to it in the `object` field. `File` structures can be grouped together in a `FileList`.

```
1 message Request {
2   required string request_url_original = 1;
3   required string request_url_absolute = 2;
4   optional string request_header       = 3;
5   optional uint32 response_code        = 4;
6   optional string response_header      = 5;
7 }
```

The `Request` structure holds parsed HTTP request message (without body) and, if the server responded, an HTTP response message (also without body). Attribute `request_url_original` represents the request URI in its original form (without normalization, can be relative) and the `response_code` is the HTTP status code any was received. Request URI in the absolute format (with protocol and hostname specification) is stored in the `request_url_absolute` but, similarly to `request_url_original`, it **is not** fully normalized (see [5] for the definition of fully normalized URIs). `request_header` and `response_header` hold full HTTP headers[2] concatenated to a single string, including newline characters.

`active` — type: boolean — condition: always
True if the web client successfully received content for the initial URL. Generally, any response that contains HTTP message body [6] should be treated as success, regardless of the HTTP status. Value of `active` determines which other attributes will be appended.

`http_request` — type: bytes — condition: always
`Request` structure describing the request for the initial URL (from the `url_normalized` or `url_original` attribute). Since redirects are not followed by the web

[2] HTTP header does not include start line, e.g. `HTTP/1.1...`

client, a single structure is sufficient to hold all information. This data could be provided in several flat attributes (in accordance with the guidelines presented in the introduction chapter on page 9) but it is kept as a `Request` structure for the sake of consistency with other attributes.

`http_code` type: integer condition: *remote server sent a reply*
HTTP response status code. This information can be retrieved from the `request` attribute but is provided here for convenience.

`reason_failed` type: string condition: ¬ a(`active`)
Human-readable error message explaining why the initial request failed to retrieve any data.

`html` type: boolean condition: a(`active`)
True if the downloaded content was a web page and was interpreted as such by the web client. It may happen that the initial URL was not a web page — in such cases value of this attribute is set to false.

`html_source` type: bytes condition: p(`save_html`) ∧ a(`html`)
Original HTML source as sent by the server — a complete HTTP message body, with no processing performed by the client.

`image_list` type: bytes condition: p(`save_images`) ∧ a(`html`)
List of image resources embedded directly in the web page (see appendix B), saved in a `FileList` structure.

`object_list` type: bytes condition: p(`save_objects`) ∧ a(`html`)
List of object resources embedded directly in the web page (see appendix B), saved in a `FileList` structure.

`multimedia_list` type: bytes condition: p(`save_multimedia`) ∧ a(`html`)
List of multimedia resources embedded directly in the web page (see appendix B), saved in a `FileList` structure.

`other_list` type: bytes condition: p(`save_others`) ∧ a(`html`)
List of other resources embedded directly in the web page (see appendix B), saved in a `FileList` structure.

`failed_list` type: bytes condition: p(`save_failed`) ∧ a(`active`)
List of all attempts to access resources embedded directly in the web page (see attribute `file_list`) that failed for any reason. Together, all `*_list` attributes combined contain a complete set of all HTTP requests performed by the web client when visiting the web page.

```
1 message FailedList {
2   repeated FailedRequest failed = 1;
3 }
```

Every entry in the list contain a sequence of requests that lead to the failure, along with a general error description in a human readable form.

```
1 message FailedRequest {
2   repeated Request requests = 1;
3   required string reason = 2;
4 }
```

`cookie_list` type: bytes condition: p(`save_cookie`) ∧ a(`active`)
List of all cookies that were set after visiting the web page. This includes cookies that are set when the web client performs additional requests for content embedded in the web page. If the `referrer_cookie` attribute is set, parent's cookies are also kept in `cookie_list`. It is expected that a deep crawl can cause accumulation of a significant number of entries.

Alternative: we could store unparsed cookies or use the standard cookie file format

```
1 message CookieList {
2   repeated Cookie cookies = 1;
3 }
```

Cookies are stored as key-value pairs, in ASCII format. All attributes (e.g. `Domain`) are also parsed and stored in a separate list.

```
1 message Cookie {
2   message Attribute {
3     required string name = 1;
4     optional string value = 2;
5   }
6   required string name = 1;
7   required string value = 2;
```

```
8     repeated Attribute attributes = 3;
9  }
```

6.1.4. New objects

There are two categories of descendant objects that can be created by the web client — ones that correspond to whole web pages (`type` set to `url`) or objects that represent individual files available for further processing (other values of the `type` attribute).

`url` descendant objects correspond to both redirects (followed) and external links (**not** followed automatically). Depending on the configuration parameters, context information can be saved in attributes of descendant objects. This information allows the web client to emulate a single browsing session, when in fact the processing might be performed by completely different instances of the service.

Objects representing individual files have their `type` set to `file` and are created in situations when a standard browser would normally automatically download a file or open a dialog window prompting the user to save a file received from the remote server. These objects contain filetype-specific information that is interpreted by relevant analyzers. Additionally, all files embedded directly in a web page (see parameters prefixed with `save` in section 6.1.1 for more information on types of embedded files) should also be reported in the same way.

`type` — type: string — condition: always
Type of the new object created by the web client. This attribute can have one of the following values:

`url` for URL descendant objects
`file` for objects corresponding to files

`mime_type` — type: string — condition: $a^n(\texttt{type}) = \texttt{file}$
MIME type of the file, as presented by the website, but converted to lower case characters.

`url_original` — type: string — condition: $a^n(\texttt{type}) = \texttt{url}$
Target URL as appeared in the web page but converted to an absolute form. Escape sequences or other types of obfuscation are not removed.

`origin` — type: string — condition: always
A short string identifying what was the reason for creat-

`origin` needs to be described in more detail.

ing the object by the web client. For URLs possible values correspond to different types of external references and redirections: (a) `redirect` for server and client-side redirections to HTML documents, (b) `frame`, (c) `iframe`, or (d) `link`. For objects corresponding to files, the value is set to `direct` for files that are downloaded automatically by the browser in a context where HTML could be served instead, e.g. `src` in `iframe` pointing to a PDF document or a plain link. It is assumed that all other automatically downloaded files are of `embedded` origin.

`parent` — type: object — condition: always
Reference to the object representing the page that is the parent of the new object. Since the service automatically follows redirections, value of the attribute may not point to the *current* object but some other newly created one. `parent` is not necessarily a web page — can be a server-side redirection.

`referrer` — type: string — condition: p(`save_referrer`)
Value of the `referer` HTTP header field, as would be normally sent by a web browser. The `url_original` attribute of the parent is used for this purpose, if available.

`referrer_cookie` — type: bytes — condition: p(`save_referrer_cookie`)
Reference to the `CookieList` structure of the parent URL. See `cookies` attribute for details.

`content` — type: bytes — condition: a^n(`type`) = `file`
Contents of the downloaded file.

6.2. Dumping JavaScript sources

The web client normally executes JavaScript embedded in a web page in order to generate dynamic content and observe any links or redirects that may emerge. We can take advantage of this existing functionality and intercept all scripts that are present in a web page before they are run by a scripting engine. Specifically, we want to capture the following data: (1) code placed in `<script>` elements, (2) remote code referenced from the `src` attribute of `<script>` elements and (3) event handlers. Additionally, arguments of every

`eval` call should be saved at runtime, which will allow to capture deobfuscated contents of scripts. We will use the term *contexts* for all fragments of JavaScript source mentioned here.

By doing this type of limited static and dynamic analysis in the web client, we provide data to other JavaScript analyzers in a format that is easier to process. A similar approach to extracting plain JavaScript code is used in Zoozle malware detector [7].

