Titans' revenge: detecting Zeus via its own flaws

Marco Riccardi[a], Roberto Di Pietro[a,b], Marta Palanques[a], Jorge Aguila Vila[c]

[a]*Barcelona Digital Technology Centre, eSecurity Research Group,*
Roc Boronat 117, 5th floor, 08018 - Barcelona, Spain
E-mail: {mriccardi,rdipietro,mpalanques}@bdigital.org
[b]*Università di Roma Tre, Maths Dept.*
L.go S. L. Murialdo 1, 00146 - Roma, Italy
E-mail: dipietro@mat.uniroma3.it
[c]*CaixaBank, CSIRT,*
Av. Diagonal, 621, 08028 - Barcelona, Spain
E-mail: jaguila@lacaixa.es

Abstract

Malware is one of the main threats to Internet security in general, and to commercial transactions in particular. However, given the high level of sophistication reached by malware (e.g. usage of encrypted payload and obfuscation techniques), malware detection tools and techniques still call for effective and efficient solutions. In this paper, we address a specific, dreadful, and widely diffused financial malware: Zeus.
The contributions of this paper are manifold: first, we propose a technique to break the encrypted malware communications, extracting the keystream used to encrypt such communications; second, we provide a generalization of the proposed keystream extraction technique. Further, we propose *Cronus*, an IDS that specifically targets Zeus malware. The implementation of Cronus has been experimentally tested on a production network, and its high quality performance and effectiveness are discussed. Finally, we highlight some principles underlying malware—and Zeus in particular—that could pave the way for further investigation in this field.

Keywords:
botnets; e-crime forensics framework; banking trojans; fraud detection system; cryptanalysis

1. Introduction

Malware is probably the most recurrent threat over the Internet, and probably one of the most dreadful ones. The monetary loss caused by cyber crimes has been recently estimated to be over one hundred billion *dollars in cash* [1]. This is a dramatic value for Anti Virus (AV) companies, considering that 54% of these crimes have been linked to computer virus/malware incidents. The Zeus malware has been defined as the world's most notorious banking malware, due to its sophisticated ways to steal banking credentials or covertly falsify electronic transfers. In July 2009, a security report from Damballa [2] ranked Zeus as the most threatening botnet, infecting around 3.6 million of computers in the US alone. Although financial institutions are reluctant to share an accurate value of the financial loss caused by Zeus, it was estimated that it has caused damages of more than 100 million US dollars since its first detection. There are some technical countermeasures against malware which can be deployed, such as AV software and Intrusion Detection Systems (IDS). However, one of the reasons for the catastrophic estimated consequences of Zeus is the low detection rate of Zeus by most of the Anti Virus (AV) vendors [3]. Moreover, the most critical element in this confrontation is time. The need for a quick way to detect malware traffic usually makes the difference between a successful malware and a defeatable one.
Malware coders, on their side, have developed some hiding techniques in order to escape detection, the simplest of which (though the most used one) being payload encryption. In particular, symmetric encryption of the payload is chosen due to its easier management, as opposed to public key encryption[1]. Algorithms used are XOR-based and range from custom developed ones, such as the ones used by TDL[4] and Torpig[5], to RC4, which is used by Zeus and hence is the focus of this paper. In addition to encryption, a very weak obfuscation algorithm is occasionally applied, consisting in base-64-encoding the payload.
Each of the mentioned techniques implies that the malware, once installed on the host machine, carries the encryption key with it. Hence, it could seem easy, once the malware has been detected on a system, to extract the key. Indeed, it was simple in the past, when the key was placed on a fixed memory location. Unfortunately, novel techniques, such as obfuscation of the key [6], have made the quest for the encryption key quite difficult to win.

Contributions- This paper provides several contributions. First, we propose a technique to extract the keystream used by

[1]However, note that the Zeus FAQ specifically address this issue and argue that public key encryption is not used due to the fact that confidentiality of the payload is not a priority, while detection avoidance is.

a specific malware (Zeus) to encipher its payload. This makes it possible to further identify the Zeus malicious traffic inside a computer network. The keystream extraction technique does not require static analysis to look for the encryption key (usually obfuscated in the executable), but only relies on analyzing the network traffic that has been generated by the infected computer. As our preliminary investigation shows, it is easily automatable and particularly efficient. Further, we propose a SW architecture that, leveraging the presented techniques, proposes a general technique to extract the keystream. Additionally, we also present *Cronus*, an IDS that is able to detect the malicious botnet i.e. both the C&C and its bots, only with a few bytes of the key that was previously found. Finally, we demonstrate the applicability of the proposed IDS by testing it inside our corporative network. Experimental results prove the correctness and viability of our findings, and pose the basis for further research focused on identifying generic malicious traffic.

Roadmap- This paper is organized as follows: in Section 2 we report on related work. In Section 3 we provide a detailed background on the Zeus malware family. A comprehensive code analysis of the malware encryption routines is also provided, together with a detailed explanation of Zeus operating mode. In Section 4, we formally introduce the methodology used for extracting a key that could be utilized to decipher certain Zeus network traffic, also providing a pseudo code that leverages the flaws discovered in the malware crypto routines. In Section 5 we describe the implementation of the designed software through our experimental phase and in Section 6 we present a case study in order to evaluate the applicability of the framework in a real-case scenario. In Section 7, we report lesson learned, while in Section 8, we provide some concluding remarks.

2. Related work

In the recent years, financial malwares and their fraudulent activities, e.g. phishing and cyber frauds, have been investigated in several ways. For instance, Chandrasekaran et al. [7] proposed an approach to detect phishing attacks using fake responses and monitoring site behavior, while Birk and Gajek [8] extended such an idea to a framework leveraging *honeytokens* [9] to track phishers.

Several technology-based strategies have been also developed to dynamically analyze and defeat botnets. Holz et al. [10] introduced a methodology to track and observe botnets using honeypot technologies. Inspired by the work of Holtz, *the Dorothy Framework* [11] was developed, which aims to analyze, track, and visualize a botnet in a highly automated fashion. Such a framework was also customized [12] to investigate the financial oriented botnets. However, recent malwares use crypto routines to cipher their traffic to communicate with their Control Center (C&C). Thus, it is often required to extract the communication symmetric key from the malware binary, in order to successfully decipher the network flow content. To fit this purpose, Caballero et al. [13] proposed a tool that leverages the binary code reuse technique to extract code fragments of the malware encryption/decryption routines, and employ this to uncover the malicious communication traffic. Leader et al. [14][15] reached the same goal in a different way: monitoring the data exchanged by the malware binary at the I/O level of the sandbox interfaces.

Along with the increasing research efforts on malware analysis/detection, a number of different countermeasures have also been explored in order to deal with the ever evolving botnet threat. For instance, DNS *sink-holing* has been frequently investigated [16] [17] [18] to avoid infected computers reaching their C&C by resolving it to a not reachable IP address, i.e. loopback or reserved IP address. Also pattern-matching identification techniques were proposed [19][20], to enroll IDSs to identify and block malicious traffic related to a botnet. In addition, defaming countermeasures around financial malwares have been investigated by Ormerod et al. [21]. In particular, they investigated the defaming botnet toolkits aiming at discouraging or prosecuting the end-user of the stolen credentials. Ford and Gordon [22] research deeply focused on attacking the Malicious-code generated revenue streams in order to economically damage the criminal business model. Regarding Zeus, the banking trojan which is the main focus of this paper, Binsalleeh et al. published a comprehensive analysis of the Zeus crimeware toolkit version 1.2.4.2 [23]. They reverse engineered the malware in order to discover its inner features and to better understand its behavior in order to inject fake information into the botnet C&C, and consequentially defame the malware toolkit.

In their paper, they pointed out that communications between the infected hosts and the Zeus C&C rely on a vulnerable implementation of the RC4 cipher which can be exploited by a keystream reuse attack. In a preliminary work of ours [24], we further investigated this vulnerability focusing on the host analysis. This work extends previous research results on Zeus, by investigating new techniques to detect and decipher Zeus network communication. In addition, our work is focused on a recent version of Zeus which shows a higher level of complexity with respect to the analysis techniques used in the aforementioned papers.

3. The Zeus malware

The Zeus malware (also known as Zbot) first appeared in 2006 when a security firm released a full reverse engineering analysis [25] about an unknown trojan named PRG. Since then, it has been modified and customized to suite specific needs and released in different variants, each one offering innovative features to steal sensitive information. Zeus was originally created, distributed, and maintained by Russian cybercrime gangs [26]. Historically, the russian cyber underground scene is mainly of criminal intent, as the ultimate goal would be to maximize the amount of money the participants could make. On top of that, the general hacking environment in Russia can be mainly characterized as financially driven. Controversially, legal persecution of cyber crime in Russia is not a priority. Strategically, Russian hackers usually avoid targeting regular Russian citizens in order to gain a shared sense of toleration and even admiration. In addition, the strategic choice to prefer targets outside the Russian Federation complicates the cross-border cooperation that is needed to investigate the cybercrime related frauds. Indeed, investigating crimes against foreign interest does not represent a priority for law enforcement officers in Russia [26].

