TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

Mikk Margus Möll 192893IVSB

Improving compatibility between Shodan Stream and IntelMQ

Bachelor's thesis

Supervisor: Toomas Lepik
MSc Cybersecurity

Tallinn 2021

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Mikk Margus Möll 192893IVSB

SHODAN STREAMI JA INTELMQ TURVATEABEVOO ÜHILDUVUSE TÄIUSTAMINE

Bakalaureusetöö

Juhendaja: Toomas Lepik
MSc

Tallinn 2021

Author’s declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Mikk Margus Möll

17.05.2021

Abstract

The work mostly focuses on Shodan and IntelMQ, the former being a source of intel on internet-facing services, and the latter being a solution for processing data from feeds such as Shodan.

The first half of this thesis covers mapping a schema for the data format given by Shodan Stream, a security feed, and the difficulties in doing that. Based on the schema, the code for processing this data in IntelMQ is significantly improved and a working proof-of-concept schema generator is given. This can also be used to improve other projects making use of data from Shodan, which currently do not make use of much of the data due to the schema being mostly unknown.

The second half covers improving data filtering methods within IntelMQ. With Shodan's wide range of scanning capabilities, there are many results which do not have any security applications. At the same time, filtering goals can change over time. Due to this, a dynamic solution is beneficial. a Sieve-based DSL was extended with with new features, permitting for a solution that can be dynamically updated via changes to a script to match changing requirements. This allows users of IntelMQ to filter and perform some simple data processing on data from Shodan, but also other security feeds, within IntelMQ, with less complexity.

Keywords: Shodan, IntelMQ, data processing, DSL

The thesis is in English and contains 30 pages of text, 6 chapters and 2 figures.

Annotatsioon

Töö keskendub peamiselt Shodanil ja IntelMQ'l. Esimene neist on infoallikas internetiga ühendatud teenuste jaoks, teine on tarkvaraline lahendus erinevate turbeandmeallikate, näiteks Shodani, töötlemiseks.

Töö esimene pool hõlmab Shodan Stream'i antud andmete skeemi kaardistamist ning tõkkeid selle tegemises. Skeemi põhjal täiendatakse ka IntelMQ's Shodani andmete töötlemise koodi ning avalikustatakse ka näidisrakendus skeemi genereerimiseks. Seda saab kasutada ka teistes projektides, milles tarbitakse Shodanist pärinevaid andmeid, mis ei kasuta seda oma täies ulatuses skeemi üldise tundmatuse tõttu.

Töö teine pool hõlmab andmete filtreerimise parendamist. Arvestades Shodani skaneerimisvõimete laia ulatust ei ole kõigile antud andmetele võimalik leida turbealast rakendust. Teisalt võivad filtreerimisnõuded ajas muutuda. Seetõttu on dünaamiline lahendus soodne. Selle jaoks laiendati Sieve'il põhinevat valdkonnapõhist programmeerimiskeelt uue funktsionaalsusega, mis lubab, kasutades lisatud funktsionaalsust, reegleid dünaamiliselt muuta vastavalt muutuvatele nõuetele. See võimaldab IntelMQ kasutajatel andmeid filtreerida ning mõningaid lihtsamaid andmetöötlusprotseduure teostada vähema komplekssusega.

Võtmesõnad: Shodan, IntelMQ, andmetöötlus, DSL.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 30 leheküljel, 6 peatükki ja 2 joonist.

List of abbreviations and terms

API	Application Programming Interface
AS	Autonomous System
ASN	Autonomous System Number
AST	Abstract Syntax Tree
CERT	Computer Emergency Response Team
CSIRT	Computer Security Incident Response Team
CTIP	Cyber Threat Intelligence Program
DNS	Domain Name System
DSL	Domain-Specific Language
ELK	Elasticsearch, Logstash, Kibana
FQDN	Fully Qualified Domain Name
HTTP	Hypertext Transfer Protocol
ICS	Industrial Control System
ISP	Internet Service Provider
JSON	JavaScript Object Notation
MAC	Media Access Control
PoC	Proof of Concept
RDP	Remote Desktop Protocol
URL	Uniform Resource Locator
VNC	Virtual Network Computing
XMPP	Extensible Messaging and Presence Protocol

Table of Contents

List of Figures

1 Introduction

1.1 Introduction

A big part of the work of CSIRTs around the world is monitoring systems under their jurisdiction. There are now many commercial services that monitor the internet and provide access, in some form, to a feed of the data they have gathered as a service. CSIRTs may also have their own private feeds of data to keep track of. Almost all of these feeds will, however, vary vastly in the channel the data is passed through, its format, and the amount and kinds of data they provide. There are, however, tools to automate the collection and processing of data from these feeds. Abusehelper and IntelMQ are examples of such tools, with IntelMQ being the preferred tool of the two in recent years, with Abusehelper being essentially abandoned and IntelMQ seeing active development and improvement along with a wider range of supported security feeds.

One of the services an organization, CSIRT or otherwise, may use to keep track of internet-connected devices, is Shodan. While they are primarily known for their search engine service, they also provide a service known as Shodan Stream (or Shodan Firehose), which offers real-time access to the latest data added to the Shodan databases accessible via the Shodan search engine.

There is existing work on bringing the data from Shodan to IntelMQ, however, it is far from perfect [1], [2]. This is mainly due to the relatively small amount of data mapped from Shodan's data representation to IntelMQ's, leaving much useful data with no chance of being processed within IntelMQ or outside of it. Aside from that, there are issues with the Sieve bot, the primary method of general-purpose filtering data filtering within IntelMQ. Those are some limitations within the control flow of the code, and limitations in functionality, such as for working with booleans or lists of values. This thesis will explore methods of improving this integration and list some of the many unexpected obstacles in doing so.

The goal of this work is to significantly improve the usability of the Shodan Stream security feed with the IntelMQ feed processor. The motivation for this is that there is a gap between Shodan Stream's abundance of information and how much of it can be processed in practice. There is currently no good way to make use of it, within IntelMQ or otherwise. Making it usable within IntelMQ is a practical application of the results of the work done, in a system which is used by many organisations for processing similar security feeds.

In order to accomplish this, a dataset must be collected from Shodan Stream, which can then be analysed. For the analysis, a schema of the data will be generated, along with a deduplicated set of values corresponding to each data field. Following this, a method of filtering the data, flexible enough to be modified quickly with changing data filtering requirements, will have to be implemented.

For the data collection and analysis steps, custom stand-alone Python code will be used. For the filtering step, IntelMQ will be extended to improve filtering capabilities, with an emphasis on flexibility.

1.2 Methodology

Previous work within this area was analysed, within IntelMQ and in other projects, in order to assess the existing progress in parsing Shodan data in general and filtering it specifically within IntelMQ. An assessment was made that it was not sufficient, and more work was left to be done. From there, solutions for generating a schema for the Shodan data, and from there improving the parser within IntelMQ, and filtering data within IntelMQ were devised.

The code written for use within IntelMQ was submitted and accepted by the upstream developers, planned for release soon, and the standalone code for schema generation was published publicly on Github.

