Sagishi: an undercover software agent infiltrating IoT botnets

Andrea Oliveri, Filippo Lauria
{firstname}.{lastname}@iit.cnr.it
Institute for Informatics and Telematics, Italian National Research Council
via G. Moruzzi, 1 - 56124 Pisa, Italy

Abstract
IoT devices are continuously proliferating, in fact, by 2030, up to 125 billion devices will be connected to the Internet. [1] Strictly related to the proliferation of IoT devices there is the proliferation of IoT malware and, in particular, IoT botnets. Mapping and classifying IoT bots forming part of IoT botnets and how these propagate over the Internet is a challenging task. Fooresec [2] aimed to address this issue by footprinting, reporting and remotely securing IoT devices that had been previously turned into bots. Its activities were based on honeypots, which allowed to "sense" the Internet in order to *solely* map and classify IoT bots, according to their behaviors.

In order to have a more detailed comprehension of the aforementioned phenomenon, other elements of a botnet, apart from bots, could be taken into account: malware samples collected by honeypots contain relevant pieces of information about the global structure of the botnet, e.g. *Command and Control server*s (*CNC server*s or *CNC*s) hostnames and/or IP addresses, relevant TCP/UDP ports[1], etc. These *features* of the botnet can be used by *a software agent* for infiltrating the botnet, then collecting pieces of information otherwise unavailable. In fact, this agent, which is capable of emulating the behaviour of a genuine infected host, can fool the CNC and discover commands issued by the botnet master.

A software architecture with the aforementioned characteristics is illustrated by this article. The most important part of this architecture is Sagishi (swindler, in Japanese) which is the software agent presented earlier and implemented in our working prototype of the described architecture. This has been used to collect data in the period from May 1, 2018 to June 18, 2018 (49 days). In order to indicate the validity of the approach, some results, produced from the collected data, are also presented.

[1] from now on, for the sake of simplicity, we will refer to these by using *address*es *and port*s.

Key ideas

As we have said in the previous section, the aim of this work is to increase our knowledge on the global behaviour of a given botnet. In order to achieve this, not only do we need more details concerning *bot clouds* but it is also essential to have thorough information involving CNC servers, which are responsible for coordinating bots forming part of their respective botnet.

Traditionally, a honeypot is a piece of software[2] that emulates a non-infected Internet node, which publicly exposes some network services (e.g TELNET, SSH, SMTP, etc.), and passively waits for connections incoming from any alleged remote attacker. An attacker, which assumes to be connected to a real victim system, interacts with the honeypot and tries to *exploit it*, e.g. it executes commands, uploads files (which might be *malware samples*[3], too), starts malicious processes, etc. Its final goal is to compromise the network node and make it join a given botnet. Within the scope of this work, we refer to this *traditional* type of honeypots as *Passive Honeypot*.

Clearly, if we consider a botnet as a one-level tree [a], a Passive Honeypot can only interact with the leaf nodes of the botnets, i.e. bots. Instead, what we need is to gather information directly from the root nodes, i.e. CNC servers. For this purpose, we introduce *Active Honeypot*s. By closely emulating an infected node, this new class of honeypots attempts to actively establish a connection with a CNC server, in order to fool it and obtain pieces of information available only to genuinely infected nodes.

Of course, the behaviour of a bot is not known a priori. It could be deduced by analyzing, i.e. reversing, its relative malware sample gathered by a Passive Honeypot. In other words, malware samples have to be analyzed[4] in order to recreate their internal structures and intended behaviours. In particular, the analysis of a malware sample should be focused on retracing harmless behaviour (e.g. communicate with the CNC, etc.), leaving aside harmful behaviour (e.g. infect other devices, attack given targets, etc.), since the main thing we are interested in is to exactly emulate the communication protocol used by the bot and the CNC. In any case, it is necessary to extract specific pieces of information from each new gathered malware sample. We refer to these as *botnet features*. For example, some common botnet features could be the address of the CNC, the port used by bots to connect back to the CNC, etc.