3.1. The crimeware toolkit

The Zeus crimeware toolkit used to be sold privately for a price that ranged between \$800 and \$4,000. However, its market value has been drastically reconsidered since the leakage of its source code in March 2011. Interestingly, on February 2011, a forum post was published on a notorious underground forum where a seller offered the full version of the Zeus source code (toolkit included) for a "high price". Speculations suggest that the price asked by the seller was floating around \$100,000 [27]. When the source code was leaked, it generated a sudden increase of new Zeus malware variants, offering to the entire Internet community a framework to easily set up a robust financial botnet for free. The builder offers the capability to create customized malware executables and the botnet configuration files needed for correct botnet operation. In this way, users could create their own malware which targets the financial institutions that have been inserted in the malware building process. Additionally, the control panel offers the ability to easily administer the botnet through a user-friendly PHP page, that allows the botnet owners to quickly retrieve a comprehensive status of their bots, and to be able to download the stolen information. Moreover, the installer comes with a bilingual manual (Russian and English) which explains the fundamental steps required to install the Zeus suite.

The Zeus malware is a software designed to make a profit also by its development. As a matter of fact, its full customizability offers to cyber-criminals the opportunity to develop new modules and sell them in the market place. For instance, the form grabber module for Mozilla Firefox and the back connect module (the latter offering the botnet owners to have a direct access to the console of the infected computer), were sold for around \$2,000. People interested in particular features can post their advertisement on such underground forums, and offer money to those who could develop the requested customized software.

3.2. Zeus communication protocol

Like most banking trojans, the Zeus's goal is to steal sensitive information that could lead the attacker to carry out a financial fraud against the victim. The Zeus ecosystem is usually composed of three different entities: the bot, e.g. the machine that has been infected, the Command and Control — here in after C&C, or *dropzone* — i.e. the main server where the control panel is hosted and where the bots send the stolen information, and the configuration server: the server where the configuration file is hosted, ready to be downloaded by the bots. The C&C and the configuration server usually overlap their role, offering to botnet owners the comfort of administering only one server.

Once the victim's computer is infected, the malware hooks every API call in order to grab sensitive information before it is sent through the network. In this way, the malware is able to steal HTTPS sessions before they are encrypted, and also to send them to the C&C. Stolen information mainly resides in HTML input data forms, POP, FTP accounts credentials, X509 certificates stored in the browser, and cookies saved in the system. Occasionally, the botnet owner can also request a screen snapshot to the owned bots, forcing them to send a screenshot of what they are currently looking at.

Stolen data is regularly sent to the botnet's *dropzone* through two different communication channels. The first one, referred in the Zeus configuration file as *log*, consists in a small keep-alive message containing all the main status information about the zombie i.e. *botID*, *botnetID*, IP-address, bot OS, etc. Notably, this packet is sent to the *dropzone* every two minutes by default, and consists of the most frequent communication type between the zombie and its C&C. Thus, its periodic emission could be leveraged by an anomaly based IDS to identify an infected computer inside a network. The second communication flow used by Zeus, referred as *report*, occurs less frequently than the *log* one i.e. by default every ten minutes. As well as the zombie status information included in the *log* packet, *report* packets also contain all the data that has been stolen in the system. Hence, this message is usually bigger than the previous one, and packet fragmentation according to the network MTU size is often required by the OS for sending the entire TCP segment.

The TCP packet structure of the Zeus *report* communication is shown in Figure 1. The packet begins with a 48 bytes long *header*, followed by *n* bytes containing the *body*, which is divided into several *items* containing all the stolen information categorized by prefixed labels. The *header* consists of 20 bytes of random padding, followed by 12 bytes used for the *header info* and 16 used to store the MD5 hash of the Zeus *body*. The *header info* is composed of three slots of four bytes each, respectively containing the Zeus *body size*, the Zeus *item flags* and the number of Zeus *items* contained in the *body*. Zeus *item flags* are used to indicate whether the Zeus *body* is compressed and, if so, how to manage contained items. The MD5 hash is used by the control panel as integrity verification: once the C&C receives the message, it firstly calculates the MD5 of the *body*, and then compares it with the one stored in the packet. If the two hashes match, the packet is processed by the C&C; otherwise the packet is dropped.
Each Zeus *item* is composed by a 16 bytes long *item header* and an *item body* with a variable length. The *item header* is divided into four 4 bytes long slots containing the *itemID*, a zero padding string, and the length of the *item body*. Note that the last two slots represent the same information — a clear rational for this is currently missing. The *item header* is used to identify the information by its type and to anticipate the length of its body. In this way, the Zeus control panel knows how to dissect the received packet, and it is able to store each *item* in the correct SQL field. Table 1 shows the main Zeus *item IDs* used to identify the stolen information. The values of *items* `10002` and `10003` are *constant*, and are fixed at the time of the mal-

ItemID	Value
10001	SBCID_BOT_ID
10002	SBCID_BOTNET
10003	SBCID_BOT_VERSION
10005	SBCID_NET_LATENCY
10006	SBCID_TCPPORT_S1
10007	SBCID_PATH_SOURCE
10008	SBCID_PATH_DEST
10009	SBCID_TIME_SYSTEM
10010	SBCID_TIME_TICK
10011	SBCID_TIME_LOCALBIAS
10012	SBCID_OS_INFO
10013	SBCID_LANGUAGE_ID
10014	SBCID_PROCESS_NAME
10015	SBCID_PROCESS_USER
10016	SBCID_IPV4_ADDRESSES
10017	SBCID_IPV6_ADDRESSES
10018	SBCID_BOTLOG_TYPE
10019	SBCID_BOTLOG

Table 1: Zeus item IDs

ware creation[2]. We refer to these items as *trojan items*. As for the other *items*, they are variable and strictly depend on the infected system — we refer to them as *environmental items*. It is important to note that all the stolen cookies are stored into the *item id* 10009, which can reach several Mbytes of length. Indeed, the more cookies the system has, the bigger the entire packet will be.

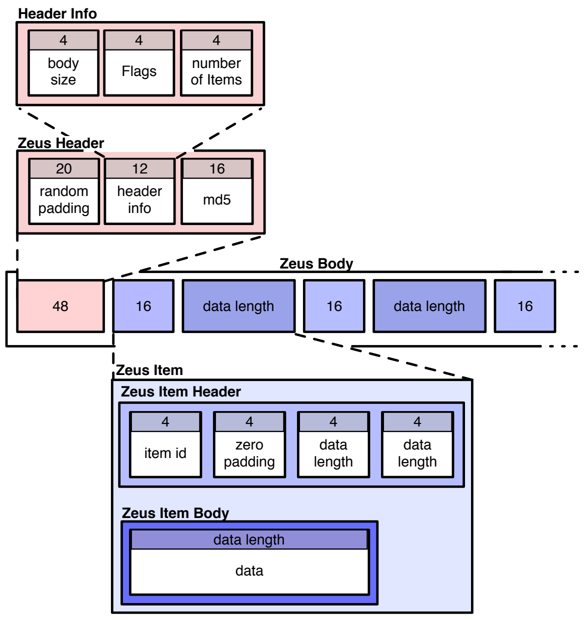


Figure 1: Zeus TCP packet structure

[2]While *item* 10002 is specified by the bot master, *item* 10003, which defines the bot version, is hard coded within the Zeus Builder toolkit.

3.3. *Zeus crypto routines*

Since its first version, the Zeus trojan encrypts the network communication with its C&C in order to avoid identification by pattern-recognition algorithms used by the IDS. The Zeus 1.x malware version used to encrypt its data using the RC4 stream cipher [23], which offers a fast and lightweight encryption in terms of CPU consumption. Indeed, since one XOR operation over a PC *word* can take from four to eight CPU clocks to be completed, a 2GHz CPU could successfully execute from 250M to 500M XOR/s. Besides performance, the RC4 algorithm is also very easy to implement, considering that it can be written from scratch in just few lines. The RC4 *seed*, the security of the encryption algorithm relies upon, is set by the botnet owner during the malware sample building phase. Once the malware is executed, the seed (also *key* in the following) is stored in a predefined and fixed memory space, ready to be used every time the bot needs to download a new configuration file, or needs to send the stolen information. Notably, the RC4 initialization vector is reused at every invocation, meaning that every outgoing packet will be encrypted starting from the beginning of the keystream. The research results provided in this paper were a direct result of leveraging this implementation weakness, allowing us to reuse the reconstructed parts of the keystream to decipher the configuration file.
For previous Zeus versions, researchers used to have several ways to automatically find out the key and to decrypt the traffic between the bot and its C&C. As long as the encryption key was stored in a fixed address space, the researchers could automatically dump all the volatile memory of the infected machine and extract the data contained inside a known range space. When the second version of the Zeus malware came out, it introduced a new layer of encryption and a new way to dynamically store its key in memory. Because of this, all the automated frameworks developed around Zeus 1.x needed to be completely reviewed.
The version analyzed in this paper, Zeus 2.0.8.9, further introduced a new layer of obfuscation on its data communication process between the infected machine and the C&C. Developers who added this additional obfuscation layer probably wanted to avoid IDSs from detecting known patterns previously learnt by Zeus 1.x malware analysis. The sequence diagram in Figure 2 summarizes the encryption and obfuscation routines used by the analyzed Zeus malware. Thus, after decrypting certain RC4 Zeus network traffic, an additional decryption function needs to be invoked to break the entire cipher ring — enabling recovery of the corresponding plain text. Notably, our research demonstrates that this additional encryption layer does not really hinder cryptanalysis. As outlined in further sections, this finding allowed us to identify certain malicious traffic patterns without necessarily breaking the obfuscation layer.