Dumping JavaScript sources feature is controlled by the following service configuration parameter:

`save_js_context` mandatory: no type: boolean default value: true
All JavaScript contexts encountered by the engine are saved in the `js_context_list` attribute.

When this feature is active, the following attribute is added to all objects corresponding to visited web pages:

`js_context_list` type: bytes condition: p(`save_js_context`) ∧ *JavaScript was present in the page*
Contains a list of all JavaScript contexts, with some additional meta-data. This attribute contains `JSContextList` structure. It is important to assign contexts to appropriate objects in the whole tree of visited pages (see section 6.1) — only JavaScript which execution was triggered by a particular subpage can be present in its `js_context_list` attribute.

7. Flash services

This chapter contains data contracts for all services dedicated to analyzing Flash (SWF) files.

7.1. Flash CVE analyzer (swf-cve)

This analyzer perform static analysis of SWF files in order to detect exploits for known vulnerabilities.

7.1.1. Service configuration

The service does not currently have any configuration options.

7.1.2. Input data

Flash CVE analyzer works with raw SWF file and does not use any contextual information.

`content` mandatory: yes type: bytes
Reference to the contents of the original SWF file.

7.1.3. Appended attributes

All results generated by the service are saved in attributes of the current object — no new objects are created.

`swf_cve_time_begin` type: timestamp condition: always
Time when the service started to process its current task. Format: seconds in UNIX time.

`swf_cve_time_end` type: timestamp condition: always
Time when the service finished processing its current task. Format: seconds in UNIX time.

`swf_cve_detected` type: integer condition: always
Number of exploits that were detected in `content` by the analyzer.

`swf_cve_list` type: bytes condition: a(`swf_cve_detected`) > 0
List with identifiers of all exploits that were found by the analyzer. Exploits are identified by CVEs, i.e. vulnerabilities they target.

```
1 message CveList {
2   repeated string cve = 1;
3 }
```

7.2. Flash shellcode scanner (swf-scs)

This analyzer perform static analysis of SWF files and extracts fragments that may contain shellcode. These fragments are reported as new HSN objects in an unobfuscated (binary) form.

7.2.1. Service configuration

The service does not currently have any configuration options.

7.2.2. Input data

Flash shellcode scanner works with raw SWF file and does not use any contextual information.

`content` mandatory: yes type: bytes
Reference to the contents of the original SWF file.

7.2.3. Appended attributes

The analyzer adds a small set of attributes to the current object since most of the output is reported as new objects.

`swf_scs_time_begin` type: timestamp condition: always
Time when the service started to process its current task. Format: seconds in UNIX time.

`swf_scs_time_end` type: timestamp condition: always
Time when the service finished processing its current task. Format: seconds in UNIX time.

`swf_scs_extracted` type: integer condition: always
Number of potential shellcode fragments that were extracted by the service.

7.2.4. New objects

When the service finds a fragment of the SWF file with containing binary data that might contain shellcode, it generates a new object with following attributes:

`type` — type: string — condition: always
Type of the new object. All objects created by the analyzer are potentially containing shellcode, so it is always set to `shellcode`.

`origin` — type: string — condition: always
Name of a service that created the object. For this service the value is always set to `swf-shell`.

`origin` needs to be described in more detail.

`parent` — type: object — condition: always
Reference to the parent of the new object. Since the service itself creates the new object, it is always a reference to the *current* object.

`content` — type: bytes — condition: always
Fragment of the SWF file that may contain shellcodes. It is saved in the binary form.

8. Shellcode analyzers

This chapter describes different shellcode analyzers in HSN 2.0. These services can perform more in-depth analysis on binary data extracted by other components.

8.1. Libscizzle-based shellcode analyzer (shell-scizzle)

This service is a wrapper around the libscizzle library for detecting shellcodes in arbitrary binary streams [8].

8.1.1. Service configuration

Configuration options of the service reflect parameters exposed by libscizzle itself or control how its results are saved.

`disassemble` mandatory: no type: boolean default value: true
If set to true, then the disassembled shellcode is saved in the `disassembled_shellcodes` attribute.

`disassemble_length` mandatory: no type: integer default value: 0
Maximum length of data to disassemble (counting from the offset detected by libscizzle). When set to 0, disassembling is performed till the end of the binary content.

8.1.2. Input data

Libscizzle works on opaque binary streams — no additional information is required, therefore there is only a single mandatory parameter:

`content` mandatory: yes type: bytes
Binary data that may contain shellcode. It is assumed that the binary data is already transformed to the "final" form, i.e. processor instructions can be explicitly detected in it and are not obfuscated in any way.

8.1.3. Appended Attributes

All results from libscizzle are appended as attributes to the current object:

`libscizzle_count` type: integer condition: always
Number of different shellcode fragments detected. Values greater than zero mean that there is a high degree of probability (since libscizzle has good false-positives rate) that the `content` attribute is malicious.

`libscizzle_offsets` type: bytes condition: a(`libscizzle_count`) > 0
Results obtained from libscizzle: a list of offsets where shellcode fragments were found. These values are relative from the beginning of the binary stream passed on input.

```
1 message ShellcodeOffsets {
2   repeated uint64 offsets = 1;
3 }
```

`libscizzle_asms` type: bytes condition: a(`libscizzle_count`) $> 0 \wedge$ p(`disassemble`)
Results obtained from libscizzle: a list of strings containing disassembled shellcode fragments. Order of elements on the list is the same as in the `libscizzle_offsets` attribute.

```
1 message ShellcodeASMs {
2   repeated string ASMs = 1;
3 }
```

8.2. Scdbg-based shellcode analyzer (shell-scdbg)

This service is able to detect shellcodes in arbitrary binary content using scdbg. Moreover, it can perform run-time analysis by executing the shellcode and emulating many aspects of the standard Windows environment [11].

The service should execute scdbg exactly $n+1$ times, where n is the number of shellcodes detected in the input file. During the first run, the tool is used to detect shellcodes and the following command line options should be sufficient for this task: `/a /getpc`. If some offsets were found, the tool is run again for each of them, using the following options: `/nc /G` *`graph-file`* `/mm /mdll /a /d /hex /foff` *`offset`*, where *`graph-file`* is a temporary filename for the output DOT file and *`offset`* is the currently examined offset.

8.2.1. Service configuration

The service does not have any configuration parameters currently.

8.2.2. Input data

Similarly to libscizzle, scdbg requires only a single a binary stream on input.

`content` mandatory: yes type: bytes
Binary data that may contain shellcode. It is assumed that the binary data is already transformed to the "final" form, e.g. processor instructions can be explicitly detected in it and are not obfuscated in any way.

8.2.3. Appended Attributes

All information gathered about the input binary data is saved in attributes of the current object.

`scdbg_offsets` type: integer condition: always
Number of possible shellcodes that were detected by the tool.

`scdbg_results` type: bytes condition: *possible shellcodes detected*
List of scdb analyses performed for all offsets. Each element of the list is a `ScdbgResult` structure holding complete output generated by scdbg.

```
1 message ScdbgResultList {
2   repeated ScdbgResult results = 1;
3 }
```

`offset` field contains offset of the suspected shellcode relative to the beginning of the file. Everything written by the tool to the standard output is stored directly in `output` field. Generated files are kept separately — DOT source in the `graph` field and the memory dump of the shellcode in the `dump` field. In some cases these files may not be written by the tool, e.g. dumps are not saved if no changes to the code in memory were detected.

```
1 message ScdbgResult {
2   required uint32 offset = 1;
3   optional string output = 2;
```

```
4   optional Reference dump = 3;
5   optional Reference graph = 4;
6 }
```

8.2.4. New objects

If the analyzer encounters an URL in arguments of any function, it creates a new object with the attributes defined below. URL detection can be performed by looking for any strings beginning with a `http` string in the scdbg output.