It also has to be noticed that botnet features are unique for each botnet. In particular, botnet features like the address of the CNC and the port used by bots to connect to the CNC are needed by the Active Honeypot to contact the CNC associated with the sample from which they have been extracted. When the analyzed sample does not belong to a known malware family or, in some cases, even when it does belong to a known one, it is very likely that botnet features cannot be extracted just by using simple tools (such as the *NIX command “strings”) or techniques, because malicious developers tend to encrypt and hide them. Therefore the analysis, i.e. the reverse engineering, of this kind of samples has to be performed by human analysts.

---

[2] e.g. a simple script, a more sophisticated application, a complete OS, etc.

[3] for the purpose of this document a malware sample is an executable file capable of turning the infected device into a bot forming part of a given botnet.

[4] in general, different malware samples can belong to the same malware family, i.e. samples created by extending or little modifying the code of a common parent. Samples belonging to the same malware family tend to maintain the same communication protocol, mainly because of the complexity and the high design cost of an always new and coherent communication protocol. As a consequence, the reverse engineering process, for this kind of samples, could be easier.

Only when the reverse engineering has been completed, it is possible to *automate* the botnet features extraction. For this purpose, the proposed architecture includes *the Parser*, which is a module composed of many *Parsing units*. Each Parsing unit is related to a previously analyzed and identified malware family. The Parser receives samples from the Passive Honeypot and tries to classify them. The classification procedure identifies the malware family of the processed sample. If a Parsing unit has been implemented for the detected malware family, then it extracts botnet features from the malware sample. Once the botnet features have been extracted, they are passed to the Active Honeypot which starts infiltrating the botnet.

All the attempts of attack detected by the Passive Honeypot, the features extracted by the Parser and the data exfiltrated by the Active Honeypot are stored in STIX[5] Objects.

The proposed architecture

As it could be noticed in Diagram 1, the proposed architecture has a modular structure that can be logically divided in four autonomous modules.

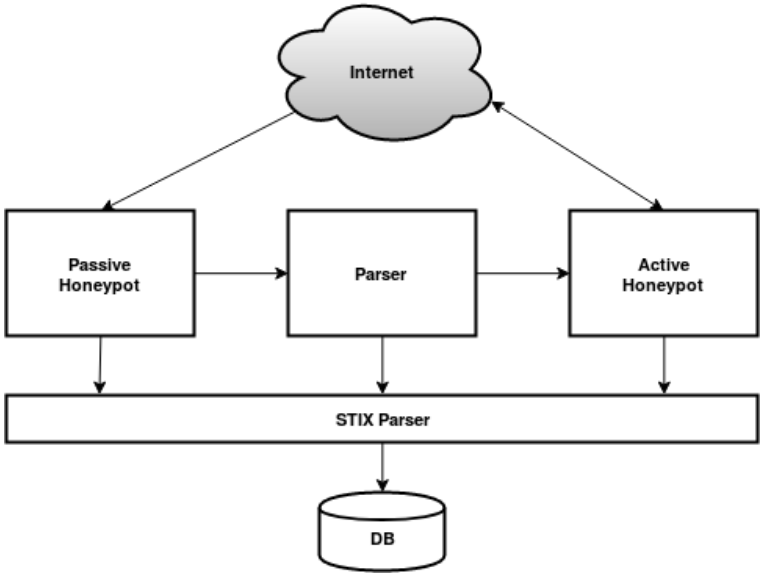

Diagram 1: an overview of the proposed architecture

Each module is responsible for a specific task:

- we have already described what a **Passive Honeypot** is. In the proposed architecture, this module is responsible for:
    - collecting data and malware samples uploaded by the attackers;
    - tracking malicious activities remotely performed by the attackers. In this perspective, an important botnet feature that can be extracted from the Passive Honeypot is the *entire command line* used by the attacker to execute the malware sample.

  All the actions performed by the same attacker on the honeypot constitutes an *attack session*;
- the **Parser** is a module responsible for analyzing malware samples collected by the Passive Honeypot. It uses some static binary analysis (SBA) techniques, for the purpose of producing reports which are then passed to the Active Honeypot;
- we have already described what an **Active Honeypot** is. Under the perspective of the proposed architecture, it is a module which uses some pieces of

[5] STIX (Structured Threat Information eXpression) is a standardized language which allows to serialize data about infosec threats, by using JSON, in order to make it easier to understand and exchange.

information contained in the reports generated by the Parser, with the aim of connecting to CNCs and collect attack/control commands issued to genuine bots;
- the **STIX Parser** is a module which parses data derived from the other modules. Moreover, this module stores parsed data into a database. Our goal is to share collected data with other organizations, using STIX 2.