At the time of writing, there are several Zeus variants in the wild, due to the source code leakage. Some of them introduce the same domain flux technique used by Confiker and Torpig to evade DNS *sink-holing*, and P2P communications as the main communication channel between the bots and their C&C. However, apart for a specific variant that uses AES instead of

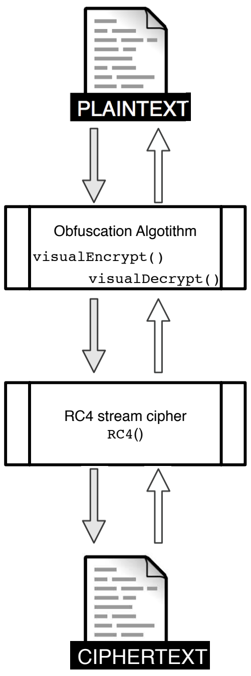


Figure 2: Encryption/Decryption flow of Zeus 2.0.x

RC4[28], the encryption mechanism has not been drastically altered from the previous version, preserving the applicability of our findings also to the latest Zeus version.

3.4. Zeus Control Panel

The HTTP control panel is in charge of decrypting incoming communications, sent by the botnet's zombies to their C&C, and to encrypt the relative answers. Its code is written in PHP, and includes three pages in its root directory: `cp.php`, `gate.php` and `install.php`. The leaked Zeus 2.0.8.9 source code was analyzed, taking advantage of the fact that all the code is well commented at every function point and, although comments are written in Russian/Cyrillic, it was possible to obtain a reasonable translation using the Google Translate web service [29].

The `install.php` file is an executable page that automatically configures the server environment with the malware requirements. The installation process takes a few minutes: once the botmaster has inserted the environment information into the install page (such as DB server credentials and user admin password) the script takes care of filling in the database with the SQL schema needed for the botnet control panel. Besides the installation page, `cp.php` is the main page which the botmaster uses for controlling the botnet. This web page mainly represents the status of the botnet by querying the MySQL database, and offers the capability to send custom commands to the owned bots. The PHP page handling the incoming HTTP POST messages is `gate.php`, where our attention was particularly focused on.
The code inside this page is in charge of decrypting the ciphered TCP flow incoming from the botnet zombies, and dissecting the payload into different variables, as previously outlined. Furthermore, the code takes care of filling in the database with the retrieved information in cleartext. The decryption functions used in this web page are stored in an external file, `global.php`, which resides in the `system` directory. The obfuscation and the decryption routines detailed later on were analyzed by closely studying these two files.

4. Methodology

In the current section, we formally define the proposed key extraction and malicious traffic detection methods. Let v_1 and v_2 be infected computers running in a controlled environment and belonging to a botnet bt_1. Let $env_{v_1,i}$ and $env_{v_2,i}$ be Zeus *enviornmental items* of v_1 and v_2 respectively — recall that those values are known. Let also ω_{bt_1} be the Zeus *trojan item*, with value and size unknown to us. The plain text of a Zeus *report* issued by v_1 is composed of:

$$R_{v_1} = [header, env_{v_1,0}, \omega_{bt_1}, env_{v_1,1}, ..., env_{v_1,l}]$$

where the *header* is unknown and depends on the *body*, which is composed of $env_{v_1,i}$ and ω_{bt_1}. Analogously, the plain text R_{v_2} is composed of:

$$R_{v_2} = [header, env_{v_2,0}, \omega_{bt_1}, env_{v_2,1}, ..., env_{v_2,l}].$$

Note that the *item* ω_{bt_1} is common to all bots in the botnet bt_1. Moreover, reports issued by bots belonging to the same botnet follow the same structure. Each of these reports is then encrypted in order to obtain packets C and D, respectively issued by v_1 and v_2.
By merging this information with the knowledge acquired during the Zeus Toolkit code analysis, we can identify certain parts of the plain text message, i.e. $env_{v_1,i}$ and $env_{v_2,i}$, that will be encrypted and sent to the C&C. As discussed before, the Zeus malware does not update the RC4 initialization vector, exposing its communications to key reuse attacks. This weakness permitted us to develop a *chosen-plaintext attack*, based on the known *environmental items*, against the encrypted stream that flows between the bot and its C&C. In particular, we were able to retrieve the keystream —from the cypher text—, and to reuse it to detect infected network traffic. To this goal, we focused on the biggest encrypted packet that a Zeus infected computer periodically sends to its C&C, which includes a full report of the stolen information.

4.1. Cryptanalysis

The RC4 is a stream cipher that uses a bit-wise exclusive-or (XOR) operation between the plain text (P) and the keystream (K) generated by a pseudo-random number generation algorithm (PRNG). Note that the XOR is subject to the cancellation propriety:

$$P \oplus K = C; \quad P \oplus C = K.$$

The cipher text C is contained inside an HTTP POST request made by the infected computer to its *dropzone*, therefore the entire data payload needs to be extracted from the network flow before being correctly decrypted. However, the new version of Zeus introduces an obfuscation mechanism that consists in recursively XORing every byte with its previous one, while the first byte of the plain text is simply copied into the obfuscated string. Algorithms 1 and 2 show the pseudo-code of the obfuscation routines `visualEncrypt()` and `visualDecrypt()` used by this malware version.

Algorithm 1 Obfuscation algorithm

```
1: function VISUALENCRYPT($data$)            ▷ The plain text data
2:     for $i \leftarrow 1, |data|$ do
3:         $data_i \leftarrow data_i \oplus data_{i-1}$
4:         $i \leftarrow i{+}{+}$
5:     end for
6:     return $data$                          ▷ The obfuscated data
7: end function
```

Algorithm 2 De-Obfuscation algorithm

```
1: function VISUALDECRYPT($data$)   ▷ The RC4 deciphered data
2:     for $i \leftarrow |data|, 1$ do
3:         $data_i \leftarrow data_i \oplus data_{i-1}$
4:         $i \leftarrow i{+}{+}$
5:     end for
6:     return $data$                          ▷ The plain text data
7: end function
```

Note that, as a result of Algorithm 1, the unknown terms in a Zeus *report header* are propagated through the obfuscated string. However, in the following we prove that a *derivate key* γ can be obtained and used to extract the last m elements of any string ciphered using key K, regardless of the application of Algorithm 1.

Definition 1. Let $P = \{\epsilon_0, ..., \epsilon_{n-1}, p_n, ..., p_{n+m-1}\}$ be a string with $|P| = n + m$ composed by a set of n unknown values ϵ_i followed by a set of m known values p_i. Let K be a cipher key and C the cipher text obtained as $C = Obf(P) \oplus K$. We define a *derivate key* associated to C as $\gamma = C \oplus O'$, where O' is the obfuscation of a vector $P' = \{\varphi_0, ..., \varphi_{n-1}, p'_n, ..., p'_{n+m-1}\}$ with $p'_i = p_i$ for $i \in [n...n+m-1]$ and φ_i taking any known value for $i \in [0...n-1]$.

Theorem 1. *Given a cipher text $Z = Obf(Q) \oplus K$ with $|Z| = n+m$, the last m elements of the string Q can be obtained using the derivate key γ associated to C.*

PROOF. The *derivate key* is defined as $\gamma = C \oplus O'$, where the first term can be expressed as

$$C = Obf(P) \oplus K = \begin{cases} Obf(\epsilon_i) \oplus k_i \\ \quad \text{for } i \in [0...n-1] \\ \\ Obf(\epsilon_{n-1}) \oplus p_n \oplus ... \oplus p_i \oplus k_i \\ \quad \text{for } i \in [n...n+m-1] \end{cases} \tag{1}$$

while O' is obtained as follows

$$O' = Obf(P') = \begin{cases} Obf(\varphi_i) \\ \quad \text{for } i \in [0...n-1] \\ \\ Obf(\varphi_{n-1}) \oplus p'_n \oplus ... \oplus p'_i \\ \quad \text{for } i \in [n...n+m-1] \end{cases} \tag{2}$$

Since $p_i = p'_i$, γ can be rewritten as

$$\gamma = \begin{cases} Obf(\epsilon_i) \oplus Obf(\varphi_i) \oplus k_i \\ \quad \text{for } i \in [0...n-1] \\ \\ Obf(\epsilon_{n-1}) \oplus Obf(\varphi_{n-1}) \oplus k_i \\ \quad \text{for } i \in [n...n+m-1] \end{cases} \tag{3}$$

Let us consider an unknown plain text $Q = \{\lambda_0, ..., \lambda_{n-1}, q_n, ..., q_{n+m-1}\}$. The corresponding cipher text Z can be expressed as

$$Z = Obf(Q) \oplus K = \begin{cases} Obf(\lambda_i) \oplus k_i \\ \quad \text{for } i \in [0...n-1] \\ \\ Obf(\lambda_{n-1}) \oplus q_n \oplus ... \oplus q_i \oplus k_i \\ \quad \text{for } i \in [n...n+m-1] \end{cases} \tag{4}$$

When applying the *derivate key* to Z, we obtain

$$Z \oplus \gamma = \begin{cases} \Delta_i \\ \quad \text{for } i \in [0...n-1] \\ \\ \Delta_{n-1} \oplus q_n \oplus ... \oplus q_i \\ \quad \text{for } i \in [n...n+m-1] \end{cases} \tag{5}$$

where $\Delta_i = Obf(\epsilon_i) \oplus Obf(\varphi_i) \oplus Obf(\lambda_i)$. Finally, values q_i can be obtained for $i \in [n...n+m-1]$ by applying $Obf^{-1}(Z \oplus \gamma)$ to eliminate the constant term Δ_{n-1}.