2 Background

2.1 IntelMQ

IntelMQ is a software project which has the goal of simplifying the automated collection and processing of various security feeds [3]. To do this, IntelMQ uses a network of connected “bots” called a “botnet”. These bots each have a specific task within the botnet and communicate with each other over a message queue protocol.

The bots may each have bot may have zero, one, or many inputs and outputs. For transferring data over these connections, the bots work with “events”, which are key-value maps (internally JSON at the time of writing, with plans to move to MessagePack).

For parsers and experts, there is no requirement that one input event maps to one output event. This is useful for parsers as one data piece of data from a feed may encompass the data for many events, such as in the case of CSV-based reports with multiple rows of data, and experts may wish to perform filtering on the events alongside modification.

Most bots can also be configured with some parameters, which can specify things such as operation procedures or credentials for authenticating with 3rd-party services.

The bots are divided into four different classes by their purpose:

- Collectors: These bots connect to different public and private security feeds to fetch raw data, such as Microsoft CTIP and Shadowserver as well as some generic collectors applicable to different feeds, and only produce output, taking no input
- Parsers: These bots are passed raw data collected by collector bots and they parse it into the IntelMQ format, often in feed-specific ways, taking input from collectors and producing parsed output

- Experts: These bots generally perform transformations on the data which do not fall under any of the other categories, such as deduplication, filtering, data enrichment (such as decoding the FQDN from an URL)
- Outputs: These bots take processed events and output them in some way, such as by adding them to a database or a file, and take input events, but do not produce output events within the IntelMQ network

A typical IntelMQ network is, composed of a series of collector-parser pairs (with some parsers potentially being shared by some collectors), feeding into a chain of experts, finally leading to output bots which pass the data out of the botnet.

One of the expert bots provided in IntelMQ is the Sieve bot. It is an expert bot which is configured using a file written in a DSL based on the Sieve language, which is used for filtering emails. It can match events based on a combination of matches on string, numeric and IP range based matches, and perform actions such as dropping the event, passing it on without further modification, passing it on through a specific output connected to the bot, and modifying the event. A graph demonstrating the data flow through IntelMQ is given in appendix 2.

2.2 Use of IntelMQ for cyber intelligence

As a project of ENISA, the European Union Agency for Cybersecurity [4], it is both developed and used by a number CERTs within the European Union. Though they may not advertise the tools they use publicly, the project has a number of contributors associated with CERTs on the Github page. Aside from that, it is remarkably useful in its capability to process many disparate formats of data from different data feeds that CERTs make use of into a common format.

IntelMQ is generally used alongside various other tools, for which it provides processed data as input. It collects, processes, potentially filters and then outputs data from many sources, but past that, the data needs to be made use of. For instane, CSIRT-KIT [5], a ready-made toolkit for data processing within a CSIRT, contains an instance of what a CSIRT may use. Within it, IntelMQ collects data from a variety of feeds, processes it and passes it on to the ELK stack, from where it can be analyzed and made use of elsewhere, such as directly through Kibana by an analyst or by other software fetching the data from Elastic.

2.3 AbuseHelper

AbuseHelper served as an automation tool similar to IntelMQ and was its predecessor and primary inspiration. It was created at a time when no other tool like it was available. At its core, it also uses discrete “bots”, which pass discrete data messages along pre-specified paths to each other. And, like IntelMQ, it has the concept of sources (collectors/parsers), data processors (experts) and outputs. For the data, it has a harmonization format as well, making automated processing possible.

Where it diverges, however, is in some finer details. For instance, instead of a message queue protocol, it uses XMPP, a chatroom protocol, to connect the bots. And unlike IntelMQ, the collection and parsing steps are combined, while IntelMQ has them as seperate bots. It is also less simple to extend and loses much of its state during unexpected shutdowns/crashes.

At its core, AbuseHelper’s design was superb. However, some issues in its implementation, some due to it being one of the first tools of its kind, it has been replaced with other similar tools like IntelMQ.

2.4 Shodan

Shodan is primarily known as a search engine offering searches to be performed on its large database of network scans around the world with, indexing millions of devices with publicly accessible services ranging from common web server software such as nginx or Apache HTTP Server to control system protocols such as Fox, Ethernet/IP and BACnet. It allows for hosts with known vulnerabilities to be found, either indirectly by allowing for services with certain versions to be searched for, or directly by allowing premium members to search for hosts that have vulnerabilities detected by Shodan itself, either actively through benign exploitation by Shodan, or, in certain cases, passively, based on metadata such as software versions.

Shodan Stream is an alternative to the search engine based service for premium users. It allows them to run software which passively queries an HTTP API endpoint to “listen” to the stream and get results based on a user-specified filter from Shodan as soon as the scans are finished. The data comes in the form of JSON objects called banners, with each banner corresponding to one service scanned on one host. Altogether, this provides a passive real-time stream of useful data from Shodan, without the need to actively poll for data from Shodan’s search engine.

Shodan Stream allows users to specify the banners they get in six different ways: all of them globally, banners from specific ASNs, from specific countries, from services on specific ports, from all subnets marked by the user for monitoring, and from specific subnet. The latter two depend on the user having specified a subnet or set thereof beforehand.

There are a selection of libraries for accessing the Shodan API, including Shodan Stream. Of particular note is the Python library, which is the only Shodan API library officially supported and developed by Shodan LLC. Though due to the simplicity of the API, all that is needed to implement a client is a TLS-capable HTTP(S) library and a JSON decoding library for parsing the response data.

2.5 Use of Shodan for cyber intelligence

As Shodan is a large repository of cyber intelligence data, it can be effectively used for identifying a large number of hosts with serious vulnerabilities. This can be done with both publicly known vulnerabilities indexed online, as well as in the case of zero-day vulnerabilities. Previous work focusing on a specific service confirmed that a number of vulnerable hosts could be enumerated, with 71.85%, a majority, of hosts checked having three year old firmware with a known vulnerability [7].

However, this also also shows that many hosts with known vulnerabilities can lie dormant on the public internet for years, long after news coverage on them has died out.

Botnets have been known to be formed using widespread unpatched vulnerabilities from years ago [8], [9]. Likewise, a service being indexed by Shodan has been demonstrated to not affect the volume of attacks, under some circumstances [10]. As such, Shodan and tools like it can be useful in identifying problematic or vulnerable hosts, such as on networks under one's jurisdiction.

In larger networks with millions of hosts, such as those under the jurisdiction of major national/governmental CERTs, there tend to be a large variety of network services. Searching for or even implementing software capable of collecting data on dozens or hundreds of different protocols, as well as performing tests for an ever-growing list of vulnerabilities, would be a major undertaking and resource drain for any organization. As such, using Shodan or tools like can be an attractive alternative to developing, maintaining and running software with similar capabilities.

There is already a variety of tools which make use of Shodan, either alone or alongside other feeds, and either using the search functionality, the stream, or both. A list of a selection of them is given in section 3. These range from simple libraries to fetch data from the Shodan APIs, such as the official Python library and some 3rd-party libraries in other languages, to full-fledged analysis frameworks such as Mihari and IntelMQ, which integrate with many feeds including Shodan.

