DOI: 10.15439/2017F415
ISSN 2300-5963 ACSIS, Vol. 11

The impact of malware evolution on the analysis methods and infrastructure

Krzysztof Cabaj, Piotr Gawkowski, Konrad Grochowski, Alexis Nowikowski, Piotr Żórawski
Institute of Computer Science
Warsaw University of Technology
ul. Nowowiejska 15/19
00-665 Warsaw, Poland
Email: {K.Cabaj, P.Gawkowski, K.Grochowski, A.Nowikowski, P.Zorawski}@ii.pw.edu.pl

***Abstract*—The huge number of malware introduced each day demands methods and tools for their automated analyses. Complex and distributed infrastructure of malicious software and new sophisticated techniques used to obstruct the analyses are discussed in the paper based on real-life malware evolution observed for a long time. Their impact on both toolsets and methods are presented based on practical development of systems for malware analyses and new features for existing tools.**

I. Introduction

ATTACKERS must continuously improve the tactics used to lure more and more users. An attempt to send an executable attached to an e-mail is well known to the most users and can be easily stopped by any anti-virus software. Nowadays attackers divide infection process into two stages. At the *first stage* some kind of executable code is delivered to the victim. Often it can be a simple macro embedded in a document or a link to an URL with malicious script. This code is responsible for downloading a *second stage* that contains the main malicious code, which is responsible for further hostile activity. In the most cases the second stage code is hosted on web servers (sites hacked without the knowledge of their owners). The more detailed description of attack techniqes used nowadays can be found in [1] [2]. Some of them are also discussed with a QNAP NAS vulnerability case study presented in [3].

The next section presents an overview of the authors' analytical infrastructure and the background for the development of some new systems due to the growing malware complexity as well as the obfuscation and other hinder techniques used. Some practical solutions are proposed and discussed.

Section III presents in details the authors' analyses of the *Locky* malware campaigns evolution since March 2016 until January 2017. The authors have observed mainly two aspects of changes: related to the code used for downloading the second stage and to some hinder techniques. These changes had a significant impact on the methods and tools used in the analyses (discussed in Section IV).

To allow continues analyses of the new tricks introduced by the *Locky* authors, some changes to the used analytical tools and a completely new tool called *StealthGuardian* were developed. Section V presents its details and the techniques used. The paper concludes in Section VI.

II. Overview of the analytical infrastructure

The first problem is the acquisition of malware samples. The well known solution utilized for years are *HoneyPot* systems [4][5]. In a *HoneyPot* the whole attacker's activity is carefully monitored and recorded for further analysis. Over the time a special kind of *HoneyPot* systems were introduced for direct gathering of malware samples (e.g. *Nepenthes* [6] or *Dionaea* [7]). *HoneyPot* systems can be of high- or low-interaction level [4]. Depending on the type, a *HoneyPot* can cover different types of malware distribution and sometimes also conduct preliminary dynamic analysis of the malware behaviour. Indisputably, *HoneyPots* are very useful.

However, to handle dynamically changing first stage malware distribution and attack vectors, *HoneyPots* have to be continuously developed. Another difficulty is introduced by obfuscation and anti-analysis techniques (e.g. multi-stage infection process) used by the malware. The authors faced these problems during this research (see further sections). Most of the papers describe *HoneyPot* systems itself and a malware analyses as separate tasks (e.g. [8][9]). Obviously, such an approach is not practical as the whole process depends on iterative improvements of *HoneyPots* to allow deeper analysis of multi-stage attack scenarios. Each stage of the attack require some specific actions to be made by the *HoneyPot*.

It is reasonable to extend the set of the sample sources and use more than just own *HoneyPots*. The results given in further sections are based on samples freely available in *malwr.com* service [10]. Everyone can send a suspicious sample to this service and it will be executed in the controlled environment (*Cuckoo* sandbox [11]). The popularity of the *malwr.com* service guarantees a very rich set of different kind of malicious software in a wide range of technologies and attack techniques. Currently 67% of more than 720 thousands of samples analysed by the *malwr.com* (as of May 2017) are public and available for security researchers. It is worth to note that *malwr.com* service is not a substitution of *HoneyPots* as a source of samples but their valuable complement.

As there is no direct data accessibility API in *malwr.com* service, the paper's authors have developed a *Malwr-Scraper* system. It downloads and parses HTML analyses description pages (details of the analysis are stored in internal database).