Note that although terms λ_i cannot be obtained, these are not relevant in order to identify malicious traffic.

4.2. Key Extraction algorithm

One of the aims of this research was to obtain enough parts of γ needed to decipher the Zeus configuration file, which is usually around 69 KBytes [30]. By executing a Zbot sample in our sandbox, it was possible to build a packet containing *chosen-plaintext* by inflating a modified cookie, i.e. here in after *contrast-cookie*, with ordinary strings. The *contrast-cookie* takes a relevant role because it allows us to enrich our *chosen-plaintext* to obtain a bigger known plain text, which implies being able to consecutively retrieve the desired size of γ. On top of that, we were able to extract the last m bytes of the *derivate key* γ and to decipher the configuration file without knowing either the RC4 keystream K or its *seed*. In addition, the proposed approach would also save the researchers from carrying out the static malware analysis to search either the keystream or the seed in the system volatile memory.

Note that, according to Theorem 1, the *derivate key* γ can be applied to any string Z obtained when ciphering a string Q

formed by a set of n unknown values and m known ones with key K; however, in order to obtain γ, it is necessary to establish a correspondence between p_i and p'_i, which requires to know n in advance. As stated in Section 3, a Zeus *report* contains two unknown substrings: the Zeus *header* and the Zeus *trojan item* 10002, i.e. the *botnet-id*. Although the length of the Zeus *header* is known, this is not the case for the Zeus *botnet-id*. Given that the mentioned field has a limited length, we tackle this issue by performing a *brute-force attack* on the Zeus *botnet-id item body* length.

According to the Zeus packet structure explained before, the length of every Zeus *item body* is declared in 4 bytes of its *header*. Thus, the maximum length of the field is 2^{32} bytes long. However, the Zeus Builder toolkit does not allow creation of malwares containing a *botnet-id* value bigger than 20 characters. Therefore, we just need to make an attempt over 20 different positions as to where to insert our known *environmental item* values. Algorithm 3 is in charge of creating the set of all the possible obfuscated texts O', depending on their different lengths. Note that in line 5 the `insert` function is used for adding the character *"A"* at the fixed position[3] x, and shifting all the next values by one byte in the string. The algorithm takes the *chosen-plaintext* as input, and returns a multimodal-array with all the related obfuscated texts.

Algorithm 3 Create Obfuscation Set algorithm

```
1: function CREATEOBFSET(text)                ▷ The chosen plain text
2:     $T \leftarrow text$
3:     $x \leftarrow 112$
4:     for $i \leftarrow 0, 20$ do
5:         insert($T_x$, "A")
6:         $O'_i \leftarrow$ visualEncrypt($T$)
7:         $i \leftarrow i{+}{+}$
8:     end for
9:     return $O'$                             ▷ The obfuscated set
10: end function
```

Once we generate our set of O', we can use it for retrieving the *derivate key* as explained before. The key extraction pseudo code is expressed in Algorithm 4.

- **Lines 1-4** The function EXTRACTKS() is called with three arguments: the intercepted payload C, the *chosen-plain text* P' previously generated, as well as a second payload D, corresponding to another infected computer belonging to the same botnet.
- **Line 5** The set of obfuscated texts O' is created in order to test all possible lengths of Zeus' *botnet-id item*.
- **Lines 6-12** A *derivate keys* matrix S is generated as explained before, where every element S_i corresponds to one possible Zeus *botnet-id item* length.
- **Lines 13-20** In order to discover the appropriate *derivate key*, every key S_i is tested against the second payload D. As a result, a matrix O'' containing all the possible decryptions of D is obtained. Moreover, the entropy of each possible decryption is computed in line 18.
- **Lines 21- 24** The appropriate *derivate key* $S_v = \gamma$ is identified due to the lower entropy E_i resulting of its application to the payload D.

Once a reasonable part of γ is successfully extracted, it is sent to *Cronus* in order to be able to identify Zeus traffic by performing *deep packet inspection*.

Algorithm 4 Key Extraction algorithm

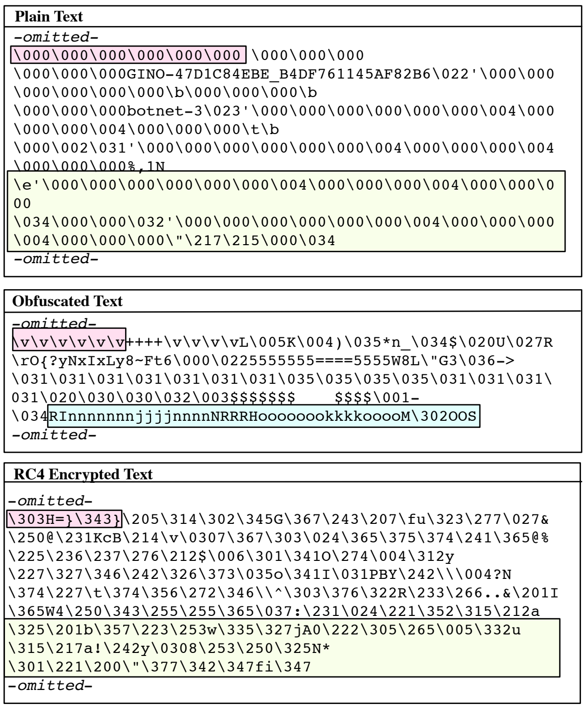
```
1: function EXTRACTKS($payload_1, payload_2, knownPlain$)
2:     $P' \leftarrow knownPlain$
3:     $C \leftarrow payload_1$
4:     $D \leftarrow payload_2$
5:     $O' \leftarrow$ createObfSet($P'$)
6:     for $i \leftarrow 0, |O'|$ do
7:         for $t \leftarrow 0, |O'_i|$ do
8:             $S_i, t \leftarrow O'_{i,t} \oplus C_t$
9:             $i \leftarrow t{+}{+}$
10:        end for
11:        $i \leftarrow i{+}{+}$
12:    end for
13:    for $i \leftarrow 0, |S|$ do
14:        for $t \leftarrow 0, |S_i|$ do
15:            $O''_{i,t} \leftarrow S_{i,t} \oplus D_t$
16:            $i \leftarrow t{+}{+}$
17:        end for
18:        $E_i \leftarrow$calcEntropy($O''_i$)
19:        $i \leftarrow i{+}{+}$
20:    end for
21:    $v \leftarrow$ calcMin($E$)
22:    $\gamma \leftarrow S[v]$
23:    return $\gamma$
24: end function
```

4.3. Cronus: an IDS for Zeus

Our research further demonstrates how the entropy of these obfuscated messages O remains nearly constant, allowing us to automatically identify malicious traffic even if it is not fully deobfuscated. This propriety derives from the *zero padding strings* used by Zeus for composing its messages. As can be seen in Figure 3, when these strings are obfuscated by the `Cronus()` routine, the resulting output contains several text parts with repeated characters (these parts are evidenced in the cited figure). As a result, the calculated entropy drastically decreases in comparison with the one calculated for the cipher text.

Algorithm 5 illustrates the pseudo code of *Cronus*, the proposed Zeus IDS. The algorithm works as follows: a) once the suspicious HTTP payload has been filtered out, it is passed as an argument to the MAKEITPLAIN() function. b) The Shannon entropy H of the message is then calculated and, if it is higher than the threshold α, this means that the analyzed payload C

[3]This position depends on the length of the Zeus *item* id 10001


Plain Text

```
-omitted-
\000\000\000\000\000\000 \000\000\000
\000\000\000GINO-47D1C84EBE_B4DF761145AF82B6\022'\000\000
\000\000\000\000\b\000\000\000\b
\000\000\000botnet-3\023'\000\000\000\000\000\000\004\000
\000\000\004\000\000\000\t\b
\000\002\031'\000\000\000\000\000\000\004\000\000\000\004
\000\000\000%,1N
\e'\000\000\000\000\000\000\004\000\000\000\004\000\000\0
00
\034\000\000\032'\000\000\000\000\000\000\004\000\000\000
\004\000\000\000\"\217\215\000\034
-omitted-
```

Obfuscated Text

```
-omitted-
\v\v\v\v\v\v++++\v\v\v\vL\005K\004)\035*n_\034$\020U\027R
\rO{?yNxIxLy8~Ft6\000\0225555555====5555W8L\"G3\036->
\031\031\031\031\031\031\031\035\035\035\035\031\031\031\
031\020\030\030\032\003$$$$$$$    $$$$\001-
\034RInnnnnnnjjjjnnnnNRRRHoooooookkkkooooM\302OOS
-omitted-
```

RC4 Encrypted Text

```
-omitted-
\303H=}\343}\205\314\302\345G\367\243\207\fu\323\277\027&
\250@\231KcB\214\v\0307\367\303\024\365\375\374\241\365@%
\225\236\237\276\212$\006\301\341O\274\004\312y
\227\327\346\242\326\373\035o\341I\031PBY\242\\\004?N
\374\227\t\374\356\272\346\\^\303\376\322R\233\266..&\201I
\365W4\250\343\255\255\365\037:\231\024\221\352\315\212a
\325\201b\357\223\253w\335\327jA0\222\305\265\005\332u
\315\217a!\242y\0308\253\250\325N*
\301\221\200\"\377\342\347fi\347
-omitted-
```

Figure 3: Zeus data through the three deciphering phases

is encrypted. Hence, the algorithm can continue its flow[4]. c) The function continues with the `RC4()` routine which XOR-decrypts C with all the γ values previously collected and, for each decrypted output O, the routine `calcentropy()` is called to calculate its entropy H'. d) If H' is lower than β, it means that O is likely to be a Zeus obfuscated message. As said, there is no need to deobfuscate O in order to confirm the detection.