2.6 Problem setup

Shodan stands out among the other security feeds handled by IntelMQ with how its data is given. Though it is, at first glance, possible to collect and parse data from Shodan within IntelMQ, it suffers from two major issues.

First, the Shodan data format schema is mostly undocumented. Very little of the schema of is documented by Shodan LLC [6], and not much more is documented by any 3rd party. Hence, not all of the data is fully processed in automated systems like the IntelMQ Shodan parser, or other similar software. This may increase costs and latency if collected data is processed manually (even partially), or reduce efficiency if much of the potentially useful data is left unprocessed by automated systems.

Second, while most of the feeds handled by IntelMQ or other such software provide only security-sensitive data, Shodan Stream is an exception to that rule. For instance, there is a large quantity of web and game servers on the internet running on versions with no known vulnerabilities. Until a point at which the software at that version is discovered to contain vulnerabilities, this data is noise. It could be stored separately in anticipation of a vulnerability disclosure in order to expedite alerts to the relevant administrators. However, at such a point, Shodan’s search engine capabilities would provide equivalent or even better results, with no need to reserve storage space.

Two solutions, one which solves the first partially solving the first issue, and the second providing tools to solve the second issue, will be provided.

For the first problem, the schema of the Shodan banners needs to be mapped. Based on that, the Shodan parser within IntelMQ can be improved to process more of the data included in each banner.

For the second problem, a sufficiently powerful and extensible method of filtering out useful Shodan banners within IntelMQ will be needed.

In order to improve integration between these two tools, I will explore methods to implement these two solutions and improve the integration between these two pieces of software.

3 Previous work

3.1 Shodan schema mapping

To see if there was significant work on mapping the Shodan schema, I examined the selection of Shodan API libraries listed by Shodan [2]. Some additional projects were also discovered on Github. However, for the purposes of this work, they were not useful. Not all of them included any banner format mappings, and those that did have it did not have many more parts of the banner format which had not already been mapped within the IntelMQ parser, and were vastly lesser than the schema mapped by my later work.

The following projects included no mapping of the Shodan banner format which could provide information on the schema:

- https://github.com/ScadaExposure/Shodan-PHP-REST-API,
- https://github.com/femiagbabiaka/shodan-rust
- https://github.com/picatz/shodanz
- https://github.com/tanprathan/Shomon
- https://github.com/Dudley5000/WWW-Shodan-API
- https://github.com/jesusprubio/shodan-client
- https://github.com/B-V-R/shodan
- https://github.com/ninoseki/mihari

The following projects included a similar or lesser mapping to that included in the IntelMQ parser:

- https://github.com/fooock/jshodan
- https://github.com/PercussiveElbow/Shodan
- https://github.com/achillean/shodan-python
- https://github.com/TerribleDev/Shodan.Net
- https://github.com/darkoperator/Posh-Shodan
- https://github.com/shadowscatcher/shodan
- https://github.com/ASRagab/scodan
- https://github.com/ns3777k/go-shodan
- https://github.com/cudeso/tools

Further attempts to find code that may have a significantly improved schema proved ineffective. In the absence of any such schema, I have decided to devise my own method of mapping it.

4 Format of Shodan banner data

4.1 General format

Each Shodan banner received through the Shodan Stream is JSON-encoded. The root object of a banner is a JSON object, a key-value map. All JSON-encodable value types (floating-point numbers, integers, booleans, strings, lists, objects and null) appear in different banners. Most of the data in banners is given as attributes of objects under the root object, meaning most data on services as FTP, HTTP, iSCSI or MongoDB is located in a objects located in the root object with the key “ftp”, “http”, “iscsi” or “mongodb” respectively. Some attributes, meanwhile, are given directly in the root object, such as the ASN, operating system, port of the given service and MAC addresses belonging to the scanned host which may be leaked by some services.

Some parts of the banner are always present. Examples of this are the IP address and port of the host and service scanned, and some metadata based on the IP address such as GeoIP data, ASN, AS name and ISP. Some metadata specific to Shodan rather than the target that was scanned is also always included.

Some sections of the banner may appear alongside many different scanned services. Examples of this are

- “ssl”, a section with info on the parameters of the SSL/TLS tunnel encapsulating another protocol (such as HTTPS, IMAPS or FTPS),
- “mac”, which contains info on MAC addresses found to belong to the host via data leaked from the scanned service,
- “vulns”, a list of vulnerabilities either suspected or confirmed to affect the service that was scanned (this is specified in the “verified” attribute)

Lastly, the vast majority of sections are specific to the type of service being scanned. These may provide varying levels of information, depending on the service and potentially on specific configuration settings.

One issue that came up during this work was that a variety of values may have very different types. For instance, the key in the chain “ssl” → “dhparams” → “generator”, which holds the generator parameter included in the Diffie-Hellman key exchange parameters sent by the server, can hold both integer and string values, depending on the size of the generator. Around 29 such occurrences were found.

4.2 How IntelMQ parses Shodan banners

The pipeline a Shodan banner ordinarily takes in an IntelMQ botnet is to get collected by the Shodan parser, then passed on to the Shodan parser. Within the parser, there are two methods of parsing the banner: with a regular mapping and a minimal mapping with less information. They work similarly enough that they will not be covered separately.

As the banner can have multiple levels of nesting, a recursive approach is used. The parser has a Python dictionary object, a key-value store similar to a JSON object, called “MAPPING”. This dictionary’s format mirrors that of Shodan banners, and it contains two types of values: nested dictionaries and strings.

Dictionary values are for subsections of the Shodan banner through which to traverse in parallel with the MAPPING dictionary. String values indicate that data stored under the same key as that of the string value in the MAPPING dictionary should be added to the output dictionary, with the key being the value from the MAPPING dictionary and value being the value from the banner.

The parsing of a Shodan banner begins with an event being received, followed by the raw JSON being converted into a Python dictionary. Following this, the two dictionaries are iterated over, with common subsections (dictionary values) being recursively iterated into, and common values being added to the output event as described above.

As some values may not be suitable for addition without extra manipulation, there is a "CONVERSIONS" dictionary, for mapping a selection of keys to be inserted into the event to functions to be applied to the data. For instance, within IntelMQ, the "source.fqdn" field is expected to hold a value of a single string (as well as a valid FQDN). However, Shodan gives a list of strings, as one host may have many hostnames associated with it. As a workaround, the first hostname given is used.

A problem that comes up later and which I address is that the keys are only one level deep, meaning the field "domains" within a subsection of the banner would be subject to the same conversions as the field "domains" within the root object.

4.3 Automatically mapping the Shodan Stream schema

A problem with trying to understand the full format of any single section of data given by Shodan is that many parts of it are given piecemeal, depending on what data could or could not be gathered from the service. The simplest example of this comes from DNS servers. A DNS server can be configured to display or hide the hostname of the server it is running on, as well as the version of the DNS server software being used. Additionally, whether or not the service recursively resolves arbitrary domain names can be examined for any DNS server.

Given a number of banners for DNS services, some with neither, one or both of the optional data points included, a mapping with all would be desired. This is illustrated in diagram 1. Aside from getting a collection of keys to which values are mapped within the banners, the values themselves should also be collected. One reason is to be able to decide on the usefulness of the data. Another is to be able to record which values may be represented by different types in different banners, in order to be able to unify them to a common type, such as a floating-point number if floating-point numbers and integers appear under the same key, or a string if a string and any other type appear together.