`type` — type: string — condition: always
Type of the new object. All objects created by scdbg-based analyzer are URLs, hence they always set this attribute to `url`.

`origin` — type: string — condition: always
Name of a service that created the object. For this service the value is always set to `shell-scdbg`.

`origin` needs to be described in more detail.

`url_original` — type: string — condition: always
Target URL in the original form, as was passed to the Windows API. Escape sequences or other types of obfuscation are not removed.

`parent` — type: object — condition: always
Reference to the parent of the new object. Since the service itself creates the new object, it is always a reference to the *current* object.

9. URL extraction services

This chapter describes analyzers that are able to extract URLs from different types of content.

9.1. Generic URL scanner (url-scan)

This service performs static analysis of arbitrary binary data in attempt to detect sequences looking like URLs. Detected URLs are represented as new objects in the system.

9.1.1. Service configuration

Service parameters control what kind of information is saved in attributes of new objects.

`add_referrer` mandatory: no type: boolean default value: true
Adds the referrer's URL to child objects in the `referrer` attribute.

`add_referrer_cookie` mandatory: no type: boolean default value: true
Adds the `referrer_cookies` attribute to child object with a reference to cookies from the parent page.

9.1.2. Input data

The service requires only a single attribute of the current object that represents data to scan. Other attributes will be used as contextual information for child objects.

`content` mandatory: yes type: bytes
Arbitrary binary data that should be scanned for URLs. The analyzer expects URLs in complete format, including protocol specification (e.g. `https://`), however they might be escaped.

`referrer` mandatory: no type: string
Referrer URL in the current context. If passed on input, this string is copied to all child objects.

`referrer_cookie` mandatory: no type: bytes
Referrer's cookies in the current context. If passed on input, it is copied to all child objects.

9.1.3. Appended attributes

Since new URLs are outputted as objects, the amount of information appended to the current information is minimal.

`url_scan_time_begin` type: timestamp condition: always
Time when the service started to process its current task. Format: seconds in UNIX time.

`url_scan_time_end` type: timestamp condition: always
Time when the service finished processing its current task. Format: seconds in UNIX time.

`detected_urls` type: integer condition: always
Number of detected URLs detected in `content`.

9.1.4. New objects

Attributes appended to a new object are not limited to basic information about the detected URL (e.g. `url_original`) but also include contextual data, if available.

`type` type: string condition: always
Type of the new object. All objects created by the analyzer are URLs, hence it is always set to `url`.

`origin` needs to be described in more detail.

`origin` type: string condition: always
Name of a service that created the object. For this service the value is always set to `url-scan`.

`url_original` type: string condition: always
Target URL. It should be saved in a form acceptable by web browsers, so some degree of unobfuscation might be necessary. It does not have to be completely normalized.

`parent` — type: object — condition: always
Reference to the parent of the new object. Since the service itself creates the new object, it is always a reference to the *current* object.

`referrer` — type: string — condition: p(`save_referrer`) ∧ a(`referrer`)
HTTP *referer* field, used if the analyzed content is executed in the browser context. a^n(`referrer`) = a(`referrer`).

`referrer_cookie` — type: bytes — condition: p(`save_referrer_cookie`) ∧ a(`referrer_cookies`)
Reference to the `CookieList` structure containing cookies that are used in this context (see page 26 for details). a^n(`referrer_cookies`) = a(`referrer_cookies`).

10. JavaScript analyzers

This chapter contains data contract for JavaScript analyzer. Currently there is only a single service of this type.

10.1. Static n-gram JavaScript analyzer (js-sta)

This analyzer is based on Low Interaction component of the first version of HSN system. Detailed design is described in a separate document [12].

10.1.1. Service configuration

Most of the classification process performed by the analyzer cannot be parametrized from the workflow yet. The only parameters exposed by the service are limited to keyword selection, used for confirmation of automatic classification.

`keywords_malicious` mandatory: no type: string default value:
`Shell.Application|ADODB.Stream|WScript.Shell|\.exe|\.bat|ms0[6-9]|shellcode|block|heap|spray|exploit|overflow|savetofile`
Regular expression (POSIX Extended Regular Expression syntax) that defines patterns that often appear in malicious JavaScript source code.

`keywords_suspicious` mandatory: no type: string default value:
`top.location|document.location|window.location|document.write|document.writeln|eval|location.replace|location.reload|location.href|document.body.innerhtml|location=`
Regular expression (ERE syntax) that defines patterns that often appear in malicious JavaScript source code but are also popular on benign sites.

10.1.2. Input data

JavaScript static analyzer uses data captured from a web page by the web client service.

`js_context_list` mandatory: yes type: bytes
List of all JavaScript contexts that were captured by the web client. See section 6.2 for more detailed description of its contents.

10.1.3. Appended attributes

The analyzer save all its results in attributes of the current object. The same set of attributes is appended, regardless of the service configuration or classification results.

`js_sta_time_begin` type: timestamp condition: always
Time when the service started to process its current task. Format: seconds in UNIX time.

`js_sta_time_end` type: timestamp condition: always
Time when the service finished processing its current task. Format: seconds in UNIX time.

`js_classification` type: string condition: always
Overall classification of the JavaScript contents in the page. It can take one of the following values:

`malicious` JavaScript is close to the "malicious" category in the training data set

`obfuscated` JavaScript is close to the "obfuscated" category in the training data set

`benign` JavaScript is considered not harmful or all suspicious contexts were whitelisted

`unclassified` it was not possible to classify the code, usually because not enough n-grams were generated

`js_malicious_keywords` type: boolean condition: always
Malicious patterns were detected in one or more contexts.

`js_suspicious_keywords` type: boolean condition: always
Suspicious patterns were detected in one or more contexts.

`js_sta_results` type: bytes condition: always
`JSStaticResults` structure holding analysis results for individual contexts.

```
message JSStaticResults {
  repeated JSContextResults results = 1;
}
```

Details for each context are put in `JSContextResults` structures. The `id` field is used to associate results with an input context. The `classification` field holds the result of the Bayes classifier, while `malicious_keywords` and `suspicious_keywords` contain a list of matched keywords that appeared in the given context. `whitelist` indicates if the classification was overridden or not.

```
message JSContextResults {
  enum JSClass {
    MALICIOUS = 1;
    OBFUSCATED = 2;
    BENIGN = 3;
    UNCLASSIFIED = 4;
  }
  required uint32 id = 1;
  required JSClass classification = 2;
  repeated string malicious_keywords = 3;
  repeated string suspicious_keywords = 4;
  required bool whitelisted = 5;
}
```

11. Razorback nuggets as services

This chapter contains the data contract for the Razorback nuggets which were adapted to work with the HSN 2.0 system. Currently there are six services in this category. Five of them are analyzers and one is a data extraction service. The title of each section that describes a nugget consists of a descriptive name of the service followed by the original name of the nugget. The workflow service name is also given in parentheses.

11.1. Information regarding all nuggets

All nuggets (except for archiveInflate) share a common set of possible classifications which can be set for the processed input.

`malicious` Input file was reported to be malicious.

`suspicious` Input file was reported to be suspicious.

`benign` The analyzer reported that the file is clean.

The return value from a nugget binary is informative whetheroften requires interpretation which may be specific to the nugget.

11.2. Flash analyzer nugget - swfScanner (rb-swfscanner)

This nugget performs static analysis of SWF files. It is capable of detecting exploits for four different vulnerabilities.

11.2.1. Service configuration

The service does not currently have any configuration options.