It is important to note that the condition $|\gamma_i| \geq |C|$ should be verified before calling the `Cronus` function. Because of this, we are able to identify only those packets with cardinality less or equal than the collected key one, meaning that the key should be big enough (300 Bytes at least) to correctly decrypt a Zeus *log* message and determine its maliciousness. By considering that this could raise a performance issue for *Cronus* due to collecting several keys and using them to analyze each HTTP POST packet, we focused on only searching for short and delimited parts of Zeus packets and analyze them accordingly.

We found two different patterns that, once encrypted and obfuscated, always maintain certain contents at the same position, and keep their entropy values constant. In order to find a pattern useful to identify Zeus traffic, we need to use the *static* parts of its messages with contents changing as little as possible, and that are always located at the same position of the Zeus message. Given these constraints, we focused on the 12 bytes reserved for the Zeus *header info*, and the 16 bytes reserved for the first Zeus *item header*. While other *item header*'s positions

[4]Note that from Information Theory we know that the Shannon entropy is not increased by more than the length of the encryption key. However, dealing with RC4 encryption, the encryption key is as long as the plaintext is. Therefore, our choice of the Shannon entropy to determine whether a message is encrypted is meaningful.

Bytes range	Content	Binary values
48 – 51	\021'\000\000	1001
52 – 55	\000\000\000\000	0
56 – 59	\000\000\000	32
59 – 63	\000\000\000	32

Table 2: Zeus ItemID 1001 header

Zeus message segment	Bytes range	n. of *null bytes*
Zeus *header info*	20 – 31	5
Zeus *item* 1001 *header*	48 – 63	12

Table 3: Pattern used by *Cronus*

are variable, i.e. depend on the size of the previous *item body*, these headers can always be found at the same position of the TCP packet's payload, i.e. between byte 20 and byte 31, and between byte 48 and byte 63 respectively.

Pattern 1: *Zeus header info* – The 12 bytes long Zeus *header info* contains information about the Zeus body size, the item flags, and the number of items. The body size of a typical Zeus log message is around 300 bytes, thus, by considering that the maximum number of bits used to represent this number is 17, and that four bytes are used for encoding the Zeus body size, in the worst case one entire byte will always be set to *zero*. Next, the four bytes long Zeus *item flags* follow a certain structure: the first byte is used as a binary selector in order to inform if the packet is compressed or not, the second byte is used for instructing the control panel about how to manage the Zeus *items* if they are encrypted, while the third byte is used for reducing the risk of item overlapping. Finally, the last byte is never used, and is always set to zero. Although we can definitely assert that in the worst case one byte will be always set to *null*, it is interesting to note that all the Zeus *log* messages that we analyzed during our research, always kept these four bytes equal to zero. Lastly, by considering that in the Zeus control panel source code only 45 different items are encoded, only one byte of the four allocated will be enough to represent the number of the items contained inside a Zeus message.

Pattern 2: *Zeus item header*– Recalling the Zeus packet structure exposed in Section 3, the 16 bytes long Zeus *item header* contains the *itemID*, four *null bytes*, and an identical pair of four bytes containing the *item body*'s length. Therefore, as we have already showed, 2^{32} bits are used to express the item's length in bytes. The *botID* is composed of the infected computer's NetBios name, which can be 15 byte length at maximum, and a 17 bytes long string randomly generated by the malware in order to assure a unique *botID*. Hence, the Zeus *item body's* length can be 32 bytes at maximum and only one of the four allocated bytes may be used to express this information, while the other three bytes will always contain zero values. As showed in Table 2, the first Zeus *item header* contains several *null bytes*.

Table 3 summarizes the pattern values by highlighting the number of *null bytes* present in the worst case. Due to their nature, these headers always contain several zero strings that drastically decrease the string entropy value. Notably, by us-

Var	Entropy value	Explanation
α	≥ 7.3	Encrypted text
β	6.0-6.6	Obfuscated text
δ	≤ 4.8	Plain text

Table 4: Entropy thresholds used during the experimental phase

ing more bytes than required, the malware coder compromised its own obfuscation algorithm by making its output easily detectable. Based on these findings, our research demonstrates that the proposed IDS can identify certain Zeus traffic with only 36 bytes long keys. In this way, the performance of *Cronus* storing and analyzing all the captured HTTP POST requests significantly increases in comparison to storing 1500 bytes long keys.

Finally, once the Zeus bot communication is identified, several further actions could be taken. For instance, the traffic could be blocked by a network firewall. This can be easily achieved with *Cronus* sending the information needed to update the firewall rules in order to deny all the traffic directed to the Zeus C&C. The α and β entropy thresholds used during our experiments are shown in Table 4. These thresholds have been set after a learning phase which included calculating the entropy of 100 Zeus plain texts with their corresponding obfuscated and encrypted transformation. Note that these values are completely customizable.

Algorithm 5 Cronus algorithm

```
 1: function CRONUS($payload$)                         ▷ HTTP POST request
 2:     $H \leftarrow$ calcEntropy($payload$)
 3:     if $H \geq \alpha$ then                          ▷ The payload is encrypted
 4:         for $i \leftarrow 1$, sizeof $\Gamma$ do
 5:             $C \leftarrow payload$
 6:             $O \leftarrow$ RC4($C, \gamma_i$)
 7:             $H' \leftarrow$ calcEntropy($O$)
 8:             if $H' \leq \beta$ then                    ▷ Zeus obfuscated data
 9:                 return 1                           ▷ Zeus obfuscated data
10:             else
11:                 return 0                           ▷ data not decrypted
12:             end if
13:             $i \leftarrow i{+}{+}$
14:         end for
15:     end if
16: end function
```

5. Experimental Setting

The different phases of our experiment are described in the following section. In order to test *Cronus*, we set up a testing environment, which comprises an in-house Zeus botnet that includes a C&C, a *dropzone*, two computers to infect (v_1 and v_2), and a web server where a fake banking webpage is running. Once the testing environment was configured, we created a Zeus trojan binary and executed it in v_1 and v_2. Furthermore, we analyzed the generated network traffic in order to extract certain parts of the *derivate key*. Finally, we used those keys to develop *Cronus*, an IDS able to detect Zeus traffic in a production network.

5.1. Set up of the testing environment

In the first phase, we recreated a complete Zeus environment in an isolated network. This task has been accomplished by customizing the *Dorothy framework* [11] that is currently employed for similar research initiatives. Such a framework is built upon a VMWare ESXi infrastructure that is automatically piloted through the *Dorothy Console*, i.e. a Ruby gem installed on an external machine which is in charge to start, stop and revert the virtual machines, besides running executables inside them once transferred. The framework comes with a network analysis module (*NAM*) which relies on a *pcapr-local* [31] instance that is dedicated to dissecting and storing the analyzed traffic into a non-SQL database for allowing fast and indexed searches. Such a module is in charge of recording and dissecting all the network traffic generated by the sandboxes for the whole execution of the VMs.

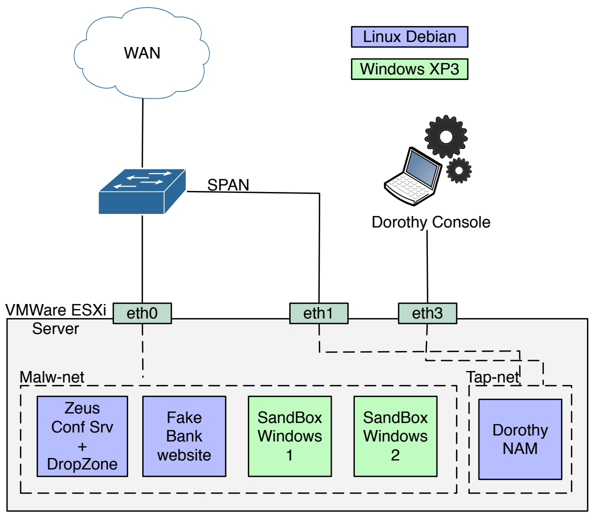


Figure 4: Test environment used during the experiment

Hence, we configured four different virtual machines: a Linux Debian distribution was used for configuring the Zeus control panel and its *dropzone* and two Windows virtual machines were selected as sandboxes for the malware execution. Finally, another Linux-based VM was used for running a fake bank website. The Windows sandboxes come with a Windows XP SP3 default installation, with Internet Explorer 7 installed on it. In addition, a *contrast-cookie*, associated to the local fake bank website was inserted in the default IE system cookies folder `/Documents and Settings/userhome/Cookies` of both sandboxes, and modified in order to contain a large quantity of repeated strings (we tested a *contrast-cookie* up to 700 Kbyte large). The fake web server ran on a TomCat server, which replicated the login page of a known home banking website. The local DNS server was configured consequentially in order to resolve the real bank site URL with the local IP address. A summarized scheme of the test environment is presented in Figure 4.