Focus on the Parser behaviour

In this section it is described, in detail, the execution flow of the Parser. Basically, this module receives structured records from the Passive Honeypot. Each record contains the malware sample uploaded by the attacker, along with other useful pieces of information coming from the attack session, e.g. the name used by the attacker to save the sample, the exact command issued by the attacker to execute the malware sample, etc. Once the module has completed its processing activities, it produces two types of output:

- a detailed report containing all the botnet features extracted from the analyzed sample. This report is then stored by the STIX Parser into the database;
- a summarized report used as input for the Active Honeypot.

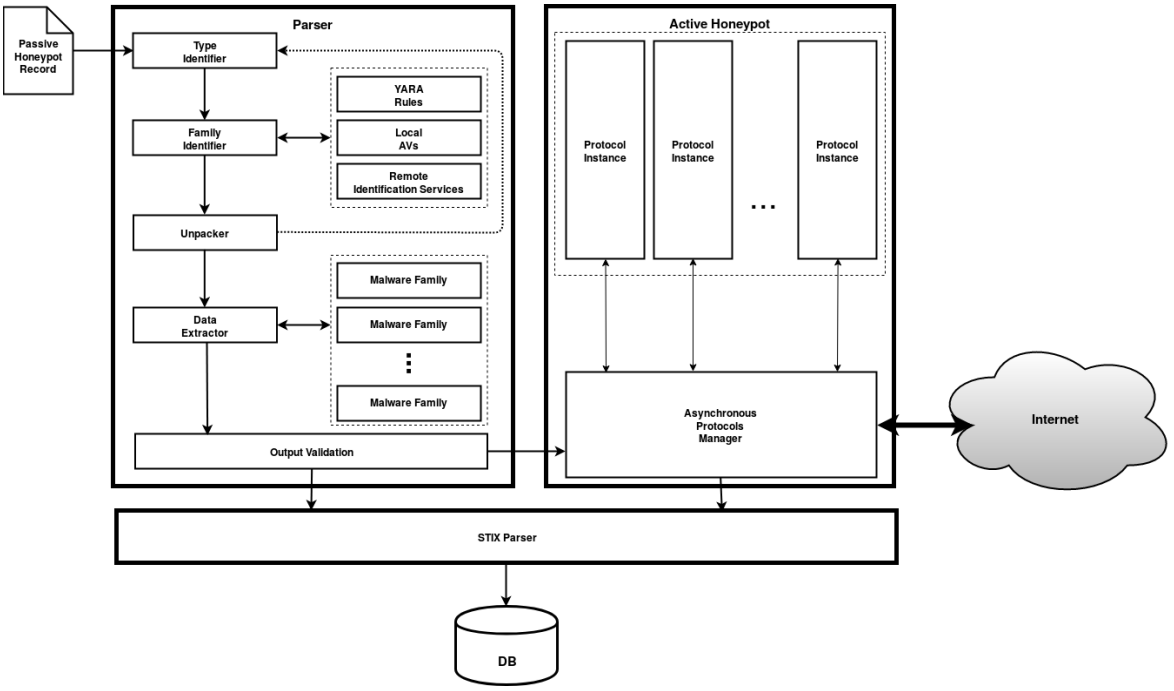

Diagram 2: a detailed view of both the Parser and the Active Honeypot units

As shown in Diagram 2, the module is composed by many units each of which performs a specific task:

- the **Type Identifier** is a unit which allows to determine the file type[6] (e.g. an executable, a text file, a XML file, a shell script, etc.) of each malware sample uploaded by the attacker on the Passive Honeypot;

[6] file types are determined by using a series of regex expressions along with some standard tools which interpret *magic bytes* at the beginning of a file.