11.2.2. Input data

The service works with raw SWF files and does not require any contextual information.

`content` mandatory: yes type: bytes
Reference to the contents of the original SWF file.

11.2.3. Appended attributes

All results generated by the service are saved in attributes of the current object — no new objects are created.

`rb_swfscanner_return_value` type: integer condition: always
The return value from the nugget binary.

`rb_swfscanner_classification` type: string condition: a(`rb_swfscanner_return_value`) = 0
The classification of the scanned file.

`rb_swfscanner_verdict_message` type: string condition: a(`rb_swfscanner_classification`) ∈ {`malicious`, `suspicious`}
The verdict message from the nugget. Typically it is the name of the exploit.

`rb_swfscanner_verdict_priority` type: integer condition: a(`rb_swfscanner_classification`) ∈ {`malicious`, `suspicious`}
The priority which was given to the verdict.

`rb_swfscanner_bid` type: string condition: *exploit with a known BID found*
Detected exploit identified by its Bugtraq identifier (BID).

`rb_swfscanner_cve` type: string condition: *exploit with a known CVE found*
Detected exploit identified by its CVE identifier.

11.3. PDF analyzer nugget - pdfFox (rb-pdffox)

This nugget performs static analysis of PDF files. It currently detects 26 different CVEs.

11.3.1. Service configuration

The service does not currently have any configuration options.

11.3.2. Input data

The service works with PDF files and does not require any contextual information.

`content` mandatory: yes type: bytes
Reference to the contents of the original PDF file.

11.3.3. Appended attributes

All results generated by the service are saved in attributes of the current object — no new objects are created.

`rb_pdffox_return_value` type: integer condition: always
The return value from the nugget binary.

`rb_pdffox_classification` type: string condition: a(`rb_pdffox_return_value`) = 0
The classification of the scanned file.

`rb_pdffox_verdict_message` type: string condition: a(`rb_pdffox_classification`) ∈ {`malicious`, `suspicious`}
The verdict message from the nugget. Typically it is the name of the exploit.

`rb_pdffox_verdict_priority` type: integer condition: a(`rb_pdffox_classification`) ∈ {`malicious`, `suspicious`}
The priority which was given to the verdict.

`rb_pdffox_cve` type: string condition: a(`rb_pdffox_classification`) ∈ {`malicious`, `suspicious`}
Detected exploit identified by its CVE identifier.

11.4. Microsoft Office analyzer nugget - officeCat (rb-officecat)

This nugget performs a static analysis of Microsoft Office files. It currently detects 51 different CVEs.

11.4.1. Service configuration

The service does not currently have any configuration options.

11.4.2. Input data

The service works with Office files and does not require any contextual information.

`content` mandatory: yes type: bytes
Reference to the contents of the original Office file.

11.4.3. Appended attributes

All results generated by the service are saved in attributes of the current object — no new objects are created.

`rb_officecat_return_value` type: integer condition: always
The return value from the nugget binary.

`rb_officecat_classification` type: string condition: a(`rb_officecat_return_value`) = 0
The classification of the scanned file.

`rb_officecat_verdict_message` type: string condition: a(`rb_officecat_classification`) ∈ {`malicious`, `suspicious`}
The verdict message from the nugget. Typically it is the name of the exploit.

`rb_officecat_verdict_priority` type: integer condition: a(`rb_officecat_classification`) ∈ {`malicious`, `suspicious`}
The priority which was given to the verdict.

`rb_officecat_cve` type: string condition: a(`rb_officecat_classification`) ∈ {`malicious`, `suspicious`}
Detected exploit identified by it's CVE identifier.

11.5. Clam AntiVirus nugget - clamavNugget (rb-clamavnugget)

This nugget passes files on to the configured clamav-daemon for scanning.

11.5.1. Service configuration

The service does not currently have any configuration options.

11.5.2. Input data

The service works with any file and does not require any contextual information.

`content` mandatory: yes type: bytes
Reference to the contents of the file.

11.5.3. Appended attributes

All results generated by the service are saved in attributes of the current object — no new objects are created.

`rb_clamavnugget_return_value` type: integer condition: always
The return value from the nugget binary.

`rb_clamavnugget_classification` type: string condition: a(`rb_clamavnugget_return_value`) = 0
The classification of the scanned file.

`rb_clamavnugget_verdict_message` type: string condition: a(`rb_clamavnugget_classification`) ∈ {`malicious`, `suspicious`}
The verdict message from the nugget. Typically it is the name of the exploit.

`rb_clamavnugget_verdict_priority` type: integer condition: a(`rb_clamavnugget_classification`) ∈ {`malicious`, `suspicious`}
The priority which was given to the verdict.

`rb_clamavnugget_malwarename` type: string condition: a(`rb_clamavnugget_classification`) ∈ {`malicious`, `suspicious`}
Name of the detected malware.

11.6. VirusTotal nugget - virusTotal (rb-virustotal)

This nugget calculates the MD5 hash of input files and checks if they are associated with any known malware, using `https://www.virustotal.com/` database.

11.6.1. Service configuration

The service does not currently have any configuration options.

11.6.2. Input data

The service works with any file and does not require any contextual information.

`content` mandatory: yes type: bytes
Reference to the contents of the file.

11.6.3. Appended attributes

All results generated by the service are saved in attributes of the current object — no new objects are created.

`rb_virustotal_return_value` type: integer condition: always
The return value from the nugget binary.

`rb_virustotal_classification` type: string condition:
a(`rb_virustotal_return_value`) = 0
The classification of the scanned file.

`rb_virustotal_verdict_message` type: string condition:
a(`rb_virustotal_classification`) ∈ {`malicious`, `suspicious`}
The verdict message from the nugget. Typically it is the name of the exploit.

`rb_virustotal_verdict_priority` type: integer condition:
a(`rb_virustotal_classification`) ∈ {`malicious`, `suspicious`}
The priority which was given to the verdict.

`rb_virustotal_report` type: string condition:
a(`rb_virustotal_classification`) ∈ {`malicious`, `suspicious`}
Full scan report provided by VirusTotal with names of AV scanners and names of detected malware.

11.7. Archive Inflation nugget - archiveInflate (rb-archiveinflate)

This nugget extracts files from several types of archives.

11.7.1. Service configuration

The service does not currently have any configuration options.

11.7.2. Input data

The service works with archives. It requires information about the type of input file it is receiving.

`type` mandatory: yes type: string
Internal name of the filetype. Required to allow the nugget to load support for the appropriate archive format. The service transforms this using it's input mapping to the types recognized by the Razorback system. An appropriate mapping for each type needs to be defined.

`content` mandatory: yes type: bytes
Reference to the contents of the file.

11.7.3. Appended attributes

The service appends one attribute to the current object.

`rb_archiveinflate_return_value` type: integer condition: always
The return value from the nugget binary.

11.7.4. New objects

Each extracted file will create a new object. Duplicate files (same file hash and file size) are ignored.

`type` type: string condition: always
Type of the new object. Objects created by the service are not analyzed so their type is set to `undefined`.

`filename` type: string condition: always
The file's original name.

`sha256` type: string condition: always
The file's SHA256 hash.

`size` type: int condition: always
The file's size in bytes.

`parent`	type: object	condition: always
	Reference to the parent of the new object. Since the service itself creates the new object, it is always a reference to the *current* object.	
`content`	type: bytes	condition: always
	Contents of the extracted file.	

12. Client honeypots

This chapter contains data contract for complete client honeypots that are working as HSN 2.0 services.