Finally, we used the provided Zeus Builder to create a new malware sample mw_1 and its relative configuration file $conf_1$ and moved them to the sandboxes and to the *dropzone* respectively. The configuration file contains the local IP addresses of our testing environment as *dropzone* and C&C. Through this file, we configured our bots v_1 and v_2 in order to communicate with the C&C every two minutes, and to send the full system *reports* every four minutes. Notably, these *reports* contain all the system cookies, including our inflated *contrast-cookie*.

5.2. Malware execution and Key extraction

The whole malware analysis process is described in Figure 5. The process begins by transferring mw_1 to the sandboxes, and executing it with administrative privileges. We define the *zombified timeframe* Δt_e as the interval Δt between the time of the malware execution and the time when the virtual machines v_1 and v_2 are reverted to their not malicious activities.

During Δt_e, the *NAM* records all network traffic generated by the infected VMs, and stores it in a PCAP file. At the early age of its infection, the infected systems attempt to retrieve the encrypted configuration file by making HTTP GET requests to the Zeus C&C (Step 1). Notably, the C&C IP address/domain name is hard coded inside the malware binary, and it could be revealed by conducting a binary static analysis. Just after the execution of the malware, both sandboxes are automatically driven to visit the fake bank website, and to enter fake login credentials (Step 2). Web automation is accomplished by executing a customized macro of the iMacros plugin [32], which successfully emulates the usual human internet browsing activity.
After $conf_1$ is downloaded, each infected system deciphers it and extracts the information related to the botnet *dropzone* domain name/IP address. At this point, the bot begins to make two different types of HTTP POST requests to the C&C (Step 3), i.e. *Zeus logs* and *reports*. Differently from the first type of request, the size of the data transmitted in the Zeus *report* depends on the information stored in the infected computer. As previously outlined, we leveraged this propriety to inflate our *contrast-cookie* in order to extract enough parts of the keystream needed to further decrypt $conf_1$. An example of a decrypted report message (where relevant parts are highlighted), can be seen in Figure 6.

After Δt_e, the sandbox virtual machines are reverted to their original state, and the saved network dump is analyzed inside the *NAM* by dissecting it and extracting all the network flows recorded during the analyzed time (Step 4). Consecutively, the *Dorothy Console* analyzes them by filtering out only those flows related to HTTP traffic directed to the local Zeus C&C, and then extracts the payload of the largest ones, which supposedly contain the stolen cookies of v_1 and v_2. Once the ciphered flows have been filtered out, they are used for extracting the *derivate key* as explained in Section 4 (Step 5).

Finally, the returned keystream is used for decrypting the encrypted configuration file, and if the retrieved output gives an acceptable entropy ($H' \leq \beta$), then the tuple $< mw_i, \gamma_i >$ can be sent to the *Cronus* database (Step 6). Notably, this operation is

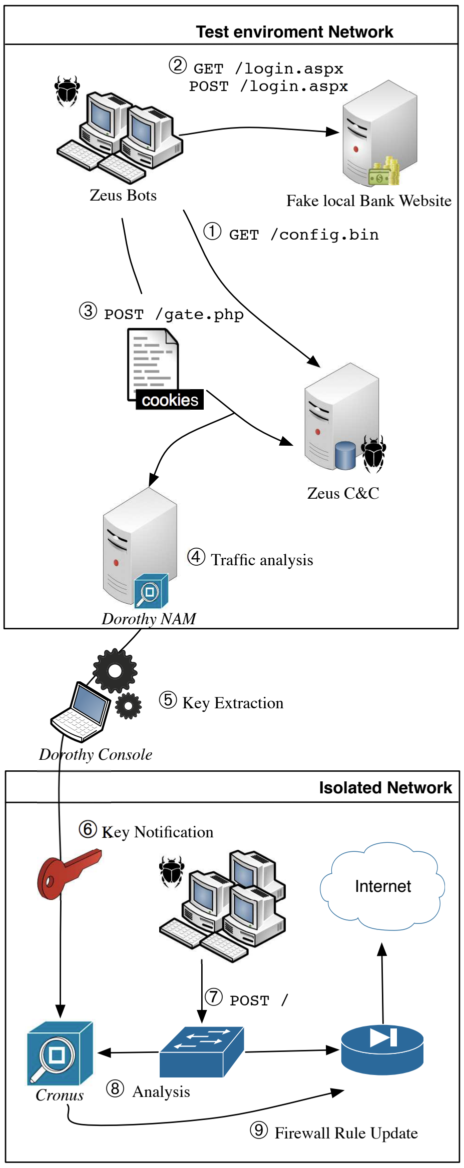
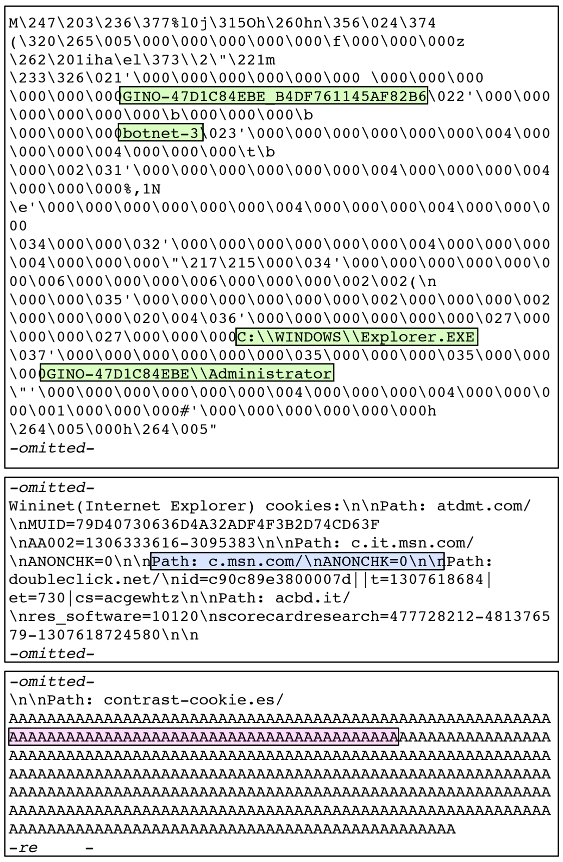


Figure 5: Analysis Flow

accomplished by the *Dorothy Console* which connects the *NAM* with the production network, e.g. where *Cronus* resides.

5.3. Malicious traffic identification

The last phase consists of detecting certain Zeus traffic in a production network, having a set of valid Zeus keystreams $\Gamma = \{\gamma_0, \gamma_1, ..., \gamma_i\}$, obtained as detailed in the previous subsection. To perform this test, we set up five different virtual machines $V = \{v_0, ..., v_4\}$ in an isolated network (Step 7), which constantly generate common network traffic towards a fixed hosts range $S = \{s_0, s_2, ..., s_i\}$, e.g. web browsing and POP mail consulting. Next, we infected one of them with a previously created Zeus malware sample mw_1, and recorded all network traffic generated by this machine during Δt_e (Step 7). Another virtual machine is configured as network sniffer in order to analyze all network traffic going through the network. This latter VM also executes our Proof of Concept (PoC) for *Cronus*,



```
M\247\203\236\377%l0j\315Oh\260hn\356\024\374
(\320\265\005\000\000\000\000\000\f\000\000\000z
\262\201iha\el\373\\2\"\221m
\233\326\021'\000\000\000\000\000\000 \000\000\000
\000\000\000GINO-47D1C84EBE_B4DF761145AF82B6\022'\000\000
\000\000\000\000\b\000\000\000\b
\000\000\000botnet-3\023'\000\000\000\000\000\000\004\000
\000\000\004\000\000\000\t\b
\000\002\031'\000\000\000\000\000\000\004\000\000\000\004
\000\000\000%,1N
\e'\000\000\000\000\000\000\004\000\000\000\004\000\000\0
00
\034\000\000\032'\000\000\000\000\000\000\004\000\000\000
\004\000\000\000\"\217\215\000\034'\000\000\000\000\000\0
00\006\000\000\000\006\000\000\000\002\002(\n
\000\000\035'\000\000\000\000\000\000\002\000\000\000\002
\000\000\000\020\004\036'\000\000\000\000\000\000\027\000
\000\000\027\000\000\000C:\\WINDOWS\\Explorer.EXE
\037'\000\000\000\000\000\000\035\000\000\000\035\000\000
\000GINO-47D1C84EBE\\Administrator
\"'\000\000\000\000\000\000\004\000\000\000\004\000\000\0
00\001\000\000\000#'\000\000\000\000\000\000h
\264\005\000h\264\005"
-omitted-
```

```
-omitted-
Wininet(Internet Explorer) cookies:\n\nPath: atdmt.com/
\nMUID=79D40730636D4A32ADF4F3B2D74CD63F
\nAA002=1306333616-3095383\n\nPath: c.it.msn.com/
\nANONCHK=0\n\nPath: c.msn.com/\nANONCHK=0\n\nPath:
doubleclick.net/\nid=c90c89e3800007d||t=1307618684|
et=730|cs=acgewhtz\n\nPath: acbd.it/
\nres_software=10120\nscorecardresearch=477728212-4813765
79-1307618724580\n\n
-omitted-
```

```
-omitted-
\n\nPath: contrast-cookie.es/
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
-re     -
```

Figure 6: Information sent to Zeus dropzone

which is designed to scrub all HTTP POST requests identified in the network flow (Step 8), and trigger an alert whenever a malicious pattern is detected (Step 9). It is relevant to note that the network flows analyzed by *Cronus* discarded HTTPS traffic, otherwise the algorithm would waste time and resources by trying to decrypt a non-RC4 traffic. Additionally, other filtering proprieties, e.g. white lists, can be added to reduce false positives and to avoid unnecessary charge on *Cronus* that can be generated by analyzing legitimate traffic. Finally, if *Cronus* returns positive results for an analyzed network flow, we can claim that v_i has been infected by mw_1, and that s_i is a specific Zeus C&C which has been proved to be online during during Δt_e.