Initially, an attempt was made to perform this mapping using the automatic schema discovery existing in Elasticsearch. However, due to issues with the data which are discussed later, this was scrapped in its early stages, and a custom method was used in its stead.

I used a three-stage process. The first stage is simply collecting a dataset of fresh banners into a plaintext file. The second is merging and deduplicating the data from the banners into a single object, alongside deduplicating the data. Finally, to give a concise representation of the schema, the single merged object is transformed into a Python Elasticsearch document class.

4.3.1 Collecting a dataset

The first part of the process is the simplest and least significant. A simple Python script which, using the official Shodan Python library, is used to write out each collected Shodan banner to a file with no modifications, one per line. This can be a time-consuming process, depending on the scope of the Stream being collected (defined alert subnets, countries or global). Later, some preprocessing steps, which are discussed later, were added to this step.

A total of 26GB of data was collected sporadically over the course of 3 months. In the beginning, only a small selection of subnets was checked, with all of Estonia being added later and, at the very end, accounting for the majority of the dataset, the global stream was used.

Later analysis showed that the size was inflated by some disproportionately large banners. For instance, each banner contains some raw data from the connection made, which was unnecessarily large in some cases. For instance, some servers were streaming public television channels as raw MPEG video data. The stream collection likely eventually reached some sort of size limit on Shodan's side. Similarly, when the index of a website serves an image directly rather than by embedding it in HTML, several kilobytes or even megabytes of compressed image data was featured in the "html" field.

For concrete figures, the top 1% of elements by size accounted for 25% of the total size of the dataset, and the top 4.2% of elements accounted for 50% of the size. The 10 smallest banners were in the range of 598 to 605 bytes, while the 10 largest were in the range of 3.1 to 7.2 megabytes. The bottom 50% of banners by size were 1836 bytes or less.

4.3.2 Merging the banners

For this process, the banners from the dataset file from before are iterated over and merged together recursively. Python's immutable data types are used to help with deduplication, as they can be included in set objects. Integers, floating-point numbers, strings, booleans and null elements are kept as-is since they are immutable, while "leaf" dictionaries are converted into immutable dictionaries (provided by the immutabledict 3rd-party Python library) and lists are transformed into tuples.

Here, "leaf" dictionaries refer to dictionaries which appear as values within lists of objects, as opposed to "trunk" dictionaries which are directly attributes of other dictionaries down to the root dictionary. The choice was made to not merge leaf dictionaries and at this stage and leave it for the next one, so as to preserve the relations between fields within a single dictionary for ease of analysis. Dictionaries containing the same exact data will, none the less, be deduplicated.

The reason for making the objects immutable was that then, sets could be used for value deduplication. Objects which can be mutated after creation may not be added to sets in Python, and hence would make data deduplication trickier or less performant.

The goal of this is to have a clear representation of all of the possible sections of data that could be represented within a single banner, similar to the MAPPING dictionary within the Shodan parser in IntelMQ. Deduplication helps with reducing noise from the data, as analysing the frequency of any configuration option or the like is not within the scope of this project. Preserving the possible values for any field helps with evaluating the usefulness of the data as well as to discover a variety of quirks within the data.

A recursive merging function is used, which iterates over both the already-merged "super-object" and the next object to be merged with it. Along with this, a second recursive function for converting mutable objects to immutable variants is used for some cases. Three different cases are considered for the merging function: the value is a single immutable value, a list of values, or a dictionary.

For single values of immutable types, they are all added together into Python set objects, which eliminates duplicate values.

Lists, are converted into tuples, with each of the list's elements being processed to be immutable. This is due to the fact that they may contain mutable objects, namely lists or objects, which would make every object containing them considered mutable as a whole, preventing the usage of sets for deduplication. For this purpose, the recursive immutability function is used to convert the list's elements to immutable types. The reason it is recursive is that the lists may contain "leaf" dictionaries, which may themselves contain other mutable objects.

Lastly, for dictionaries, the process is to merge the dictionary and its contained key-value pairs into the merged object, recursively calling the same merging function on the subdictionary. They are left mutable as, they're considered "trunk" dictionaries.

After the objects have been merged into one, this super-object is written out to a JSON file. Since the JSON format does not support sets or tuple types, those are converted into lists. This file then contains info on the full structure of the banner data format, along with all the possible values encountered throughout.

A diagram visualizing this process and the equivalent JSON objects matching those objects are given in appendices 3.1 and 3.2

4.3.3 Generating a template

For the final step of data processing, an Elasticsearch document class from the Elasticsearch DSL Python library is generated, based on the merged super-object. This is primarily, however, used to further help with manual analysis of the format for the use of improving the IntelMQ Shodan parser's MAPPING. This offers a simple view of the structure of the banners, as the possible values for each field stored in the results from the previous step account for gigabytes of data, compared to the kilobytes taken up by the template. Where needed, the previous results can be referred to for info on the possible values.

Some extra steps on the values are also performed at this stage, such as:

- enumerating what data types may appear together in a single field,
- enumerating fields which may contain null,
- enumerating fields in which the value can sometimes be a single value of some type, sometimes a lists of values of the same type, rather than containing a list of values even if the list would have less than two values,
- enumerating integer value fields which would overflow a signed 64-bit number,
- merging lists of dictionaries together,

The first three steps help to tag fields for which parsing may cause type safety issues unless properly handled, either explicitly with those fields or with a more robust scheme of parsing. Examples of potential issues would be type confusion in weakly and dynamically typed languages, such as PHP and JavaScript, when a value is not of the type it is expected to be, or a null pointer exception if a JSON parser in, for instance, Java, assigns the JSON null type to a Java null pointer and the field is accessed, with the expectation of it being of a non-null type.

The fourth step, checking for integer values beyond a signed 64-bit integer's range, is an example of another possible source of issues for parsing of the values. This is not an issue for IntelMQ, as the Python int class supports arbitrarily large integers, however parsers expecting integers to fit within a 64-bit signed integers would run into issues with this.

The merging of the dictionaries is useful as they typically have the same or similar format of data. For confirmation of this, they were deduplicated, but not merged, at the previous step.

To actually do all of this, yet another recursive function is used for converting the super-object into an Elasticsearch document class file. To simplify things, there is a helper class, "Nested", which is used for dictionary values.

For the template format, for each line, two main factors are considered: the types represented in a field, and whether the field can contain multiple values (such as a list of hostnames rather than one hostname). Some fields may also be tagged, in the form of appended comment strings after the field name, as well as with its path being placed in a list after the template.

The class comes with a custom __str__ method (used when converting an object into a string), which converts its contained data to text to the template's format, and the attributes "multi", "properties" and "level". "multi" specifies whether the value is a single object or a list of objects, "properties" is a dictionary containing the properties of the wrapped dictionary, and "level" is the indentation level the dictionary will be written at.

The recursive function is passed the root of the super-object, and it iterates through its keys and values. In case of dictionary values, indentation is incremented and they are recursed into, with a "Nested" element being returned. In the case of lists, a string is returned, with results varying depending on the values in the list.