After manual analyses of the most recently added samples, some dedicated queries are prepared to identify all the samples of a given malware family and these samples are downloaded from the *malwr.com*. During the conducted research (further described in Section III), the whole process of data gathering lasted for 6 days (more than 620 thousands analyses of the *malwr.com* service was downloaded and parsed - more than 815 GiB of HTML). The analysis revealed more than 5900 samples associated with the *Locky* ransomware family.

In the next step a malicious code is investigated with static and/or dynamic analyses. During static analysis a malicious code is carefully investigated by a security researcher. This process gives much valuable information concerning code internal structure, used libraries and overall functionality. However, it is a very time consuming one.

Contrary, a dynamic analysis approach can reveal useful information automatically [11], almost without any security researcher activities. The gathered sample is executed in a specially crafted environment, often called a *sandbox*. All activities of the malicious code are carefully monitored (e.g. created processes, files downloaded from the Internet and all of the network communication). The evident hostile activity (e.g. sending SPAM or working exploits) are denied by the environment protection mechanism. Of course, the Internet traffic cannot be completely denied due to the fact that modern malware, during the infection, downloads further elements from the Internet (e.g. the second stage of *Locky* ransomware) or contact Command and Control servers (C&C) [12]. The security researcher must determine the trade off between the risk introduced and collection of possibly valuable information when some protection mechanisms are loosen.

One of the most notable sandbox environment is *Cuckoo* [11]. In the most cases *Cuckoo* uses *VmWare* or *Virtual Box* virtualization hypervisors. Unfortunately, due to the great popularity of this system, these two virtual environments are the most often detected by the malware (in such case it simply stops its hostile activity). To deal with that, during our research we have developed two different environments dedicated for dynamic analysis - *Maltester* [13] and *MESS* [12].

Both systems have similar structure. The management system receives commands from the user. In effect, a clean virtual machine is created (a sandbox system) – a snapshot feature of the hypervisor is used. Launched sandbox machine has a custom software responsible for receiving a malware sample for the analysis and its execution. Due to security concerns all the traffic between the Internet and the sandbox system (with a suspicious file) is forwarded by an additional gateway system which implements Firewall and NAT services. Any hostile activity is stopped at this system. The overall infrastructure of the developed *Maltester* and *MESS* systems is very similar to the one used by the *Cuckoo* sandbox. However, our systems utilize not so common (in a security world) hypervisors: respectively *Xen* and *Microsoft Hyper-V*. Our research shows that for some malware samples the analysis has failed in well known systems but they were successfully evaluated in our custom dynamic analysis environments.

III. Locky case study

The results presented in this paper are continuation of the previous works associated with the analysis of the *CryptoWall* ransomware conducted at the beginning of 2015 [13]. Because of unknown reasons new samples of the *CryptoWall* were not observed in the January 2016 and later the whole *CryptoWall* infrastructure was shut down. However, around the middle of the February a new ransomware family appeared - called *Locky*. Like its predecessor, it encrypts user data and uses asymmetric cryptography. Public key used for the encryption is downloaded from a C&C server. Contrary to the *CryptoWall*, the *Locky* family uses more complicated schema for C&C access. Each sample has a few hard-coded C&C IP addresses. If they are shutdown, *Locky* uses domain generation algorithm (DGA) for finding other working C&C servers.

Since the middle of the March 2016 to the beginning of the January 2017 more than 5900 samples associated with *Locky* malware were reported in the *malwr.com* service. Around 700 of them are in Windows executable (*PE32*) format. In the remaining 5200 samples of the first stage we identified 278 hostile Excel and 753 hostile Word documents (around 20% of samples).

The characteristic for *Locky* campaigns is that the first stage code is very often in JavaScript and is sent to the victims as files with *.js* and *.wsf* extensions. The conducted research revealed that in more than 75% of the *Locky* first stage code samples. What should be emphasized, Microsoft Windows silently executes JavaScript code if given file extension is *.js*.

Among all the analysed JavaScript-based *Locky* samples, the simplest and the shortest code is contained in 17 lines (693 bytes)[1]. In more recent samples, this first stage code become obfuscated using various methods. In effect, the code became longer and more complicated for analysis. The longest observed *Locky* first stage code has a length slightly more than 1 Megabyte - exactly 1064661 bytes[2]. The code presented in the Fig. 1 as Original Code with high probability was manually de-obfuscated by a security analyst (used variables, function names and all parameters use human readable names). However, obfuscated code with the same functionality can be observed in real samples sent to the victims. Fig. 1 presents a few sample obfuscation techniques (parts A, B, and C).