- the **Family Identifier** is a unit which tries to detect if the analyzed sample is actually a malicious file. In that case, it also attempts to identify the malware family to which the sample belongs. This unit relies on the following components:
    - the subunit named **YARA Rules** tries to identify the sample by using a local set of YARA rules [b] of already known samples;
    - the subunit named **Local Antivirus** leverages local antivirus installations in order to identify the sample;
    - the subunit named **Remote Identification Services** is used as a last resort. It sends the signature of the sample (or the sample itself) to third party services specialized in sample recognition, e.g. VirusTotal [c].
- the **Unpacker** is a unit that checks if the sample is packed with a known packer[7] and tries to extract the unpacked content. If the sample is not packed or if it is packed with an unknown packer, the analysis continues with the next unit (Data Extractor). Otherwise, if this unit is able to unpack the sample, the unpacked content is processed again, as a new sample, by the Type Identifier;
- the unit named **Data Extractor** is responsible for calling every single subunit dedicated to malware family detection (MFDEs) that could be applied to the processed sample. Each MFDE is called according to the classification performed by previous units. Moreover, multiple MFDEs can be applied to a single malware sample;
    - a subunit dedicated to detect the malware family of a given sample is called **Malware Family Data Extractor (MFDE)**. In other words, each MFDE (one per malware family) is an independent subunit which tries to extract, from the sample, some known features, by leveraging SBA techniques. In particular, MFDEs try to extract pieces of information which will be then used by the Active Honeypot to infiltrate the botnet, e.g. the address of the CNC, the port to connect to, *handshake parameters* [8], etc.;
- the **Output Validation** is a unit which tries to validate the address and the port of the CNC discovered by previous units. The validation process is simple: the unit tries to connect to the CNC and waits for replies. If a reply from the CNC is sent back, the unit passes the necessary pieces of information to the Active Honeypot which will start infiltrating the botnet. In any case, a copy of the results obtained by previous units is passed to the STIX Parser which elaborates and stores it in STIX objects.

[7] e.g. simple compression algorithms like ZIP, or with executable packers like UPX [f], etc.
[8] these parameters are sent, in the very early stages of a bot/CNC communication, by the bot to the CNC server, in order to prove that it is a genuine bot.

Focus on the Active Honeypot behaviour

As in the case of the Parser, the Active Honeypot can be divided, as shown in Diagram 2, in the following logical units:

- different Protocol Instances. A **Protocol Instance** is a unit dedicated to accurately emulate a genuine bot of a given malware family. Each Protocol Instance maintains a connection with a given CNC (e.g. using keep-alive mechanisms, etc.), forging packets to send to the CNC and/or parsing packets received from the CNC, with the aim of gathering issued commands;
- the **Asynchronous Protocol Manager** is the unit which controls the entire module. It is implemented using the asynchronous programming paradigm [g], in order to efficiently manage local and remote connections.
  In particular, it is responsible for:
  - starting new Protocol Instances, one per each CNC server received from the Parser;
  - reconnecting/restarting each Protocol Instance in case of network failures or disconnections;
  - receiving/sending every data packet from/to the Protocol Instances;
  - forwarding produced results to the STIX Parser.

Prototype implementation

In order to reduce the complexity of the architecture of our prototype, automation mechanisms between modules and units have not been implemented. This implies that the process requires a manual intervention of the analyst who has to check and validate the output of both the Passive Honeypot and the Parser, in order to discard false positives (e.g. non-malicious server addresses mistaken for CNC addresses, etc.).
Furthermore, we focused only on malware samples of the MIRAI family [5] and their relative CNCs, which is appeared on the Internet for the first time in 2016 [3]. This choice is due to:

- the relative abundance of malware samples of the MIRAI family in our collection (98.3%);
- the little changes in the internal structure of those samples: even though it has been years since the MIRAI source code has been released [4], cyber-criminals still have not modified its encryption function nor the CNC communication protocol.

For what has been said so far, both the Parser and the Active Honeypot implemented in our prototype (both coded in Python 3) are able to extract and contact only CNCs that belong to the MIRAI family. Nevertheless they are developed with a modular structure: it is possible to insert a new MFDE/Protocol Instance for the new malware family, by merely loading a Python module.

The Cowrie Project [d], a medium interaction honeypot which allow us to emulate both a Secure Shell (SSH) and a Telnet server, has been chosen as Passive Honeypot. We have extended it by writing an output module for parsing session commands and producing related STIX objects. A Remote Identification Service, i.e. VirusTotal with the AVClasser post-elaboration [c], has been used as Family Identifier.