First, a set of all types occurring within the list is calculated. From there, three cases are first considered. If there is a null value within the set of types, the field is marked as nullable. Second, if the set of types is just integers and floating-point numbers, the values are marked as such and set to be "upgraded" to floating-point numbers. If no float upgrade is performed and integers are included in the types, a check is performed to see if any of the integers are beyond the range of signed 64-bit integers.

Next, a return value is chosen, depending on the types contained within the input list. If only list elements exist within, a new list with all of the elements within the sublists is generated and the function calls itself recursively, with the field being marked as containing multiple values. If only dictionaries are contained, they are merged together and the function is called recursively on it. In case there are only two (non-null) types, with one of them being the list type, the type is marked as containing both single and multi-values and a type is taken based on the non-list type. For a single (non-null) type, the template type string matching that type is returned. Finally, for an empty set of (non-null) types, it is marked as a generic "object".

The result, based on this dataset and with a number of measures taken to reduce the end result's size, was a template with 3476 fields, including 88 directly in the root object, 567 containing objects/subsections, and 2910 containing other types of values. A snippet of the schema is given in appendix 4, and in template.py within the repository.

4.3.4 Issues identified in the schema

Having collected all of this data, a collection of issues with the automated processing of the format, making automated processing of the data an issue.

Some info on fields with type issues:

- 29 fields were found to be strings in some banners and numeric types (integer or float) in others,
- 28 fields were found to contain integer values in some banners and floating-point values in others,
- 5 fields were found to contain integers that would overflow a 64-bit signed integer,
- 65 fields were found to occasionally contain null values,
- 14 fields were found to occasionally contain subsections, occasionally some other value (usually null),
- 13 fields were found to have both single value types (e.g a string) and list value types (e.g list of strings),
- 11 sections were determined to be better served being converted from key-value pairs into lists of objects with the key being a field of the associated object

Several workarounds for type issues had to be implemented within the code. First, as a general issue, the same field holding different types of data can cause type safety issues in some setups (as was extrapolated in 4.2.3) unless special care is taken to handle this. The only solution to this is to not trust most fields to hold predictable types.

Aside from “leaf” fields only ever containing terminal values (integer, floating-point number, string null, list), some fields which usually contained subsections were occasionally found to be marked with null or even completely different types. Instances of possible null values are “ssl” → “cipher” and “ssl” → “dhparams”. An example of a completely different type is “redis” → “clients”, which can be a subsection with 4-6 attributes giving some metadata or a list of objects, each with 18-23 attributes detailing info on each client.

A number of subsections had an unfortunate format where one piece of information unique to the a discrete data point was set as the key, and the rest of the data was set as the value. An example of this is the “vulns” section directly under the root object. The info for one vulnerability exists in two pieces within one key-value pair: an identifier, such as a CVE ID like CVE-2013-1643, as the key, and a selection of other info on the vulnerability, such as the CVSS score, a short summary and references like security advisories and exploit PoCs, as the value. This splitting of values may significantly increases the number of fields perceived to exist within the Shodan banner, especially in cases where the value considered as the key is entirely service-configurable and likely to differ from host to host, such as the names of database tables and indices. This increase in perceived field count may cause issues in some setups, such as if the banners were to be directly fed into a NoSQL database with automatic schema discovery. In Elasticsearch, this significant increase in field names can cause a problem known as a mapping explosion, which significantly decreases the performance of the database or may even crash it. A solution to this for users of Shodan Stream would be to identify these problematic sections and handle them through means such as converting the dictionary’s key-value pairs into a list of the values with the data present in the key expressed within the value section it was assigned to. As an example, by converting the vulnerability identifier → vulnerability info mapping in “vulns” to a list of vulnerability info objects, each with a “identifier” attribute with the value coming from the key (e.g CVE-2013-1643).

At the very least the “ntp” and “rsync” sections consistently produced junk fields. For “ntp”, these were identifiable by their names, consisting of only digits and the string “refid”, or the string “refid” followed by numbers. For “rsync”, a variety of strings were identified as modules, with most of their names indicating they are greeting messages or chunks thereof. Examples of this are “This is stage.opensuse.org, restricted rsync server of openSUSE.org.” and “questions should be addressed to [...].“ (email address redacted). Besides this, some keys within banners were simply empty strings, which may cause errors or confusion.

Other small quirks that anyone making use of the data should also be wary of also exist. One example is the Redis client addresses being a list of two values, with the options being a hostname and port, or an absolute filesystem path and 0, or a question mark and zero. Another is that a number of integer fields contain values beyond the range of a 64-bit signed integer, which may be a reasonable type to assume to be able to fit integer values. Using unsigned 64-bit integers instead would cause issues in places where negative numbers may appear, and hence would not be an appropriate alternative. Widespread use of bigint types is also hampered by the existence of mixed integer/float valued fields.

For human parsing, the data expressed in a banner clear enough. However, all the issues above result in the Shodan banners being tricky to automatically parse on a large scale. Manually parsing the format in a comprehensive manner is near-impossible due to the quantity of and variance in the banner data, and automatically generated schemas would become massive off the top of the number of objects with automatically generated field/section names. They may also have issues with the field type quirks. As such, custom software, with iterative tweaking to handle any newly discovered issues, as was done here, appears to be a good solution. Due to the issues listed, the template contains lists of fields which are known to need different types of special handling. Instances of special handling would be type conversions, converting key-value mappings into lists of values, converting single elements into lists, potentially converting overflowing integers into strings, or explicitly avoiding null values for some fields.

The data within the template is very likely not conclusive and as these issues are pervasive, any parser would require highly robust handling of data issues and would work better with known data types.

In order to reduce noise in the output template, code for preprocessing the data to remove some of the above issues was implemented. This is visible in transformations.py in the repository.

4.4 Mapping the Shodan banner schema to the IntelMQ format

This is the first step which produces practical results. For this, the template and super-object can be analysed together to determine fields which IntelMQ users would benefit from having. In particular, this means:

- The data in the fields extracted from the banner should be useful in security-related analysis and data processing,
- The values and types thereof of the fields should be relatively stable

A description of how the Shodan parser works is given in section 4.1. There, a problem is stated: there is a method of performing specific conversions on some fields, though this only considers the key of the attribute within the deepest object it is at and not the path to it.

When recursively walking through the "MAPPING" object and the banner, the values in some fields may for some reason need to be converted in some way. As an example, the character sequence "+00" is appended to a timestamp in the root object in order to make it clear its timezone is UTC and it can be automatically converted into an IntelMQ-compatible timestamp. However, in order to perform this conversion, only the key of the field is considered, while the location of the field in relation to the rest of the banner is not, which could become problematic.

For instance, besides the "timestamp" attribute within the root object, there is also a timestamp attribute at "elasticsearch" → "cluster" → "timestamp". If both of them were to be included in the IntelMQ event and the latter field already had the "+00" suffix, the conversion would break that value. Similarly, if some field inside a subsection, e.g "section" → "subsection" → "foo" were to be converted, a comment explaining the location of "foo" would be necessary, as the full path would not be known from the field name "foo" alone.