12.1. Thug (thug)

Thug descirption from the official website [9]:

> The number of client-side attacks has grown significantly in the past few years shifting focus on poorly protected vulnerable clients. Just as the most known honeypot technologies enable research into server side attacks, honeyclients allow the study of client-side attacks.
>
> A complement to honeypots, a honeyclient is a tool designed to mimic the behavior of a user-driven network client application, such as a web browser, and be exploited by an attacker's content.
>
> Thug is a Python low-interaction honeyclient aimed at mimicing the behavior of a web browser in order to detect and emulate malicious contents.

12.1.1. Service configuration

Service configuration is described by a set of parameters which correspond to underlying Thug configuration or complement it. The parameters described below are related to URL processing.

`useragent` mandatory: no type: string default value: Parameter provides name identifying the personality which Thug will use to visit the website.

`proxy` mandatory: no type: string default value: Parameter provides an URI to the proxy that Thug will use to visit the website. Supported URIs are present in Thug specification.

`verbose` mandatory: no type: boolean default value: true
If set to true, Thug log will contain verbose output.

`debug` mandatory: no type: boolean default value: false
If set to true, Thug log will contain debug output.

`save_zip` mandatory: no type: boolean default value: false
If set to true, the zipped directory with analysis results will be saved and referenced in the `thug_analysis_zip` attribute.

`save_js_context` mandatory: no type: boolean default value: true
All JavaScript contexts encountered by the engine are saved in the `js_context_list` attribute.

12.1.2. Input data

In order to visit a web page, the service requires that the current object carries information about the target URL.

`url_original` mandatory: if a(`url_normalized`) missing type: string
The URL in its original form. If present, it will be used for requests to the remote server instead of the normalized form, in order to avoid detection.

`url_normalized` mandatory: if a(`url_original`) missing type: string
A canonical form of the URL to visit. It will be used for the initial HTTP request only if `url_original` attribute is not supplied.

`referrer` mandatory: no type: string
Referer string that will be sent to the remote server.

12.1.3. Appended Attributes

The service always appends some attributes related to the processed URL. Saving of additional data like log files and archived modifications from the virtual machine is entirely optional and depends on the configuration.

`thug_active` type: boolean condition: always
True if Thug produced syntactically correct `analysis.xml` file. Value of `thug_active` determines which other attributes will be appended.

`thug_behaviors` type: bytes condition: a(`thug_active`)
The attribute contains list of behaviors detected by Thug. The list is created from MAEC `analysis.xml` file.

```
message BehaviorsList {
  repeated Behavior behavior = 1;
}
```

Each of the behaviors is described by set of parameters as presented below. Parameters are taken directly from the MAEC output file produced by Thug.

```
message Behavior {
  required string description_text = 1;
  optional string discovery_method = 2;
}
```

`thug_error_message` type: string condition: $\neg$ a(`thug_active`)
Error message explaining why the processing failed.

It is copied from the `stderr` produced by Thug.

`thug_time_start` type: timestamp condition: always
The attribute provides information about time the processing of URL started.

`thug_time_stop` type: timestamp condition: always
The attribute provides information about time the processing of URL finished.

`thug_analysis_file` type: bytes condition: a(`thug_active`)
The attribute provides reference to `analysis.xml` file produced by Thug.

`thug_analysis_zip` type: bytes condition: a(`thug_active`) $\wedge$ p(`save_zip`)
The attribute provides reference to zipped directory with analysis results produced by Thug.

Path to this directory is given at the end of Thug's log.

`js_context_list` type: bytes condition: p(`save_js_context`) $\wedge$ *JavaScript was present in the page*
Contains a list of all JavaScript contexts, with some additional meta-data. This attribute contains `JSContextList` structure. In that structure `eval` flag should always be

set to false, as Thug only lists context available on the website and does not list evaluated contexts.

12.1.4. New objects

The service does not produce new objects.

12.2. Cuckoo (cuckoo)

Cucko Sandbox description from the Cuckoo website [10]:

> In three words, Cuckoo Sandbox is a malware analysis system.
>
> Its goal is to provide you a way to automatically analyze files and collect comprehensive results describing and outlining what such files do while executed inside an isolated environment.
>
> It's mostly used to analyze Windows executables, DLL files, PDF documents, Office documents, PHP scripts, Python scripts, Internet URLs and almost anything else you can imagine.

12.2.1. Service configuration

Parameters below represent a selection of columns from Cuckoo Sandbox SQLite `queue` table. Apart from columns referenced here, other data must be automatically generated and provided to Cuckoo in order to start analysis. This includes md5 file hash, submission date etc. SQLite table structure can be found in Cuckoo Sandbox documentation on the official website [10].

Additional input parameters define detail level of the analysis report.

`timeout` mandatory: no type: integer default value: according to Cuckoo configuration
Analysis timeout.

`priority` mandatory: no type: integer default value: 0
Analysis priority. File with highest priority is analysed first.

`package` mandatory: no type: string default value: according to Cuckoo configuration
Package which should be run on the analysis object.

`vm_id` mandatory: no type: string default value: according to Cuckoo configuration
ID of the virtual machine on which Cuckoo should run an analysis.

`save_pcap` mandatory: no type: boolean default value: false
If set to true, the network traffic related to visitation of the URL will be saved and referenced in the `pcap` attribute.

`save_report_json` mandatory: no type: boolean default value: true
If set to true, JSON report will be saved and referenced in the `report_json` attribute.

`save_report_html` mandatory: no type: boolean default value: false
If set to true, HTML report will be saved and referenced in the `report_html` attribute.

`save_screenshots` mandatory: no type: boolean default value: true
If set to true, the screen-shot files from the virtual machine processing should be saved.

Parameters below are taken into consideration when classifier ([13]) runs:

`operating_system` mandatory: no type: string default value: see description
Operating system version which will be used in Volatility analysis. If this attribute is missing, Volatility should determine image type. This attribute must have a value of one of supported Volatility profiles (e.g. `WinXPSP3x86`). It should be noted that determining image type is an extremely slow operation, so this parameter should be omitted only at the last resort.

`rating_whitelist` mandatory: no type: see[13] default value: 0.5
Rating of the "Whitelist of connected processes" heuristic.

`rating_services` mandatory: no type: see[13] default value: 0.5
Rating of the "Running services" heuristic.

`rating_hidden` mandatory: no type: see[13] default value: 1.5
Rating of the "Hidden processes" heuristic.

`rating_orphan` mandatory: no type: see[13] default value: 1.5
Rating of the "Orphan threads" heuristic.

`rating_api_unknown` mandatory: no type: see[13] default value: 1.5
Rating of the "API hooks (unknown)" heuristic.

`rating_api_known` mandatory: no type: see[13] default value: 0.5
Rating of the "API hooks (not unknown)" heuristic.

`rating_malfind_pe` mandatory: no type: see[13] default value: 1.5
Rating of the "Malfind plugin (contains executable)" heuristic.

`rating_malfind` mandatory: no type: see[13] default value: 0.5
Rating of the "Malfind plugin (does not contain executable)" heuristic.

`behavioral_importance` mandatory: no type: see[13] default value: 1
Importance (multiply factor) of the behavioral analysis blacklist heuristic.

`rating_threshold_benign` mandatory: no type: see[13] default value: 1
Upper limit for the rating of a process to be considered benign. Any rating less than this value is considered benign. Rating equal to this value is considered suspicious or malicious (when rating_threshold_benign = rating_threshold_suspicious).

`rating_threshold_suspicious` mandatory: no type: see[13] default value: 1.5
Upper limit for the rating of a process to be considered suspicious. Any rating less than this value is considered benign. Any rating higher equal to this value is considered malicious.