The *Cronus* PoC was developed to decipher all the suspicious traffic identified in the network traffic. Since the Zeus botnet is based on the HTTP protocol, and the stolen information is sent to the *dropzone* through HTTP POST requests, the following is recognized as *suspicious* traffic and analyzed by *Cronus*:

1. TCP traffic;
2. An HTTP POST request containing an encrypted body.

While filtering a TCP network stream is quite easy, filtering HTTP POST requests requires analyzing the upper lever of the TCP protocol. This technique is commonly referred as *deep packet inspection* and requires a fast CPU in order to process the network flow within a reasonable time. The proposed *Cronus* PoC was developed in Ruby language, and can be deployed and executed on any system with the Ruby Framework installed on it.

The next section reports on the results of our experiment.

6. Evaluation

In the following we describe the applicability of the proposed approach inside a corporate network during a regular working day. The goal of this experiment was to demonstrate that the proposed approach could be used in a real scenario, by offering to the network administrator the capability to detect Zeus infected systems.

6.1. Environment configuration

As first step we created a customized Zeus Trojan V. 2.0.8.9 by using the toolkit leaked on Internet. The malware was configured in order to communicate to our internal server where a Zeus control panel was previously set up. In addition, we defined an interval of 2 minutes for the Zeus *log* messages, and 10 minutes for the *report* ones. The experiment began by executing an instance of `tcpdump` in a machine which was physically connected to a SPAN port of the corporate router. In order to limit the impact of the experiment, the SPAN port was configured in order to redirect only the network traffic[5] belonging to a VLAN composed by 13 hosts, i.e. 11 desktop computers used by researchers for their daily job, and 2 virtual machines dedicated for the experiment's purpose. *Cronus* was executed in a Linux Debian Etch machine, with 2Gb of dedicated memory and a 2.6Ghz CPU.

The sniffer began to record the local traffic at 9:50 AM[6], and continued its activity for the next 5 hours. At 11:35 AM, a Windows XP SP3 virtual machine inside the same network was infected with the previously generated malware, and an instance of iMacros was configured to mimic a typical web browsing activity by executing a macro which starts from the Google News web page and browse its link every 10 minutes. At 3:00 PM, the `tcpdump` instance was stopped, and overall 17.50 GB of network traffic was correctly collected.

The malware keystream was previously extracted by using the explained approach, and it was stored into a text file which was already containing 28 different keys. In addition, 200 more different keys were inserted into the key file, in order to assess the computational overhead experienced by *Cronus* during the keys loading function

Once the network dump was generated, it was moved to the *Cronus* machine. *Cronus* was configured to load a file containing 229 different keys but to test only the first 30 in order to simulate a real case scenario[7].

[5]The entire experiment was conducted in accordance with the Spanish laws on Privacy. Every user involved in the experiment had been previously informed about the experiment purpose, the chain of custody of the generated data, and the data retention period. It is important to note that every user was free to decide to take part of the experiment or not, by signing an informative document.

[6]In the following, times refer to Central European Time (GMT+1).

[7]We decided to test only 30 keys because at the time of this writing, there are only 26 Zeus active domains according to Zeus Tracker[30].

6.2. Results

Cronus analyzed 7,133,170 TCP packets finding 5,990 HTTP POST requests, 141 of them containing an encrypted payload. For each of these, 30 different keys were tried in order to decipher the payload and retrieve a reasonable entropy result. As result, 138 HTTP POST requests were correctly identified as Zeus traffic.

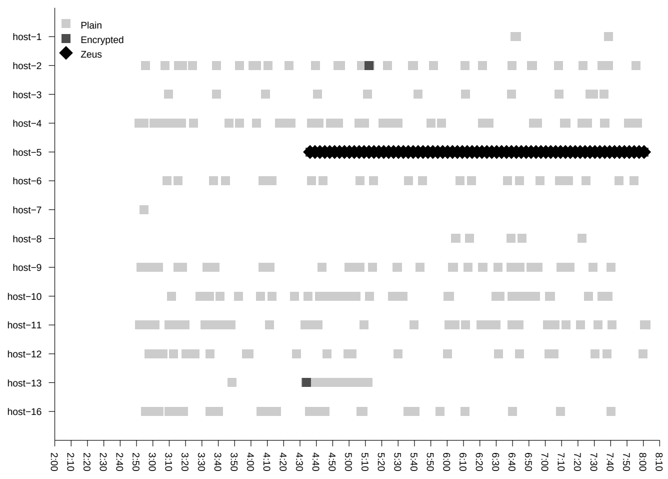


Figure 7: Analyzed HTTP POST requests

Figure 7 shows that the three missing encrypted POST requests didn't belong to the infected machine (`host-5`), thus are not false negatives. However, after a manual analysis of the packets sent by the infected machine, we discovered that it generated 139 HTTP POST packets sent to the Zeus C&C. The packet which was not detected by *Cronus* belonged to the Zeus *report* communication flow, which contains all the cookies of the infected system. As explained before, this packet is typically larger than the network MTU, and packet fragmentation is often needed to correctly send it. As a matter of fact, the TCP segment was 5785 bytes large, and it was split into 9 frames. The first frame, contained only the HTTP POST header, and it was correctly detected by *Cronus*. However, the following ones were not detected because they did not contain the *POST* keyword inside their body — as shown, *Cronus* only considers these packets for its purpose. It is important to note that big Zeus *report* packets are definitely less frequent than the *log* ones. Hence, taking into account *packet reassembling* would not substantially enhance the detection rate, while drastically decreasing the analysis performance.

Figure 8 highlights the frequency of all the HTTP POST requests sent by the most active hosts. Note that `host-5` has a constant request rate, that identifies the typical Zeus *log message* activity. Interestingly, this graph could help a system administrator to identify suspicious Zeus traffic by only mapping the host HTTP POST requests.

The time of the analysis was of 5.39 minutes, which means that the estimated throughput of the proposed IDS is around 400 Mbit/s. The performance of *Cronus* strictly depends on the number of the keys tested, and on the network dump file size. However, as shown in Figure 9, the number of tested keys does

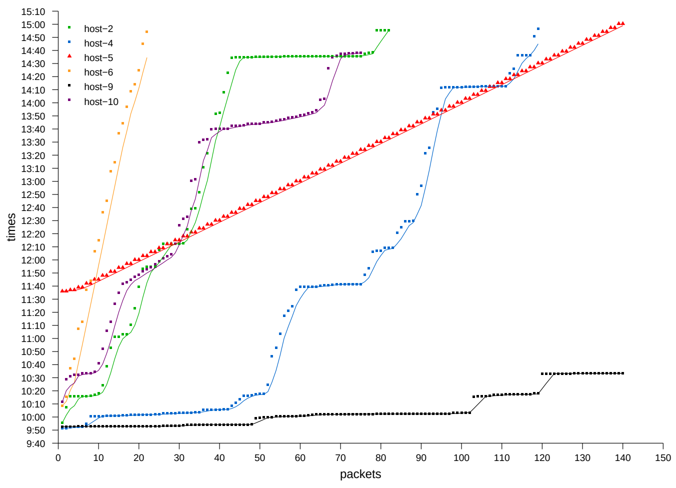


Figure 8: HTTP POST requests per host

not drastically impact on the execution time[8] if we consider that there are less than 90 different Zeus botnets spotted in the wild. However, as the file size grows, the execution time increases as well, according to a linear relationship.

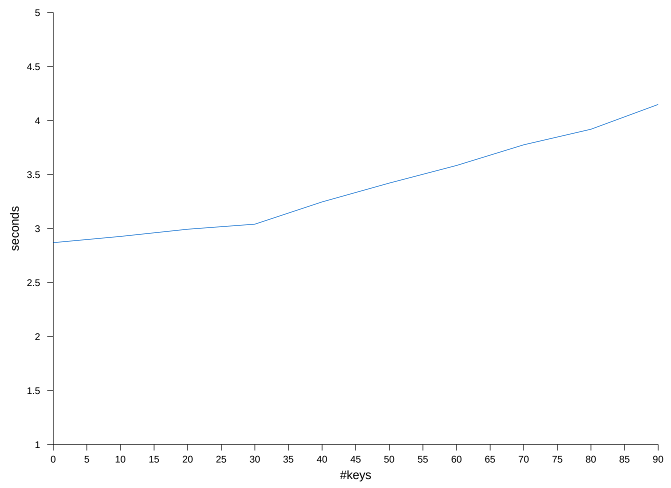


Figure 9: Execution time per number of used keys

7. Lesson learned

One of the main contributions of this paper, apart from the detailed techniques outlined so far, is the fact that we have highlighted a general methodology to attack a class of malware that can be considered generic enough to account for a reasonable part of the total amount of malware posing a serious threat to the Internet. The approach can be essentially decomposed into the following steps: 1. Isolating crypto modules of the malware from the rest of the application; 2. Analyzing the

[8]The execution time values has been calculated by executing *Cronus* on a 16,5 Mb large PCAP.

crypto modules as for their operating procedures; 3. Discovering flaws in the crypto modules. Note that the weaknesses we have been investigating concern the composition of the crypto module with the rest of the application; we did not focus on breaking the crypto algorithm. In our investigation, we have not devoted resources to make an attempt to break the RC4 implementation that the crypto module is based upon. Instead, with the above highlighted approach, we have discovered two weaknesses: a. the crypto module is subject to a re-inizalitation attack; b. the plain text to be encrypted can be provided as input to the algorithm. These two weaknesses, combined with the fact that the one time pad encryption (based on the keystream generated by the RC4) is subject to the cancelation property, have paved the way to our attack in recovering a relevant portion of the keystream. It may be legitimate to think that another weakness is represented by the short length of the *botnet-id* field. Although a bigger *botnet-id* would certainly impact the keystream's recovered fragments, it would also imply several drawbacks for the C&C server management: this unique field is used to update botnet information for every incoming packet, thus doing a MySQL distinct query by filtering a —let us suppose— 2^8 bytes long *botnet-id* would be cumbersome in terms of system resources. In addition, Zeus *log* communications length between the bot and its C&C, would drastically increase, hence making them easier to detect[9].