In order to do this, each recursive call from within the recursive mapping application function keeps track of a sequence of key values, leading all the way to the current object being worked on. This is used as the key and the keys are joined together with dots as separators, leading to "timestamp" still being identified by "timestamp" and “section” → “subsection” → “foo” being identified by “section.subsection.foo” rather than “foo”.

After this, a selection of the mapping which matched the requirements on the data fields given above were added to the mapping. The results of these changes are visible within the git history of the IntelMQ Github, with a majority of the existing mappings (as of the time of writing) being provided by the results of this work. A link of these results from the time of writing are provided here.

The code used for generating the mapping, described above, is available in full here:

https://github.com/monoidic/shodan-schema

The results of the changes, as of the time of writing, are visible here.

https://github.com/certtools/intelmq/blame/
5c90ed4591b2f450b7761f32e71095c821fe7d35/intelmq/bots/parsers/shodan/parser.py

5 Filtering Shodan banners

Due to the far-reaching nature of Shodan's scans, much of what it captures may not be useful in identifying services of interest. Examples of what would make a service interesting are that it is (potentially) vulnerable, or that it is generally accepted not considered acceptable to be on the public internet for other reasons. A banner which does not match any of the filters would be discarded.

Using the mapping and data from before, some common issues with services which Shodan provides info on can be mapped out. As a starting point, the following 5 filters were considered:

- The scanned service is from an explicitly created set of services which should not be exposed to the internet (databases, ICS devices, remote control protocols like RDP or VNC etc.)
- The service was marked by Shodan as being a database, already compromised, or an ICS service
- The service provides open DNS resolution over UDP (potentially opening itself to being used in DNS-based DDoS amplification attacks)
- The service is marked with one or more vulnerabilities, with options to match any vulnerabilities or only those confirmed by Shodan to exist on the service
- The service's TLS certificate has expired

Though this list could be expanded, a more important consideration is to make the system performing this filtering expandable. That is, it should be simple to add new filtering methods or discard ones that are no longer relevant. Aside from that, any and all filtering methods matching the banner to should be marked in the output, as the reason for a banner having passed through the filter may become more difficult to distinguish as more filtering methods are implemented down the road.

5.1 Attempt one – specialized Shodan expert bot

My first approach was to introduce a new bot with the specific goal of performing filtering on IntelMQ events coming from Shodan. Considering the importance of configurability, the goal was that a developer could, with minimal effort, introduce a new filter.

For this, a class was designed which would have a number of class methods, each taking an IntelMQ event containing the Shodan banner data and returning a boolean value representing whether, according to that filtering method, the event should be passed on or not. These methods would represent the filtering methods described above. The relatively simple interface would make this easy to extend. To ensure a filtering method is checked, the method would have to be added to a list of methods, existing as an attribute of the list. For simplicity, filters that are always enabled could be enabled by adding the "@add_filter" decorator on the method. Finally, a method would, given an event, apply all given filters to the input event and return a list of strings representing the names of methods which were matched so they could be added to the event.

For the IntelMQ bot class, there would be a standard bot initialization method which would take care of configuring the filter with any admin-defined configuration options. Examples of where extra configurations could exist would be for enabling or disabling optional filters or, for instance, specifying the list of services that should be matched by the first filtering method described.

This bot was fully implemented[1]. However, a core developer pointed out[2] that this bot may not be very useful to IntelMQ in general due to the how specific to Shodan it is. An alternative tool for doing this was named: the Sieve expert bot. It was also mentioned, however, that some of what the Shodan expert bot implemented would not work without improvements to the Sieve expert bot.

Though there was some time put into this bot, it was clear that a more general solution to this filtering problem would prove far more beneficial than a solution only benefitting Shodan Stream.

1 https://github.com/certtools/intelmq/commit/ea7a1f75157cc8b89bb4ab109a0b418fb4fd9073
2 https://github.com/certtools/intelmq/pull/1835

5.2 Attempt two – Sieve bot DSL improvements

While I had not considered it earlier, IntelMQ offers a powerful bot: the Sieve bot. It implements a language reminiscent of the Sieve language, specified in RFC 5228, modified for use within IntelMQ. The language consists of match expressions and statements. Match expressions are checks on the data in the banner, which may be composed together using the logic operators && and || as well as parentheses. Statements are actions performed on the data, such as adding, overwriting or deleting key-value pairs from the event.

Improving this tool would more closely align with the goal of making an extensible tool. The end result would be that an existing fairly flexible tool being more useful for a variety of uses, including for Shodan Stream events, rather than that there is a new tool only useful for Shodan, and even then only after heavy changes to it.

Upon closer look, however, the DSL was missing some vital features. I would not even be able to implement any of the original 5 filtering methods described above without first extending the DSL. This would come in two main parts: matches and actions.

There were matches for string-valued fields (exact or regex match of the value), number-valued fields (comparison of numeric values) IP addresses (checks if an address is located within some range) and existential checks on the fields. The matches I described, however, used list- and boolean-valued fields. As such, matches for those fields would have to be implemented.

As for actions, there are some for manipulating string-based values into field, removing fields, and stopping further execution of the sieve file while still passing the event on or dropping it entirely. Actions for creating and appending to lists of values in fields of the event (and not just strings) would be useful. This would allow for strings identifying all filter methods which were applied to be collected into a field.

The DSL within the Sieve bot is implemented using the TextX Python library, and it exists in two parts. The first part is a grammar file describing the syntax of the DSL. Based on this, the sieve file used to configure the bot is parsed into an AST. This AST describes the structure of the code given in the user-written sieve file in a tree structure.

This AST is then walked through whenever an event is passed in, with conditional paths followed and actions taken in the code depending on the data.

In the grammar file, a total of 7 new operators were added, those being :equals, :setequals, :overlaps, :subsetof and :supersetof for list values, and == and != for booleans (== and != appear for other types as well). The types TypedValue and ListValue were also added, representing a single value of either a floating-point, integer, boolean or string value for the former, and a list of such values for the latter. These are used for all of the matches and actions besides the boolean matches.

The new list-based matches are:

- :equals, which compares for exact equality of all types and values in the list in the field and the one passed as an argument,
- :setequals tests for the set of all values, ignoring duplicates and order, between the field value and the argument being equal,
- :overlaps tests if there are any elements occurring both within the field value and the argument list,
- :subsetof tests if the field value contains a subset of the values within the argument,
- :supersetof tests if the field value contains a superset of the values within the argument

The boolean matches are comparatively simpler, with the operators == and != testing for equality and inequality between a boolean in a field value, with the possible arguments true and false.

The actions append and append! actions were added. The former appends to list values, creates new lists if the the the key is not defined in the event, and silently ignores non-list values. append! works similarly, but in the case of an existing non-list value, it replaces the field with a list of two values: the original value, and the appended value.

The add, add! and update actions were updated to be able to allow non-string values, namely booleans, integers and floating-point values, to be set in each banner.

Finally, some bigger changes to the syntax were implemented. First off, any expression (or collection thereof enclosed in brackets) can be negated by prepending it with an exclamation mark (!), allowing for more expressive rules than before and removing the need for negated variants of each operator, such as :contains and its negated :notcontains variant. This was accomplished through changes in the grammar file, adding an optional negation token to expressions, and negating the results of the expression if it exists.