12.2.2. Input data

Input for Cuckoo consists of a file or URL of a web page that should be visited. In the second case the URL is converted to a windows internet shortcut file (`.url`) automatically.

`content` mandatory: if a(`url_original`) and a(`url_normalized`) are missing type: bytes
Content of a file which should be a Cuckoo Sandbox target.

`url_original` mandatory: if a(`url_normalized`) and a(`content`) are missing type: string
The URL in its original form. If present, it will be used

for requests to the remote server instead of the normalized form, in order to avoid detection. If `content` is set, this attribute is ignored.

`url_normalized` mandatory: if a(`url_original`) and a(`content`) are missing type: string
A canonical form of the URL to visit. It will be used for the initial HTTP request only if `url_original` and `content` attributes are not supplied.

`filename` mandatory: no type: string
Filename under which file with `content` should be saved. If this is not set, filename should be an md5 hash of content. This attribute should be ignored if `url_original` or `url_normalized` is set.

12.2.3. Appended Attributes

The service always appends some attributes related to the processed file or URL. Saving of additional data like log files and archived modifications from the virtual machine is entirely optional and depends on the configuration.

`cuckoo_time_start` type: timestamp condition: always
The attribute provides information about time the processing of URL started.

`cuckoo_time_stop` type: timestamp condition: always
The attribute provides information about time the processing of URL finished.

`cuckoo_pcap` type: bytes condition: p(`save_pcap`)
The attribute provides reference to PCAP file produced when URL was visited.

`cuckoo_report_html` type: bytes condition: p(`save_report_html`)
The attribute provides reference to the HTML report produced by Cuckoo.

`cuckoo_report_json` type: bytes condition: p(`save_report_json`)
The attribute provides reference to the JSON report produced by Cuckoo.

`cuckoo_screenshots` type: bytes condition: p(`save_screenshots`)
Attribute provides reference to saved zipped screen-shot from URL processing.

`cuckoo_classification` type: string condition: optional
Value of `classification.class` attribute from JSON report. This is one of the following values:

`malicious` Input file / URL is considered to be malicious.

`suspicious` Input file / URL is considered to be suspicious.

`benign` Input file / URL is considered to be clean.

This attribute should be provided by an external classifier (like the one mentioned in [13]), which is not a part of vanilla Cuckoo Sandbox distribution.

`cuckoo_classification_reason` type: string condition: optional
Value of `classification.reason` attribute from JSON report. This is a human readable reason for the classification. This attribute should be provided by an external classifier (like the one mentioned in [13]), which is not a part of vanilla Cuckoo Sandbox distribution.

12.2.4. New objects

The service does not produce new objects.

12.2.5. Exceptions

Whenever Cuckoo analysis fails for some reason, Task Error message should be send. This should also be the case when there is a missing file that is supposed to be saved (e.g. JSON report).

13. Yara

13.1. Introduction

As Yara developers say about their project:

YARA is a tool aimed at helping malware researchers to identify and classify malware samples. With YARA you can create descriptions of malware families based on textual or binary patterns contained on samples of those families. Each description consists of a set of strings and a Boolean expression which determines its logic.

YARA is multi-platform, running on Windows, Linux and Mac OS X, and can be used through its command-line interface or from your own Python scripts with the `yara-python` extension.

13.2. Performance

Yara performance highly depends on the rule format. Some simple performance guidelines can be found on yara project website. It is important to follow those rules, as they affect the performance significantly.

13.3. Rule sources

There are several sources of Yara rules:

- ClamAV to Yara Python converter(available from Malware Analysis Cookbook) - these rules have to be manually adapted after conversion!
- Rules for detecting popular packers (from Yara Project Website)

13.4. Yara analyzer (yara)

Yara provides a well documented Python API. The rules have to compiled as rarely as possible, so I would suggest using weak reference cache for them.

13.4.1. Service configuration

Yara needs a set of rules in order to work. Rules can be given in two ways: as a string or in a file. There should also be a caching mechanism present, so that rules that are used several times are compiled only once.

`rules_filename` mandatory: if a(`rules_string`) is missing type: string
default value: empty
If this attribute is set, rules should be read from this filename, if they aren't already compiled in cache.

`rules_string` mandatory: if a(`rules_filename`) is missing type: string
default value: empty
If this attribute is set, rules should be read from string, if they aren't already compiled in cache. If attribute `rules_filename` is present, this attribute should be ignored.

13.4.2. Input data

Input for Yara consists of a file that need to be matched against rules.

`content` mandatory: yes type: bytes
Content of a file which Yara will match against rules.

13.4.3. Appended Attributes

The service always appends some attributes related to the processed file.

`yara_matches_found` type: boolean condition: mandatory
True, if Yara found any matches. False otherwise.

`yara_matches_list` type: bytes condition: if `yara_matches_found`
A list of rules that matched specified input file.

```
1 message YaraMatchesList {
2   repeated YaraMatch matches = 1;
3 }
```

Each match contains basic information returned by Yara. Only successful matches should be listed here.

```
1 message YaraMatch {
2   required string rule = 1;
3   optional string namespace = 2;
4 }
```

`yara_time_start`	type: timestamp	condition: always
	The attribute provides information about time the processing of the file started.	
`yara_time_stop`	type: timestamp	condition: always
	The attribute provides information about time the processing of the file finished.	

13.4.4. New objects

The service does not produce new objects.

14. Analyzing PCAP files

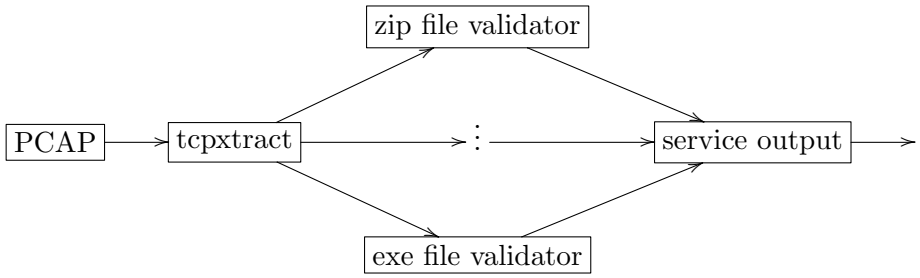
PCAP file analysis consists of several steps:

- Extracting files from PCAP (e.g. PE files, zip archives).
- Looking for unexpected traffic.
- Looking for traffic with malicious domains / IPs.
- Extracting known protocols (e.g. DNS).

PCAP files can also be submitted to Yara analyzer to find some suspicious patterns.

14.1. PCAP file extractor (pcap-extract)

Easiest way to extract specific files from PCAP traffic is to use tcpxtract. It can extract various types of files from a TCP traffic. Each of the files should be run trough an additional validator, to check if the file has been correctly extracted. Service should be written in a modular way, i.e. it is easy to add new validators. Default validator doesn't check anything and assumes that file is properly formatted. Diagram below illustrates a flow of a PCAP file through an extractor:



Because validators are in close link with tcpxtract configuration, they must be configured via a separate configuration file, not via workflow. Also, files that do not pass validation must not be in the service output.

14.1.1. Service configuration

All the service configuration is done trough the configuration files.

14.1.2. Input data

Input data consists of the PCAP file on which extraction will occur.

`pcap_content` mandatory: yes type: bytes
Content of a PCAP file.

14.1.3. Appended Attributes

There are no appended attributes.

14.1.4. New objects

This services produces objects that originates from a PCAP file (e.g. trough extraction).

`file_type` type: string condition: a^n(`type`) = `file`
This is a type of an extracted file. This type is extension of an output file produced by `tcpxtract`.

`content` type: bytes condition: a^n(`type`) = `file`
Content of a file extracted by `tcpxtract` and properly validated. Files that are not validated are not included in a service output.