The sequel of our technique and proposed architecture has just been an exercise in security engineering and secure network design. Interestingly, the attack we proposed is just a reviewed version of the Crib-based one used at Bletchley Park while breaking the Enigma crypto system [33]: the known Zeus *environmental items* represented our *cribs* and the proposed key extraction algorithms our *bombe*. We can definitely assert that old-fashion sound cryptanalysis still works, even 70 years later.

Finally, it is important to note that while the proposed technique is unique to Zeus malware versions prior to 2.0.8.9 and to some later variants that still continue to use the same crypto routine, trends of malware flaws in the implementation and use of crypto algorithms (like the exploited one) could be potentially used as directions for the reverse engineering of other malware in the wild.

8. Conclusions and Future Work

In this paper we have showed a complete solution to detect certain families of Zeus, one of the most dreadful financial malwares. In particular, we have exposed a technique that allows to extract the keystream used by Zeus to encrypt its payload. Based on these results, an IDS to detect Zeus (*Cronus*) has been detailed, and it has been experimentally tested on a production network. Excellent performance and effectiveness results achieved by *Cronus* support our findings. Finally, we have reported on lesson learning and highlighted future work.

[9]Readers who might be interested in finding out more about the countermeasures which could be used to avoid the exposed detection mechanism may contact us. As a result of an internal discussion it was decided to not disclose those techniques in this paper to avoid them being used by malware developers.

Acknowledgements

This work has been carried out with the support of ACC1Ó, the Catalan Business Comptitiveness Support Agency. We woud like to thank "CaixaBank" for the insightful information support for the development of this research, and to Marco Cremonini, Zafar Kazmi and Nelson Madrigal for their precious comments and suggestions.

[1] Norton, Cybercrime report 2011, Tech. rep., Symantec (2011) [cited 19/12/2011].
URL `http://us.norton.com/content/en/us/home_homeoffice-/media/pdf/cybercrime_report/Norton_USA-Human%20Impact-A4_Aug4-2.pdf`

[2] Damballa, Top-10 botnet outbreaks in 2009 [cited 19/12/2011].
URL `http://blog. damballa.com/?p=569`

[3] Trusteer, Banking malware zeus successfully bypasses anti-virus detection [cited 31/10/2011].
URL `http://www.trusteer.com/company/press/-trusteer-warns-zeus-trojan-bypasses-date-anti-virus-systems-77-percent-time`

[4] S. Golanov, TDL4 – Top Bot [cited 19/12/2011].
URL `http://www.securelist.com/en/analysis/204792180/TDL4_Top_Bot`

[5] E. Florio, K. Kasslin, Your computer is now stoned(...again!), Virus Buletin.

[6] M. Sharif, A. Lanzi, J. Giffin, W. Lee, Impeding malware analysis using conditional code obfuscation, in: Network and Distributed System Security (NDSS), Citeseer, 2008.

[7] M. Chandrasekaran, R. Chinchani, S. Upadhyaya, Phoney: Mimicking user response to detect phishing attacks, in: Proceedings of the 2006 International Symposium on on World of Wireless, Mobile and Multimedia Networks, IEEE Computer Society, 2006, pp. 668–672.

[8] D. Birk, S. Gajek, F. Gr
"obert, A. Sadeghi, A forensic framework for tracing phishers, IFIP Summer School on The Future of Identity in the Information Society, Karlstad, Sweden.

[9] L. Spitzner, Honeytokens: The other honeypot, Security Focus 21.

[10] P. Bacher, T. Holz, M. Kotter, G. Wicherski, Know your enemy: Tracking botnets, The Honeynet Project.

[11] M. Cremonini, M. Riccardi, The Dorothy Project: An Open Botnet Analysis Framework for Automatic Tracking and Activity Visualization, in: Proceedings of the 5th European Conference on Computer Network Defense (EC2ND), IEEE, 2010, pp. 52–54.

[12] M. Riccardi, D. Oro, J. Luna, M. Cremonini, M. Vilanova, A framework for financial botnet analysis, in: eCrime Researchers Summit (eCrime), 2010, IEEE, 2011, pp. 1–7.

[13] J. Caballero, N. M. Johnson, S. Mccamant, D. Song, Binary code extraction and interface identification for security applications, in: In ISOC NDSS'10, 2010.

[14] F. Leder, P. Martini, A. Wichmann, Finding and extracting crypto routines from malware, in: Performance Computing and Communications Conference (IPCCC), 2009 IEEE 28th International, IEEE, 2009, pp. 394–401.

[15] F. Leder, P. Martini, Ngbpa next generation botnet protocol analysis, Emerging Challenges for Security, Privacy and Trust (2009) 307–317.

[16] Y. Kim, H. Youm, A new bot disinfection method based on dns sinkhole, Journal of KIISC 18 (2008) 107–114.

[17] S. Ji, C. Im, M. Kim, H. Jeong, Botnet detection and response architecture for offering secure internet services, in: Security Technology, 2008. SECTECH'08. International Conference on, IEEE, 2008, pp. 101–104.

[18] Y. Kim, D. Lee, J. Choi, H. Youm, Preventing botnet damage technique and its effect using bot dns sinkhole, Journal of KISS (C): Computing Practices 15 (1) (2009) 47–55.

[19] G. Gu, P. Porras, V. Yegneswaran, M. Fong, W. Lee, Bothunter: Detecting malware infection through ids-driven dialog correlation, in: Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium, USENIX Association, 2007, p. 12.

[20] X. Jiang, X. Wang, D. Xu, Stealthy malware detection through vmm-based out-of-the-box semantic view reconstruction, in: Proceedings of the 14th ACM conference on Computer and communications security, ACM, 2007, pp. 128–138.

[21] T. Ormerod, L. Wang, M. Debbabi, A. Youssef, H. Binsalleeh, A. Boukhtouta, P. Sinha, Defaming botnet toolkits: A bottom-up approach to mitigating the threat, in: 2010 Fourth International Conference on Emerging Security Information, Systems and Technologies, IEEE, 2010, pp. 195–200.

[22] R. Ford, S. Gordon, Cent, five cent, ten cent, dollar: hitting botnets where it really hurts, in: Proceedings of the 2006 workshop on New security paradigms, ACM, 2006, pp. 3–10.

[23] H. Binsalleeh, T. Ormerod, A. Boukhtouta, P. Sinha, A. Youssef, M. Debbabi, L. Wang, On the analysis of the zeus botnet crimeware toolkit, in: Privacy Security and Trust (PST), 2010 Eighth Annual International Conference on, IEEE, 2010, pp. 31–38.

[24] M. Riccardi, R. Di Pietro, J. Aguila, Taming zeus by leveraging its own crypto internals, in: eCrime Researchers Summit (eCrime), 2011, IEEE, 2012, to appear.
URL `availabe at: http://ricerca.mat.uniroma3.it/users/dipietro/eCrime11.pdf`

[25] S. S. Corporation, M. Ligh, [prg] malware case study, Tech. rep., Secure Science Corporation (2006).

[26] R. Howard, Cyber Fraud: Tactics, Techniques and Procedures, 1st Edition, Auerbach Publications, Boston, MA, USA, 2009.

[27] B. Krebs, Zeus source code for sale. got 100,000 dollars? (2011) [cited 19/12/2011].
URL `http://krebsonsecurity.com/2011/02/zeus-source-code-for-sale-got-100000/`

[28] J. Manuel, Another modified zeus variant seen in the wild [cited 19/12/2011].
URL `http://blog.trendmicro.com/another-modified-zeus-variant-seen-in-the-wild/?awid=7917255160271489866-1985`

[29] G. Inc., Google translate service (2011) [cited 19/12/2011].
URL `http://translate.google.com/`

[30] Abuse.ch, The swiss security blog [cited 19/12/2011].
URL `https://zeustracker.abuse.ch/statistic.php`

[31] MuDynamics, Annuncing pcapr-local [cited 19/12/2011].
URL `http://blog.mudynamics.com/2011/04/18/announcing-pcaprlocal/`

[32] iOpus, imacros plugin (2011) [cited 19/12/2011].
URL `http://www.iopus.com/imacros/`

[33] D. Kahn, The Codebreakers: the story of secret writing, Scribner Book Company, 1996.