Second, conditions (if/elif/else) and actions can be mixed more freely. Previously, the main scope of the script could either only contain actions, or conditionals which contain actions, but no nested conditions. Now, however, conditionals can be nested to arbitrary depths, and actions and conditions may appear in the same scope. This was done by categorizing both conditionals and actions as statements in the grammar file and modifying the control flow in the code to handle the changes in the AST.

With these changes implemented in the Sieve bot[1], it is, in turn, possible to use the 5 example filter methods within IntelMQ, as seen in appendix 5. More importantly, however, it is now much more capable of filtering non-string data, being able to match on numeric types (integers and floating-point numbers), strings, IP addresses, lists and booleans, more than the full range of types given within a JSON object. Complex rules, which would previously have required workarounds such as the use of multiple Sieve bots conditionally passing messages to each other, and rules which always match true in order to introduce invariant actions alongside variant actions, are now unnecessary as well.

1 https://github.com/certtools/intelmq/pull/1895, https://github.com/certtools/intelmq/pull/1923

6 Summary

I started off with one problem: the integration between Shodan Stream and IntelMQ was far from as good as it could be. From there, I divided the issue into two parts, and set out to make them as useful for issues not directly relating to the integration between Shodan and IntelMQ. This involved exploratory research into how best to process and describe JSON-based data structures with a variety of irregularities between different samples, and was in part a case study on how data filtering and Shodan data parsing is done in IntelMQ and how it could be improved, with this work being submitted to the upstream.

For both tasks, initial attempts were made at using different tools than what was eventually decided upon. For mapping the schema, using just the existing schema mapping functionality of Elasticsearch was deemed insufficient and ineffective. And for filtering the Shodan data flowing through IntelMQ, a specialized bot was deemed both ineffective, as to deliver on the requirements of flexibility, it would be no more than a template for a bot, to be modified actively, and modifying the bot itself could cause unexpected issues, such as if two or more are expected to have differing configurations.

Finally, better solutions to both issues were developed. A proof-of-concept solution for collecting feed data, processing it to receive a usable schema and analyse the data to determine fields with some inconsistencies. This was used to improve the Shodan parser within IntelMQ, as a practical usable example of its usefulness and a result of this work.

For filtering the data within IntelMQ specifically, a DSL within it capable of performing filtering duties was improved significantly, allowing for more functionality and ease of use, through better flow control capabilities.

A number of issues in the Shodan data format were documented, such as type safety or null values in unexpected places. To counter these issues, a number of preprocessing methods were also given, which remove or mitigate these issues.

In total, the IntelMQ Shodan parser’s support for data fields rose dramatically, with 58 fields supported before and 379 after this work, alongside an update from 3 to 26 defined protocols and some code quality changes.

This work could still, however, be continued. The 26GB dataset collected over the course of this work likely did not contain the full range of the schema, and it could also be updated as Shodan adds support for new services or returns more data on existing ones in the future. Hence, a piece of software continuously taking data from the feed to update a generated schema, based on this work, could be made. Based on the schema, work could also be done to improve other software to make more/better use of the data received from Shodan.

Besides that, as this work was in part explorative, not much care was put into optimization. Storing all of the data in data structures within memory was convenient for testing new ideas, though increasingly memory-expensive as more data was collected. A solution offloading storage to a database, while taking care to handle issues in the data similar to those discovered in this work, would likely be more performant and less resource-expensive.

References

[1] S. Wagner and B. Schacht, IntelMQ Shodan parser. https://github.com/certtools/intelmq/blob/2255206ffd2f8559983fc5f4ae96c8a867216915/intelmq/bots/parsers/shodan/parser.py . Accessed: 2021-05-01

[2] Various contributors, IntelMQ Sieve expert. https://github.com/certtools/intelmq/tree/9d224f90debc08fec705691d65fc59940d18daa1/intelmq/bots/experts/sieve . Accessed: 2021-05-01

[3] S. Wagner, README of IntelMQ. https://github.com/certtools/intelmq/blob/82b174fe6c5587a17c0333c7e984770c9e8507c5/README.rst . Accessed: 2021-05-01

[4] European Union Agency for Cybersecurity, "Incident Handling Automation". https://www.enisa.europa.eu/topics/csirt-cert-services/community-projects/incident-handling-automation . Accessed: 2021-05-01

[5] CSIRT-KIT website. https://csirt-kit.org/ . Accessed: 2021-05-01

[6] Shodan Inc, "API Reference – Clients/Libraries" https://developer.shodan.io/api/clients . Accessed: 2021-05-01

[7] F. Hellmann, P. Hellmann, "Implications of vulnerable internetconnected smart home devices", BsC thesis, Faculty of Computing, Blekinge Institute of Technology, Karlskrona, Sweden, 2017

[8] H. Wang & Rootkiter, "BCMPUPnP_Hunter: A 100k Botnet Turns Home Routers to Email Spammers", https://blog.netlab.360.com/bcmpupnp_hunter-a-100k-botnet-turns-home-routers-to-email-spammers-en/ . Accessed: 2021-05-01

[9] S. Yerushalmi & O. Shaty, "CrimeOps of the KashmirBlack Botnet – Part II", https://www.imperva.com/blog/crimeops-of-the-kashmirblack-botnet-part-ii/ . Accessed: 2021-05-01

[10] R. C. Bodenheim, "Impact of the Shodan Computer Search Engine on Internet-facing Industrial Control System Devices", thesis, Dept. of Electrical and Computer Engineering, Air Force Institute of Technology, Ohio, USA, 2014

Appendix 1 – Non-exclusive licence for reproduction and publication of a graduation thesis[1]

1 The non-exclusive licence is not valid during the validity of access restriction indicated in the student's application for restriction on access to the graduation thesis that has been signed by the school's dean, except in case of the university's right to reproduce the thesis for preservation purposes only. If a graduation thesis is based on the joint creative activity of two or more persons and the co-author(s) has/have not granted, by the set deadline, the student defending his/her graduation thesis consent to reproduce and publish the graduation thesis in compliance with clauses 1.1 and 1.2 of the non-exclusive licence, the non-exclusive license shall not be valid for the period.