14.2. PCAP analyzer (pcap-analyze)

14.2.1. Looking for unexpected traffic

In some cases it is unusual for an analyzed object to produce network traffic. That is why this service should add an attribute if some network traffic was registered.

However, in every analysis there is an expected network traffic (e.g. Windows Update, ntp etc.). In some cases this traffic is impossible or very hard to eliminate (DHCP request, ARP). That is why some of the network traffic should be "whitelisted" by the configuration file.

Each line of file is empty, contains a comment or one tshark filter. Comments starts with a hash sign (`#`). Filter corresponds to a traffic that is considered

benign. All filters should be provided to tshark, concatenated using `or` operator, and if there is any other network traffic than the filtered one, flag `pcap_network_traffic` should be set.

Example

This is raw (i.e. without applying any filters) output of tshark:

```
$ tshark -r dump.pcap
  1   0.000000              ->              Ethernet 60 [Packe...
  2   6.380619 CadmusCo_bd:3f:98 -> Broadcast    ARP 60 Who ha...
  3   6.380733 RealtekU_12:35:02 -> CadmusCo_bd:3f:98 ARP 42 1...
  4   6.380855    10.0.2.15 -> 10.0.2.2     ICMP 60 Echo (ping...
  5   6.380958 RealtekU_12:35:02 -> Broadcast    ARP 42 Who ha...
  6   6.381069 CadmusCo_bd:3f:98 -> RealtekU_12:35:02 ARP 60 1...
  7   7.381964    10.0.2.15 -> 10.0.2.2     ICMP 60 Echo (ping...
  8   7.382081     10.0.2.2 -> 10.0.2.15    ICMP 75 Time-to-li...
  9   8.383605    10.0.2.15 -> 10.0.2.2     ICMP 60 Echo (ping...
 10   8.383742     10.0.2.2 -> 10.0.2.15    ICMP 75 Time-to-li...
 11   9.386616    10.0.2.15 -> 255.255.255.255 DHCP 345 DHCP R...
 12   9.386914     10.0.2.2 -> 10.0.2.15    DHCP 590 DHCP ACK ...
 13  14.139602    10.0.2.15 -> 195.187.238.2 DNS 84 Standard q...
```

Suppose that the config file looks like this:

```
not eth
arp
# BOOTP is used as a transport protocol for DHCP
bootp

icmp
dns
```

Then, because all of the traffic is whitelisted, `pcap_network_traffic` attribute is set to false. However, if the config file looks like this:

```
not eth
arp
# BOOTP is used as a transport protocol for DHCP
bootp

icmp
# dns
```

Then `pcap_network_traffic` attribute is set to true, because of the dns traffic (frame number 13).

14.2.2. Looking for traffic with malicious domains / IPs

Some domains or IP address are considered malicious. there can be multiple external sources of malicious domains IPs. Service should create a list of all IPs involved in a network communication (`ip.addr` tshark filter) and all DNS domain queries (`dns.qry.name` tshark filter). This can be later used to check whether any of these values are considered malicious or not.

14.2.3. Extracting known protocols

Some useful information can be gathered by extracting known protocols from PCAP file. In order to do this, user can define filter on tshark (which would only filter packets that he is interested in) and fields that interest him. Optionally, service should provide an ability to remove duplicates from the results. This can be useful, when we want to list all domains that were contacted during an analysis.

Example

Assume that user set the following service configuration:

```
protocol_filter = dns
protocol_fields = dns.qry.name, dns.qry.type
protocol_unique = true
```

Then, in the example used above, service should create an attribute `pcap_protocol_dns` with one element list with the following contents:

```
frame = wp.pl 0x0001
```

This has been extracted using output of the following command:

```
tshark -r dump.pcap -Tfields -e dns.qry.name -e dns.qry.type
```

It is worth noting that all empty lines were removed from the output.

14.2.4. Service configuration

`allowed_traffic` mandatory: no type: string default value: no filter provided
Parameter providing a configuration file with the allowed (i.e. not considered suspicious) network traffic. Format of the file is provided above. If this parameter is not set, then there isn't any traffic allowed. If the configuration file does not exists, service should fail.

`save_ip` mandatory: no type: boolean default value: false
Parameter that defines if IP adresses that are present in the PCAP file should be saved or not.

`save_dns` mandatory: no type: boolean default value: false
Parameter that defines if DNS queries that are present in the PCAP file should be saved or not.

`protocol_filter` mandatory: no type: string default value: no filter provided
Filter that will be used by tshark to read the PCAP file. If there is no filter provided, protocol extraction will not take place.

`protocol_fields` mandatory: no type: string default value: default tshark fields
Fields that will be used in tshark output when extracting protocols.

`protocol_unique` mandatory: no type: boolean default value: false
If this attribute is provided, only unique protocol extraction results should be provided.

14.2.5. Input data

Input data consists of the PCAP file on which extraction will occur.

`pcap_content` mandatory: yes type: bytes
Content of a PCAP file.

14.2.6. Appended Attributes

`pcap_network_traffic` type: boolean condition: always
The attribute provides information whether or not network traffic (apart from the whitelisted one) was present. For the detailed description, read the section above.

`pcap_protocol_<protocol_filter>` type: bytes condition: p(`protocol_filter`)
This attribute provides results of PCAP protocol extraction. This has a form described below. It should be noted that this parameter name depends on the value of `protocol_filter` parameter.

```
1 message ProtocolEntries {
2   repeated string frame = 1;
3 }
```

`pcap_ip` type: bytes condition: p(`save_ip`)
This attribute contains a list of all IP addresses present in the PCAP file as presented below.

```
1 message IpAddresses {
2   repeated string ip = 1;
3 }
```

`pcap_dns` type: bytes condition: p(`save_dns`)
This attribute contains a list of all DNS queries present in the PCAP in a format described in 3.2.

14.2.7. New objects

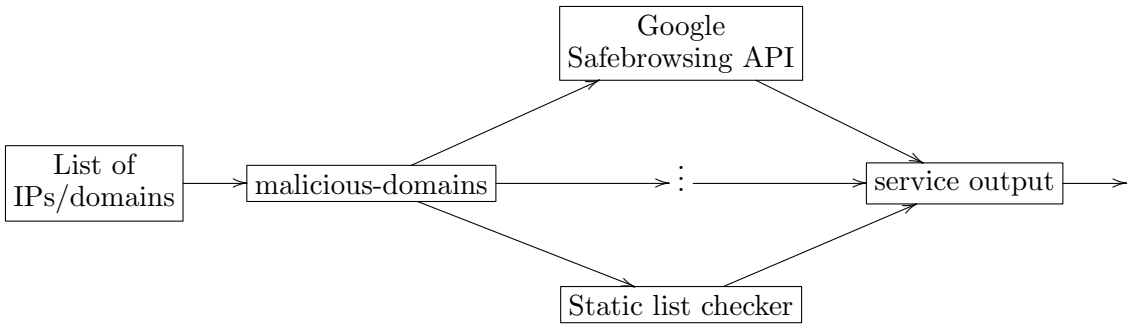
The service does not produce new objects.

14.3. Checking for malicious domains (malicious-domains)

There are several malcious domains APIs, one of the most prominent examples being Google safebrowsing API. There are also external sources, from which one can obtain a list of malicious domains and update it regularly. This service should provide an ability to query any kind of malicious domain service.

Therefore, it should be flexible enough to connect to any type of API and also check a list of malicious domains.

Architecture should be designed much like the `pcap-extract` service.