In all three presented cases variables with strange names (`Njofagi`, `DqWgVQeF`, and `JBGUHYm2e`) represents ActiveXObject *MSXML2.XMLHTTP*, which is used for preparation of a HTTP request and downloading of the *Locky* second stage code. The first obfuscated excerpt code presented in the Fig. 1 obfuscate only variable names. Code presented in excerpt B encodes parts of JavaScript code using Unicode. Despite complicated appearance, this code can be easily de-obfuscated even using Unicode decoding services freely available online[3]. The last excerpt in Fig. 1 presents a technique in which the program text parts (like web server address and

[1]Sample from *malwr.com* with MD5 *dafb1c1626d822e9de4a7ae5b33eae59*.
[2]Sample with MD5 *9b823aeed9fda550bddeb735f35e6d3b*.
[3]For example https://www.branah.com/unicode-converter.

```
/** Original Code **/
xmlhttp['open']
  ('GET', 'http://XXX.YY/45g456', false);
xmlhttp['send']();

/** A **/
Njofagi[Uzkoy]
  ("GET", "http://XXX.YY/45g456", false);
Njofagi["send"]();

/** B **/
DqWgVQeF['o\u0070\u0065n']
  ('G\u0045T',
   '\u0068\u0074\u0074\u0070...', false);
DqWgVQeF['se\u006E\u0064']();

/** C **/
JBGUHYm2e[TTBLVVx3k]
  ("G\x45T",
   "ht"+"tp"+"://"+"XX"+"X."+"YY"
    +"/a"+"se"+"32f"+"f",
   false);
JBGUHYm2e["s"+"end"]();
```

Fig. 1. Various obfuscation techniques (A, B, and C) and the Original Code.

function names as well) are divided into short chunks and dynamically concatenated before use. Code excerpts are taken from the samples with code lengths of 2468[4], 1707[5], and 533256[6] bytes.

A huge span of sizes of the *Locky* first stage code sizes was observed - from below 1000 bytes to more than 1 MiB during the analyses period. The most evident strange behaviour which was observed between April and May concerns a sharp rise of the JavaScript code size. Analysis of these samples revealed that core part of the code is similar to the already observed. However, some random text is placed in variously defined comments[7]. Further samples, with even larger sizes have introduced various lines of random text, for example, a repeated pattern of '12345667890'[8]. We suspected that these changes in code utilize some flaws in security software, for example, anti-virus software which cannot properly detect malware in such a big JavaScript code.

Analysing another sharp rise in the size of the exploits (from a few kibibytes to more than 10 KiB) showed that a new kind of obfuscation was introduced by the attackers: the whole protected JavaScript code is partitioned into some small chunks and concatenated just before the execution[9].

In the third case, instead of partitioning the code into chunks, a final code is included as an encrypted text. The simple mono-alphabetic cipher is used. The most characteristic part of this type of downloader is a slightly obfuscated array used in decryption procedure of the final downloader code.

[4]Sample with MD5 *73a65a07887c705971d6d01a546bc748.*

[5]Sample with MD5 *0ed65a747b98989f24e660d495c71524.*

[6]Sample with MD5 *e04892726b496ce5f0c9fc9d08fd73b5.*

[7]e.g. a sample with MD5 *c9e26aec4405e79131a585802bcd0de9.*

[8]Sample with MD5 *fcbfe7604f94f15abdbe6fea1c865cc4.*

[9]Sample with MD5 *d6eeeb79c1be9decd781a200c67a92e6.*

In another case a completely different kind of the first stage code was identified. Previously used *Locky* downloader was directly downloading a second stage executable. However, around 24th of May this behaviour changed: *Locky* started to download a second stage malware which was encrypted.

During the conducted research we observed the evolution of the used encryption techniques. The first encrypted samples (which appeared first on distribution servers at 24th of May) were using a simple mono-alphabetic substitution cipher. Decryption code implementation uses *XOR* function with a single byte key - even during viewing of such file in hex-editor, the repeating strings of the same byte can be observed (due to many 0-valued bytes in the Windows executable format). To hinder the analysis, the attacker reverses the whole file and adds a few random bytes at the end of the file. Due to the used key - *0x73*, which represents in the ASCII letter *s*, a downloaded second stage file in hex-editor have a catching in the eye numerous strings of letter *s*.