Appendix 2 – Data flow graph

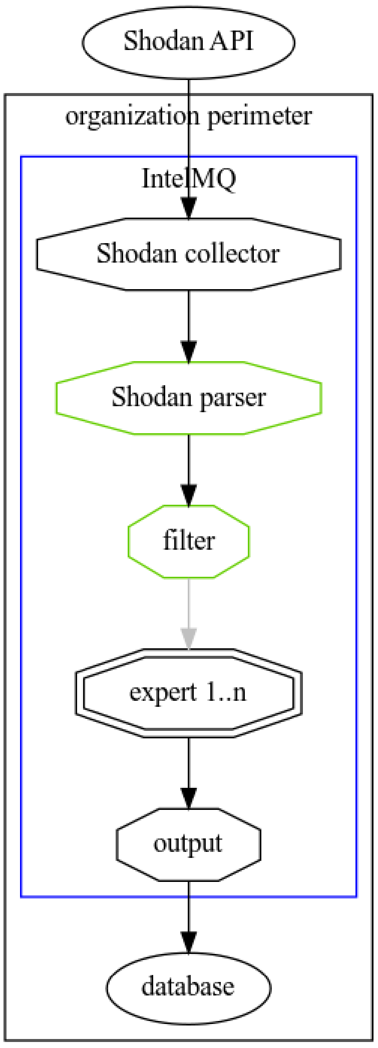

Figure 1: A visualization of the flow of data from Shodan through IntelMQ

Appendix 3.1 – Figure with an example of Shodan banner data collection

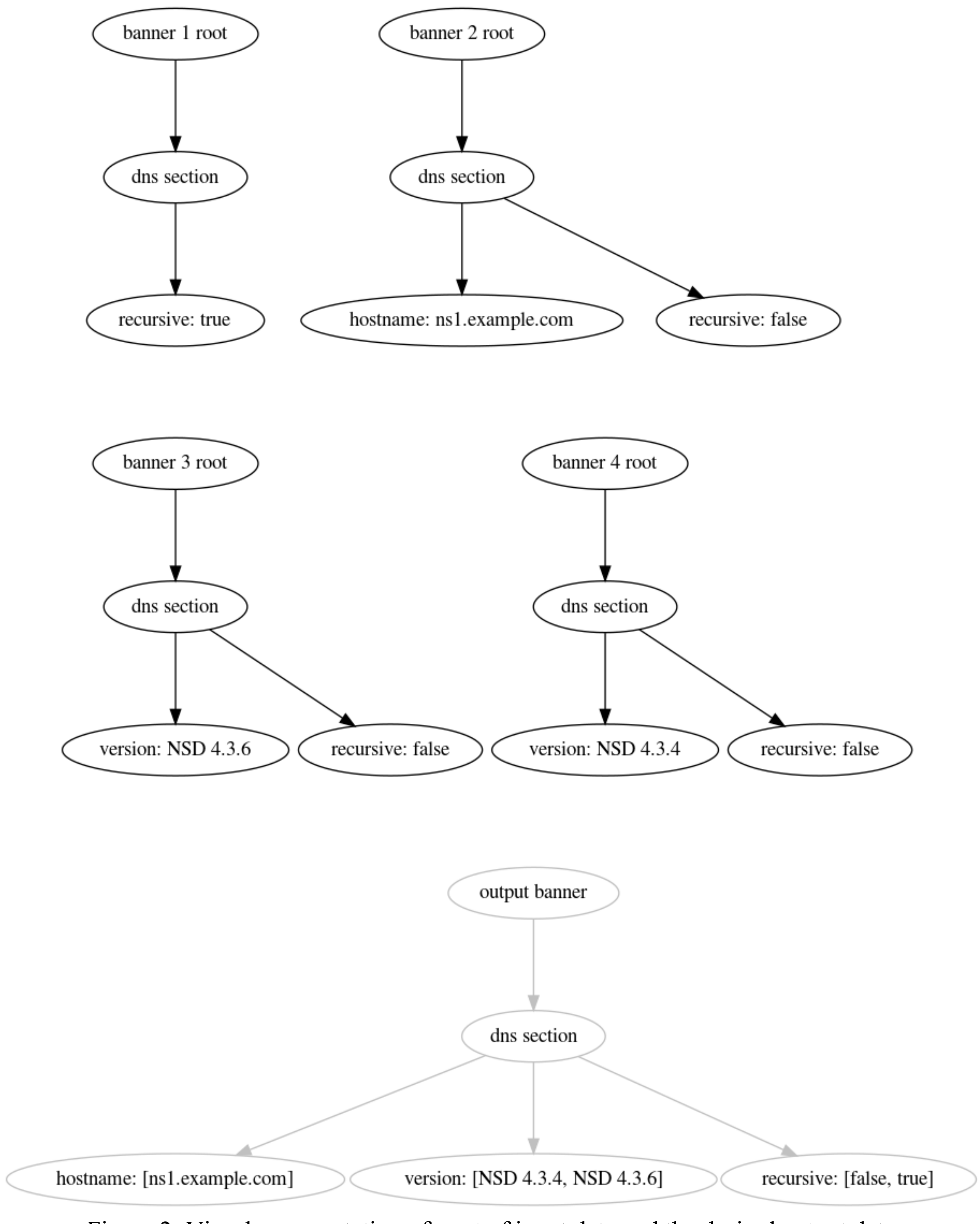

Figure 2: Visual representation of a set of input data and the desired output data

Appendix 3.2 – Textual example of Shodan banner schema collection

```
## Inputs
{
      “dns”: {
            “recursive”: true
      }
}
{
      “dns”: {
            “hostname”: “ns1.example.com”,
            “recursive”: false
      }
}
{
      “dns”: {
            “version”: “NSD 4.3.6”,
            “recursive”: false
      }
}
{
      “dns”: {
            “version”: “NSD 4.3.4”,
            “recursive”: false
      }
}

## Output
{
      “dns”: {
            “hostname”: [“ns1.example.com”],
            “version”: [“NSD 4.3.4”, “NSD 4.3.6”],
            “recursive”: [false, true]
      }
}
```

Appendix 4 – Sieve ruleset example

```
if extra.shodan.unique_keys :overlaps [
    'coap', 'elastic', 'etcd', 'ethernetip', 'influxdb', 'iscsi',
'msrpc',
    'mssql', 'netbios', 'redis', 'rsync', 'smb', 'snmp',
'unitronics_pcom', 'upnp'
] {
    append extra.shodan_filter.matched_filters 'has_filtered_section'
}

if extra.tags :overlaps ['database', 'compromised', 'ics'] {
    append extra.shodan_filter.matched_filters 'has_filtered_tag'
}

if extra.dns.recursive == true && protocol.transport == 'udp' {
    append extra.shodan_filter.matched_filters 'open_recursive_dns'
}

if :exists extra.verified_vulns && ! extra.verified_vulns :equals [] {
    append extra.shodan_filter.matched_filters 'has_verified_vulns'
}

if :exists extra.vulns && ! extra.vulns :equals [] {
    append extra.shodan_filter.matched_filters 'has_vulns'
}

if extra.ssl.cert.expired == true {
    append extra.shodan_filter.matched_filters 'expired_cert'
}
```

Appendix 5 – Snippets of the schema

```
    ripple = Nested(properties={
        "peers": Nested(multi=True, properties={
            "ip": Text(),
            "public_key": Text(),
            "type": Text(),
            "uptime": Long(),
            "version": Text(),
            "complete_ledgers": Text(),
            "port": Text(), #multiple types: int, str
        })
    })
...
    snmp = Nested(properties={
        "contact": Text(), #nullable,
        "description": Text(), #nullable,
        "location": Text(), #nullable,
        "name": Text(), #nullable,
        "objectid": Text(), #nullable,
        "ordescr": Text(),
        "orid": Text(),
        "orindex": Text(),
        "orlastchange": Text(),
        "oruptime": Text(),
        "services": Text(), #nullable,
        "uptime": Text(), #nullable
    })
```