No simplifying assumptions have been made to implement the Active Honeypot (i.e. it has been implemented in a complete way). This implies that the Active Honeypot can:

- receive new MIRAI CNC addresses to contact (dynamically via UNIX socket or statically via its configuration file);
- manage sessions;
- produce a JSON output of commands received from the CNC.

Results

The following tables and charts have been produced with data gathered by our prototype in the period from May 1, 2018 to June 18, 2018 (49 days). From the collected malware samples, we have extracted hostnames of 19 different MIRAI CNC servers. We have found that only 4 CNCs were online and active, while the others were unreachable or not exposing network services[9].

Label	CNC hostname:port	Activity Period	Command Count	Unique Targets
S1	cnc.hackurbotnet.cf:27	9/5 - 25/5 (17 days)	1351	464
S2	q39485gnq8349y598gdr7ytoyg5.stream:23	3 /5 - 17/6 (43 days)	139	38
S3	0day.life:50000	9 /5 - 12/5 (4 days)	40	24
S4	serversrus.club:23	8/5 - 10/5 (3 days)	6	6

Table 1: online and active CNCs detected in the observation period

In Table 1 it is shown a summary of the captured data. In particular, we have detected diverse behaviours: 3 CNCs, S1, S3 and S4, have concentrated their malicious activities in a relative small time window, while S2 has spread its activity in a larger time frame. It also has to be noticed that S2 had long periods in which the *CNC service,* normally exposed on port 23, could not be reachable[10].

Focus on the CNC labeled S1

We have particularly focused on the CNC labeled S1, because, as evidenced in Table 1, it has reported a statistically significant number of records which allowed us to carry out a deeper analysis. As it could be noticed in Chart 1, this botnet has been capable of using different attack vectors[11] to perform DDoS attacks. The most used attack vector was UDP (36.6% of the total of the attacks), followed by GRE IP (34.0%), GRE ETH (17.7%) and DNS Water Torture (5.0%).

From our perspective, it is worth mentioning both GRE [7] and STOMP [8] protocols: in our dataset, GRE is the second most used attack vector after UDP, while STOMP has the highest payload size. In general, these protocols are known to be not blocked on perimetral firewalls, [9], [10] which possibly constitutes the most desired scenario for DDoS attackers.

[9] management services like TELNET or SSH excluded.

[10] we do not know why the CNC service (in a Malware-as-a-service perspective) was occasionally down. A possible explanation is related to a bug which we have discovered into the library *go-shellwords* [e] used by the original CNC software included into the MIRAI source code: when a *customer* send some non alphanumeric chars (e.g. "<") as input to the CNC service, it crashes.

[11] TCP SYN and ACK, UDP, HTTP, GRE and STOMP packet floods. Morevore, this kind of botnet is able to perform DNS Water Torture attacks [11], too. [4], [7]

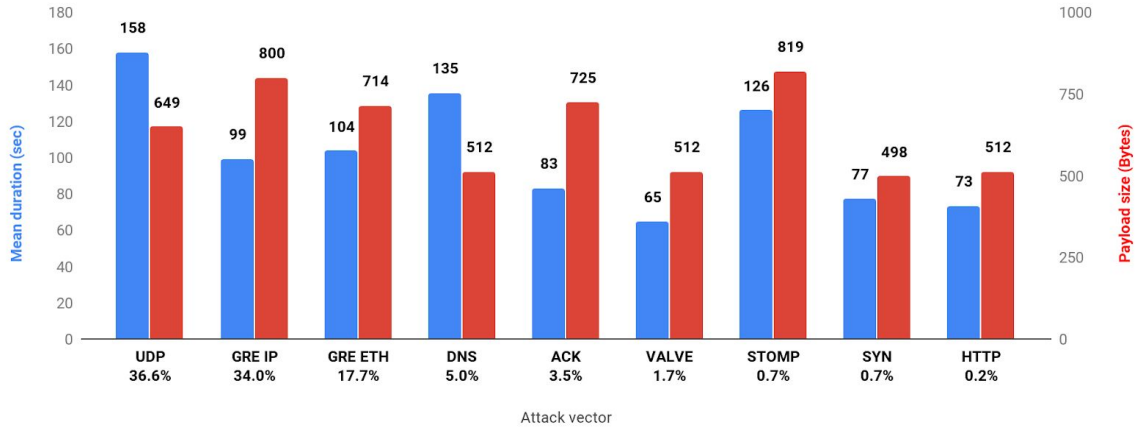

Chart 1: mean duration and payload size of attack vectors used by S1, ordered by number of occurrences (%) in the dataset