In the following weeks some longer keys were observed. The longest one was automatically generated from two numbers and have a length of 256 bytes. However, from the June 2016 in most cases some shorter keys were observed - in the most cases using 32 bytes of ASCII characters. Due to the fact that most hex-editors presents in one line multiplicity of 8 or 16 bytes, this size of a key produce repeatable pattern easily visible in the viewed file, which in effect simplify reproduction of the original key. Despite these drawbacks, this behaviour was most common to the end of the year. However, occasionally other key lengths appear, but all of them are only within ASCII characters range.

Additional change in the downloaded second stage of the *Locky* malware, which appears together with encryption, concerns a format of the executable which takes the form of DLL library. The DLL-based second stage samples appeared for the first time at the 29th of August 2016. Usage of the DLL is well known hinder behaviour used nowadays by the attackers. However, owners of the *Locky* have extended this technique. In the previously analysed malware, the samples distributed as a DLL required a usage of the *rundll32* utility – *Locky* samples do not run by the execution of a standard DLL entry function executed by the *rundll32*. To make it difficult, *Locky* malware uses additional entry function, which name is is included in the encrypted first stage code. Moreover, the name of that secret entry function was not given in the exported functions table. After some initial investigation, we suspect that the used entry function is dynamically decrypted by some other standard entry function. In effect, to run *Locky* malware sample, this entry function must be discovered before any further analyses. So, to conduct dynamic analysis of such sample, some modifications of the analysing environment have to be introduced.

To the end of the November only one simple entry function name was used ("qwerty"). Later, this entry function was changed more and more often – at the end almost on daily basis. The last analysed *Locky* campaign used 25 of such functions.

IV. Malware defence techniques

Malware evolution is driven not only by new attack possibilities, but also by the need to obstruct analysis efforts. The longer analyses of malware behaviour means longer activity in the environment – infection of thousands of additional machines. So, the obstruction of analysis is a natural next step in the evolution of any malware.

To overcome static analysis efforts, malware can use various mixes of encoding, encryption, mutation and other operations (some real-life examples are presented in Sec. III). This made the dynamic analysis more and more important in the past years, yet malware creators are aware of that and also enhanced their software. The most basic, but very effective, malwares' defence strategy is to detect the fact of being analysed and just stop to do anything. Because dynamic analysis requires some supervisor software to be present, the easiest way for malware to detect the analysis is to check if it runs in a supervised environment or not. For example, a debugger connected to the infected process or execution on a virtual machine can mean that the software is being analysed. Malware does not need to make any additional checks for the potential reasons of debugging or presence of virtual environment – vast majority of users does not use debuggers and works on a real hardware, so, even if malware will loose some targets, it still gains a lot more.

Some common supervision detection techniques targeting Microsoft Windows operating system are described below. Nevertheless, variations of the same techniques can be also used by malware working on any other system.

1) Checking for debugger presence using system API: Standard Windows API library *kernel32* provides two functions which can be used by any software to detect debugger connected to the current process: `IsDebuggerPresent` and `CheckRemoteDebuggerPresent`. It is one of the simplest check a malware can perform, but will be effective against simple debugging of infected process.

2) Checking for remote debugger presence: Instead of relying on system API, a process can retrieve remote debugger information by directly querying the kernel using `NtQueryInformationProcess` from *ntdll* library. This method is used to retrieve `EPROCESS` structure that contains information about potentially connected remote debuggers.

3) Checking PEB structure: Low level check of debugger presence can be achieved by direct read of *BeingDebugged* flag from the *PEB* structure which is available at a predefined address for each process (i.e. `fs:[0x30]` on 32-bit system and `gs:[0x60]` on 64-bit). Reading this flag requires some assembly code, but makes software independent from any external library.

4) Instruction execution time measurement: It is harder to determine if a program executes in a virtual or real machine, as, in theory, virtualization should be transparent to the guest system (end its processes). However, the simplest check can be achieved by measuring the execution time of a single processor instruction - in a virtual system this time is longer. Using the `RDTSC` instruction a program can read *TSC* registry value, which contains the number of cycles since the last processor power-on. Comparing two consequent reads with some expected difference can hint the presence of additional virtualization layer between the hardware and the software. Although simple, this method is not very accurate and can yield many false positives.