Each of the intermediate checker receives domains/IPs in a bulk list (because it can optimize searching) and performs one round of checking. For each domain it derives one of the following ratings:

- benign
- suspicious
- malicious
- unknown

Of course this intermediate checker is also responsible for any kind of mapping that takes place between the API and rating set described above. After all the checkers have run, `malicious-domains` should combine all of the results and put them in the `malicious_domains_verdict` attribute according to the format described below.

Each of the intermediate checkers should also provide its human-readable name and define accepted lists (IPs, DNS queries or both). If service does not support IPs or DNS queries, its result for that list should be assumed to be `unknown`. This means that for every IP/DNS query, there should be a result obtained in the `malicious_domains_verdict` attribute.

14.3.1. Service configuration

Service does not need any configuration. Available intermediate domain checkers should be listed in the configuration file.

14.3.2. Input data

`pcap_ip` mandatory: yes type: bytes
List of IPs, as described in the Appended Attributes section of the `pcap-analyze` service i.e.:

```
1     message IpAdresses {
2   repeated string ip = 1;
3 }
```

`pcap_dns` mandatory: yes type: bytes
List of DNS queries, as described in 3.2.

14.3.3. Appended Attributes

Service appends only one attribute, which is described in details above.

`malicious_domains_verdict` type: bytes condition: always
The attribute provides a verdict from the intermediate checkers. Verdict is in the following form:

```
1 message DomainVerdicts {
2   repeated DomainVerdict verdict = 1;
3 }
```

Where `DomainVerdict` is defined as follows:

```
1 message DomainVerdict {
2   required string checked_domain = 1;
3   repeated SingleCheckerVerdict checked = 2;
4 }
```

Where `SingleCheckerVerdict` is defined as follows:

```
1 message SingleCheckerVerdict {
2   required string name = 1;
3   required string verdict = 2;
4 }
```

14.3.4. New objects

The service does not produce new objects.

A. Redirects in web pages

This appendix contains information on all possible methods of redirecting a WWW browser that should be supported by the web client (see section 6.1).

A.1. Server-side redirects

A server-side redirect occurs when a remote HTTP server replies with a message with an `3xx` status code. Anything that happens before a HTTP request is sent, e.g. a DNS query that returns `PTR`, is are never considered as a redirect.

If a web client is redirected during its first request to the HTTP server, it creates a new object of the `url` type and stops processing. If the redirect occurs for any subsequent request for the content of the web page, it does not result in creating a new `url` object. Moreover, if the HTTP `3xx` reply does not contain the `Location` field, no new object is created (even when it is the first HTTP request).

A.2. Client-side redirects

Work in progress

- `window.open`
- `window.openDialog`
- `window.showModalDialog` Redirects are based on JavaScript. Code opens new window with URL given as argument for the function.
- `<meta http-equiv="refresh">` It should be treated as redirect regardless of the timeout value.
- `document DOM` Redirects of this kind are based on JavaScript. The JavaScript usually takes URL from a HTML element of the page and creates an *iframe*, or uses it as an argument for more complex transformations.

- `iframe` Redirect can be based on JavaScript. JavaScript usually writes an *iframe* with URL to redirect to.
- `location.href`
- `document.location.href`
- `window.location.href`
- `window.location`
- `self.location`
- `top.location`
- `location.replace`
- `window.location.replace`
- `window.navigate` Redirects are based on JavaScript. JavaScript that uses these objects or functions changes location of whole web page.
- `document.myform.submit` Redirect is based on JavaScript. It uses a form to force loading a new web page.
- `frameset` Creates a *frameset* with new URL. Redirect is based on JavaScript. It changes location of whole page.

Client-side redirects does not cause a web client to stop loading contents of the current page although. However, for redirects triggered from JavaScript, no further code should be executed.

B. Embedded resources and links

Webclient support only external sources defined in an HTML source. It does not support any CSS references, so e.g. background images defined in a CSS file (or `style` attribute) would not be processed in any way. Also tags that are undocumented shortcuts for defining styles (like non-standard use of `background` attribute) will not be processed.

Below is a list of all HTML tag/attribute pairs that can contain an URL as its value. This appendix specifies `webclient` behavior when encountering these attributes.

B.1. Embedded resources

Tag	Attribute	Remarks	Group
`applet`	`archive`	**list** of space separated URLs with applet data	`object`
`applet`	`code`	(relative) URL with applet data	`object`
`applet`	`codebase`	base URL for resolving relative URL given in other `applet` attributes	not downloaded
`audio`	`src`	source file of the audio	`multimedia`
`body`	`background`	image for the website background	`images`
`command`	`icon`	icon for the menu command supported only by Internet Explorer	`images`
`embed`	`src`	external file to embed to website	`objects`
`html`	`manifest`	application cache manifest	`others`
`img`	`src`	link to the image	`images`
`input`	`src`	**only when** `<input type = "image"...` image to use as a submit button	`images`
`link`	`href`	**only when** `<link rel = "stylesheet"...` stylesheet	`others`
`link`	`href`	**only when** `<link rel = "icon"...` favicon (downloaded by major browsers)	`images`

link	href	**only when** <link rel = "shortcut icon"... favicon (downloaded by major browsers)	images
object	classid	**only when proper URL** object implementation	objects
object	codebase	base URL for resolving relative URL given in other object attributes	not downloaded
object	data	object data	objects
video	poster	poster (thumbnail) image of the video	images
video	src	source file of the video	multimedia
script	src	source of the script	others
source	src	different sources for audio/video content	multimedia
td	background	image for the table row background	images

B.2. URL child objects

Tag	Attribute	origin value	Remarks
a	href	link	hyperlink
area	href	link	hyperlink on part of image
frame	src	frame	website frame
iframe	src	iframe	website iframe
img	longdesc	longdesc	long description for the image

B.3. Not present in `webclient` output

Tag	Attribute	Remarks
`blockquote`	`cite`	source of quotation (used by search engines)
`button`	`formaction`	requires a completed form
`del`	`cite`	reason for deletion (used by screen readers)
`form`	`action`	requires a completed form
`frame`	`longdesc`	not supported by major browsers
`head`	`profile`	not supported by major browsers
`iframe`	`longdesc`	not supported by major browsers
`input`	`formaction`	requires a completed form
`ins`	`cite`	reason for insertion (used by screen readers)
`q`	`cite`	source of quotation (used by search engines)
`td`	`background`	see the reason at the beginning of appendix
`tr`	`background`	see the reason at the beginning of appendix
`track`	`src`	not supported by major browsers

References

[1] HSN 2.0 Feeders List

[2] HSN 2.0 Crawler: Design Notes

[3] HSN 2.0 Communication Protocol

[4] HSN 2.0 Initial Data Contract Specification

[5] HSN 2.0 Format of Normalized URLs in HSN 2.0

[6] Hypertext Transfer Protocol — HTTP/1.1, `http://www.w3.org/Protocols/rfc2616/rfc2616.html`

[7] C. Curtsinger, B. Livshits, B. Zorn, C. Seifert, *Zozzle: Low-overhead Mostly Static JavaScript Malware Detection*, 2011. Technical Report MSR-TR-2010-156, Microsoft Research.

[8] Georg Wicherski *libscizzle* `http://code.mwcollect.org/projects/libscizzle/`

[9] Angelo Dell'Aera *Thug* `https://github.com/buffer/thug`

[10] Claudio "nex" Guarnieri, Dario Fernandes, Alessandro "jekil" Tanasi *Cuckoo Sandbox* `http://www.cuckoobox.org/`

[11] *scdbg* `https://github.com/dzzie/SCDBG`

[12] Specification of HSN 2.0 JavaScript Static Analyzer

[13] Specification of HSN 2.0 Cuckoo Classifier