As it could be seen from the MIRAI bot and CNC source code [4], there are some hardcoded constraints and default values related to data shown in Chart 1:

- an attack cannot last more than 3600 seconds;
- a payload size cannot be greater than the packet maximum payload size which depends on the protocol used as attack vector. Furthermore, if the attacker does not indicate a payload size, the default one, i.e. 512 bytes, is used by bots.

The original MIRAI communication protocol allows to set every single flag in the IP and TCP headers of packets used to perform DDoS attacks. For example, fine-tuning those flags can allow to bypass blocking rules on stateless/stateful firewalls [12]. However, in our dataset, neither IP or TCP flags have been found to be set.

During the aforementioned observation period, Sagishi has been able to detect 464 unique DDoS targets, involving 149 targeted Autonomous Systems in 39 countries. For the sake of simplicity, in Chart 2 it is shown the number of unique targets for the top 10 most attacked countries.

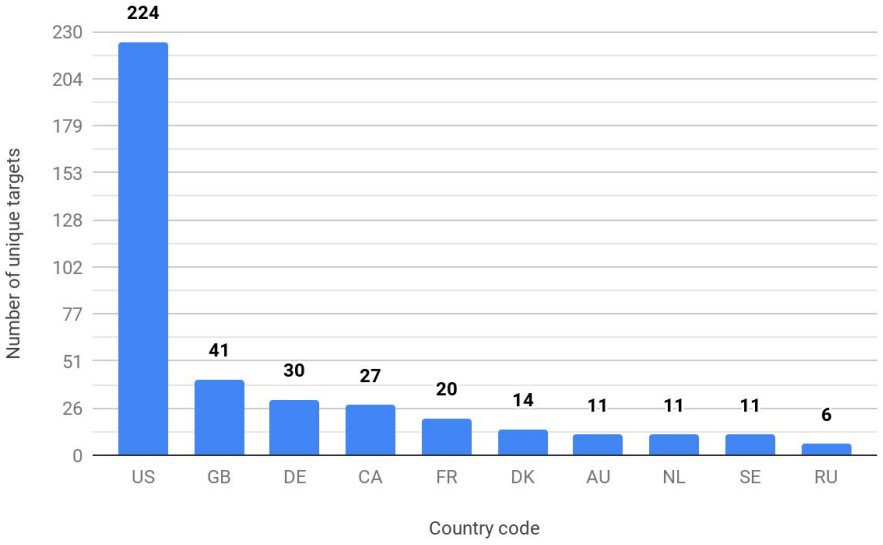

Chart 2: number of unique targets, grouped by country code (top 10)

We have performed WHOIS lookups of each single targeted address, in order to obtain the related AS Number and Country. Moreover, according to targeted network protocols/ports and the core business of the AS Owner, we have deduced, for each AS, the main type of service susceptible to DDoS attacks.

In Table 2 we have reported the top 5 attacked Autonomous Systems, ordered by the number of detected DDoS attacks. Furthermore, for each AS it has been reported the most used attack vector and the main type of attacked service deduced as previously described.

AS Number	Country Code	Main type of service	Number of attacks	Attack vector
3XXX0	US	Gaming Servers	539	UDP
1XXX6	FR	Hosting	109	GRE IP
1XXX6	US	Gaming Servers	58	UDP
2XXX3	US	Hosting	46	GRE IP
1XXX9	US	Hosting	43	GRE IP

Table 2: top 5 attacked Autonomous Systems

As it could be noticed, the most attacked targets are companies which own or host game servers. In these cases, besides WHOIS lookups, there is another indicator which validates the aforementioned deductions: the attack destination ports, indicated by S1 to bots. In fact, detected UDP ports are contained in the interval [27000, 27040]. Those ports are known to be used by servers of various online games [13]. Also, this particular distribution of the targets (i.e. ~44% against game servers), supports the hypothesis of a *DDoS-as-a-Service* dedicated to gaming communities.