5) Validating machine and user name: Names of machines used by analysis systems can contain strings like *maltest*, *sandbox*, *virus* etc. Malware can compare current machine or user name against dictionary to quickly get hint for being analysed. Library *advapi32* provides the method `GetUserName` for reading user name while the *kernel32* library provides machine name via `GetComputerName` method.

6) Validating peripherals properties: Various hipervisors can use some predefined names for emulated peripherals. For example *Hyper-V* uses it's name in the name of some peripherals, including BIOS name and version. Other properties which can be used by a malware include MAC address of a network card, which usually comes from a pool assigned for the virtual machine manufacturer. Reading those values usually requires checking various keys from *Windows Registry*.

7) Checking parent process name: Usually a malware after infection is executed on each user login (parent process: *explorer.exe*) or as a system service (parent process: *svchost.exe*). Other parent processes can mean that malware sample was executed by some kind of a supervisor process, which is sometimes needed even in virtualized environment. Process parent can be found in *PROCESSENTRY32* structure, which can be queried using `Process32First` and `Process32Next` functions from the *kernel32* library.

8) Expecting cursor movements: Some malware tries to detect if there is any real live user interaction ongoing before deciding to attack the system. One of the easiest techniques malware uses is to check a mouse cursor position on the screen. Basic, virtual machine based, sandboxes, executed in automated way for the analysis do not simulate mouse cursor movements. Process can acquire cursor status using `GetCursorInfo` provided by the *user32* library.

V. StealthGuardian

Overcoming malware counter-analysis efforts is required to continue with efficient dynamic analysis of future samples. The idea is to enhance the system for the analysis in such a way that it will appear as stealth to the analysed sample as possible. Various techniques of achieving that goal were developed, tested and implemented in the authors' Institute as a *StealthGuardian* software. It wraps the execution of a sample providing additional layer upon the operating system. It can be integrated with any analysis system as it does not introduce any new requirements for the supervisor system. In our Institute it became a part of the MESS infrastructure - a supervisor program running inside a sandbox virtual machine, which is used in MESS to launch the analysed sample, can now launch *StealthGuardian* which then executes the sample for the analysis.

First tested solution was based on attaching to the executed sample as a debugger and switching into *single-step* execution mode. However, this extremely reduces the performance of the analysis – sample execution can take even a couple of thousand times longer. This technique could still be used if a *single-step* would be turned on only for some critical parts of the sample execution, but some preliminary analysis of the sample has to be performed to determine these parts.

The obstruction techniques described in Section IV can be divided into two groups based on a method of implementation: 1) direct processor instruction execution, 2) Windows API methods call. Using a debugger can handle the first group and some of the calls from the second group - those which can be easily recognized as assembly instruction sequence. But recognizing all the calls using a debugger seems to be too complicated and costly in terms of the performance. So, the proposed solution requires capturing all the calls of API functions and temper with their results before providing them to the analysed sample. It will not cover all the techniques used by malware but it is effective and efficient.

Capturing Windows API calls can be achieved using *inline hooking* technique. It involves replacing the code of the selected library functions with the calls to the substituted functions, which then can call the original functions. A *trampoline* is quite well known and popular technique, so various implementations are available. *StealthGuardian* uses *MinHook* library[10] to intercept Windows API calls responsible for malware defence techniques described in Section IV. To make it work properly, it was necessary to prepare both *x32* and *x64* versions of *StealthGuardian*, so it can work with all architectures used by different malware. It was also necessary to overcome some other technical issues, like supporting both ASCII and UTF versions of some API functions and hooking libraries loaded by direct calls to the *LoadLibrary* function, even in new threads or processes spawned by the original (parent) process. Proper behaviour of the hooked methods had to be designed – returning completely random values may not trick smart malware (e.g. user name can be random, but has to be constant for the whole sandbox execution).

Final version of *StealthGuardian* was able to trick special sample prepared in the Institute, which was using all supported by *StealthGuardian* defence techniques. Following experiments on some real samples also proved usefulness of this solution. For example, it allowed to observe the whole known behaviour of the *Win32/Urnsnif*[11].

VI. Conclusion

For the last few years we observe arms race between black hats and security community. Cybercriminals introduce new attack tactics which later are discovered, analysed and mitigated using new security mechanisms. When more and more users start using these mitigation mechanisms, cybercriminals introduce new methods and the cycle starts once again. However, the time between the next cycle is decreasing. Few years ago we observed a new sample of a given malware family once a few weeks. During the analysis of the *Locky* we observed from two to three distinct samples daily.