Conclusions

In this work we have firstly introduced the concept of *Active Honeypot* which allow us, emulating a genuine bot, to exfiltrate information usually intended only for infected machines. Secondly, we have described how the Active Honeypot is integrated into a proposed software architecture capable of receiving malware samples (directly from a *Passive Honeypot*) and extracting *botnet features* with the aim of infiltrating botnets, by using Sagishi. Finally, we have implemented a working prototype of the proposed architecture in order to demonstrate its validity by presenting some results that lead us to discover a DDoS-as-a-Service botnet mainly dedicated to gaming communities.

Future works shall include:

- a complete automation of the prototype;
- the implementation of a larger number of supported malware families;
- the addition of new capabilities to MFDEs, e.g. dynamic binary analysis (DBA) techniques for botnet features extraction, etc.;
- an early warning mechanism to signal imminent DDoS attacks directly to alleged targets.

References

1. 'Internet of Things: a movement, not a market'. IHS Markit. Accessed July 2018.
2. F. Lauria. ‘How to footprint, report and remotely secure compromised IoT devices’. Network Security, Volume 2017, Issue 12.
3. 'Akamai's [state of the Internet] / security - Spring 2018 report'. Akamai Technologies, Inc. Accessed July 2018.
4. ‘Leaked Mirai Source Code for Research/IoC Development Purposes’. Accessed March 2018. www.github.com/jgamblin/Mirai-Source-Code.
5. Kambourakis G. et al. ‘The Mirai botnet and the IoT Zombie Armies’. MILCOM 2017, IEEE. October 2017. Accessed May 2018. https://ieeexplore.ieee.org/document/8170867/
6. Jain, Hemant. ‘Mirai Botnet: Protect Your Infrastructure with FortiDDoS’. Fortinet, Inc. October 2016. Accessed February 2018. fortinet.com/blog/industry-trends/mirai-botnet-protect-your-infrastructure-with-fortiddos.html
7. Farinacci, D. et al. ‘GRE Protocol Specifications’. March 2000. Accessed July 2018. https://tools.ietf.org/html/rfc2784
8. ‘STOMP Protocol Specification, Version 1.2’. Accessed July 2018. https://stomp.github.io/stomp-specification-1.2.html
9. ‘Identifying and Mitigating Exploitation of the GRE Decapsulation Vulnerability’. Cisco Systems Inc. September 2006. Accessed July 2018. www.cisco.com/c/en/us/support/docs/cmb/cisco-amb-20060912-gre.html
10. Bekerman, Dima. Breslaw, Dan. ‘How Mirai Uses STOMP Protocol to Launch DDoS Attacks’. November 2016. Accessed June 2018. www.incapsula.com/blog/mirai-stomp-protocol-ddos.html
11. ‘Whitepaper: DNS Reflection, Amplification, & DNS Water-torture’. Akamai Technologies, Inc. Accessed July 2018. www.akamai.com/de/de/multimedia/documents/technical-publication/dns-reflection-vs-dns-mirai-technical-publication.pdf
12. ‘Firewall TCP Established Rule Bypass’. Rapid7 LLC. Accessed July 2018. www.rapid7.com/db/vulnerabilities/generic-firewall-tcp-established-bypass
13. SG TCP/IP Port Database.Accessed July 2018. www.speedguide.net/port.php?port=27000

Resources

a. ‘Tree (data structure)’. Wikipedia.
b. ‘YARA: Simple and Effective Way of Dissecting Malware’. https://resources.infosecinstitute.com/yara-simple-effective-way-dissecting-malware
c. VirusTotal Project Homepage. www.virustotal.com
d. Cowrie Project Homepage. Accessed February 2018. www.github.com/micheloosterhof/cowrie
e. go-shellwords Project Homepage. Accessed March 2018. www.github.com/mattn/go-shellwords
f. UPX: the Ultimate Packer for eXecutables. Accessed March 2018. https://upx.github.io
g. Asynchronous I/O Python 3.x Native Library. Accessed March 2018. https://docs.python.org/3/library/asyncio.html