These rapid changes of cybercriminal tactics are challenging for the security community. Conducted research showed that the usage of dynamic analysis could reduce the time needed for performing the analysis of a new malware sample. However, sometimes the changes introduced by the malware are so significant that the environment for the analysis must be upgraded. This paper describes with details some of these developments which allow the analysis of the most recently appearing malware samples.

[10]https://github.com/TsudaKageyu/minhook

[11]Sample with MD5 *4df3ce5c9a83829c0f81ee1e3121c6ea*.

References

[1] T. Herr and E. Armbrust, "Milware: Identification and implications of state authored malicious software," in *Proceedings of the 2015 New Security Paradigms Workshop*, ser. NSPW '15. New York, NY, USA: ACM, 2015, pp. 29–43. [Online]. Available: http://doi.acm.org/10.1145/2841113.2841116

[2] C. Lever, P. Kotzias, D. Balzarotti, J. Caballero, and M. Antonakakis, "A Lustrum of malware network communication: Evolution and insights," in *S&P 2017, 37th IEEE Symposium on Security and Privacy, May 23-25, 2017, San Jose, USA*, San Jose, UNITED STATES, 05 2017. [Online]. Available: http://www.eurecom.fr/publication/5177

[3] K. Cabaj, K. Grochowski, and P. Gawkowski, "Practical problems of internet threats analyses," in *Theory and Engineering of Complex Systems and Dependability. Proceedings of the Tenth International Conference on Dependability and Complex Systems DepCoS-RELCOMEX*, ser. Advances in Intelligent Systems and Computing, W. Zamojski, J. Mazurkiewicz, J. Sugier, T. Walkowiak, and J. Kacprzyk, Eds., vol. 365. Springer International Publishing, 2015, pp. 87–96.

[4] K. Cabaj and P. Gawkowski, "Honeypot systems in practice," *Przegląd Elektrotechniczny*, vol. 91, no. 2, pp. 63–67, 2015.

[5] M. L. Bringer, C. A. Chelmecki, and H. Fujinoki, "A survey: Recent advances and future trends in honeypot research," *International Journal of Computer Network and Information Security*, vol. 4, no. 10, p. 63, 2012.

[6] P. Baecher, M. Koetter, T. Holz, M. Dornseif, and F. Freiling, *The Nepenthes Platform: An Efficient Approach to Collect Malware*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 165–184. [Online]. Available: http://dx.doi.org/10.1007/11856214_9

[7] T. Sochor and M. Zuzcak, *Study of Internet Threats and Attack Methods Using Honeypots and Honeynets*. Cham: Springer International Publishing, 2014, pp. 118–127. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-07941-7_12

[8] P. Baecher, M. Koetter, T. Holz, M. Dornseif, and F. Freiling, *The Nepenthes Platform: An Efficient Approach to Collect Malware*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 165–184. [Online]. Available: http://dx.doi.org/10.1007/11856214_9

[9] M. Xu, L. Wu, S. Qi, J. Xu, H. Zhang, Y. Ren, and N. Zheng, "A similarity metric method of obfuscated malware using function-call graph," *Journal of Computer Virology and Hacking Techniques*, vol. 9, no. 1, pp. 35–47, 2013. [Online]. Available: http://dx.doi.org/10.1007/s11416-012-0175-y

[10] C. Guarnieri and A. Tanasi. malwr.com website. [Online]. Available: http://malwr.com

[11] M. Vasilescu, L. Gheorghe, and N. Tapus, "Practical malware analysis based on sandboxing," in *2014 RoEduNet Conference 13th Edition: Networking in Education and Research Joint Event RENAM 8th Conference*, Sept 2014, pp. 1–6.

[12] K. Cabaj, P. Gawkowski, K. Grochowski, and A. Kosik, "Developing malware evaluation infrastructure," in *Proceedings of the 2016 Federated Conference on Computer Science and Information Systems*, ser. Annals of Computer Science and Information Systems, M. Ganzha, L. A. Maciaszek, and M. Paprzycki, Eds., vol. 5. IEEE, 2016, pp. 1001–1009.

[13] K. Cabaj, P. Gawkowski, K. Grochowski, and D. Osojca, "Network activity analysis of cryptowall ransomware," *Przegląd Elektrotechniczny*, vol. 91, no. 11, pp. 201–204, 2015.