NGBPA
Next Generation BotNet Protocol Analysis

Felix S. Leder and Peter Martini

Abstract The command & control (c&c) protocols of botnets are moving away from plaintext IRC communicationt towards encrypted and obfuscated protocols. In general, these protocols are proprietary. Therefore, standard network monitoring tools are not able to extract the commands from the collected traffic. However, if we want to monitor these new botnets, we need to know how their protocol decryption works. In this paper we present a novel approach in malware analysis for locating the encryption and decryption functions in botnet programs. This information can be used to extract these functions for c&c protocols.
We illustrate the applicability of our approach by a sample from the *Kraken* botnet. Using our approach, we were able to identify the encryption routine within minutes. We then extracted the c&c protocol encryption and decryption. Both are presented in this paper.

1 Introduction

Botnets have been a major, growing threat in the Internet in the last years. Today, botnets are the source of more than 90% of all SPAM mails. They collect email addresses, passwords and sometimes even banking information. In addition, botnets have the ability to coordinate and conduct distributed denial of service attacks.

While the core functionality and behavior of malware is quite stable, obfuscation and polymorphic techniques[21] are used to circumvent signature detection. As

Felix S. Leder
University of Bonn, Institute of Computer Science IV, Roemerstr. 164, 53117 Bonn, Germany, e-mail: leder@cs.uni-bonn.de

Peter Martini
University of Bonn, Institute of Computer Science IV, Roemerstr. 164, 53117 Bonn, Germany, e-mail: martini@cs.uni-bonn.de

a consequence, only behavioral analysis can be used to classify a given malware specimen.

The state-of-the-art method of classifying botnets is to run the bot in a monitored environment and analyze the behavior. The network traffic is a very reliable way to classify specimen to specific families. For commonly used protocols like IRC and HTTP, there is a wide range of automated analysis and monitoring tools[20, 23]. These tools are very reliable for known protocols but fail for encrypted traffic.

Most botnets are sticking to traditional IRC communication [22] but more and more botnets are moving towards “stealthier” and robust communication. This includes P2P protocols as well as obfuscated and encrypted protocols[10, 12]. In order to extract information from collected network data, the encryption and decryption has to be known and added to the monitoring tools.

The botnet software itself contains those encryption and decryption routines for the bot’s communication with the control nodes. The recovery of encryption and decryption functionality from executables usually requires a lot of manual work and analysis. In this paper, we present an approach that automates the localization of possible en- and decryption functions. This enables analysts to extract the functionality and create decryption add-ons for monitoring tools.

While traditional tools only scrutinize data leaving the malware, we correlate this information with details from inside the malware. For that, we determine the creation functions of I/O buffers, which are often close to the encryption functions or even include the functionality. A similar approach is used for input buffers and decryption routines.

Using our approach, we were able to find the encryption and decryption functions inside a *Kraken* botnet sample within minutes. We illustrate the applicability of our approach based on the Kraken sample. In addition, we release a *C* re-implementation of the encryption and decryption code extracted from the sample. This code can be used to monitor Kraken traffic.

The rest of the paper is structured as follows: Section 2 provides an overview of related work. Section 3 describes our approach in more detail. Section 4 shows the applicability of our approach based on this Kraken botnet sample and describes the extracted encryption and decryption routines. Implications of publishing our approach are discussed in section 5. Section 6 concludes and gives an overview of future work.

2 Related Work

Malware may be analyzed in two different ways: *Static analysis* and *dynamic analysis*.

Static analysis is performed on the binary without executing it. This is typically conducted by disassembling the binary and extracting information about data and control flow. This approach is usually faster than dynamic analysis [5]. Christodorescu et al. [8] have presented malware analysis techniques based on static analysis.

A major drawback of static analysis is that the code analyzed may be different from the code executed. This is caused by packers [4, 18], encryptors, polymorphism[21], or obfuscation techniques[16].

Dynamic analysis tools monitor the malware while it is running. Classical examples of dynamic analysis tools are debuggers. A series of dynamic analysis tools that monitor typical actions, like e.g. file, registry and network access, exist [9, 11]. Some are based on API hooking and monitor malware from inside the system [24]. Others emulate a whole PC and monitor the malware behavior from outside [5, 6]. Automated botnet monitoring systems, like e.g. [20, 23], often rely on this kind of systems for extracting the c&c information.

These tools are designed for the mass-analysis of malware and obtain valuable information from malware using standard protocols. They fail for proprietary, encoded, and encrypted data if the decryption algorithms are not known. Typically, they only monitor the data leaving the malware, details from inside the malware are not taken into account.

Different debuggers are available [1, 15, 25], for scripting and flexible monitoring of Windows API calls. They can be used for locating the encryption and encoding functions of malware but require a lot of additional manual work. However, they are not able to automatically determine the data origin and correlations to I/O.

The approach closest to ours is the automated reverse engineering framework PaiMei [3]. It traces program execution and collects information at different trace points. PaiMei is a generic framework. Data is collected about every function inside the application. It is left to the analyst to extract the necessary relations out of the lot of information collected.

3 Methodology

We have observed that the encryption and decryption functions are often close to the creation points of the buffers they use. From a software developer point of view, this is an intuitive behavior since the buffers are allocated only when they are needed.

In general, the encryption of data the last operation performed on the data, before it leaves the executable. The buffer passed to I/O interfaces is the one containing the result of the encryption process. The same holds for the decryption as displayed in figure 1. In order to receive (encrypted) data from an input interface, a buffer has to be created. The buffer is then passed to the input interface. It may pass an arbitrary number of management functions (c.f. section 3.2). The buffer is filled with encrypted data behind the input interface. After returning the buffer, it must be decrypted before data can be used.

We are monitoring the I/O interfaces, like e.g. *send()* or *recv()*. The buffer addresses detected at I/O interfaces allow us to automatically determine the buffer creation function inside the malware. As this creation point is close to the encryption or decryption function, it can be used as a starting point for deeper analysis and extraction of the cryption functions.

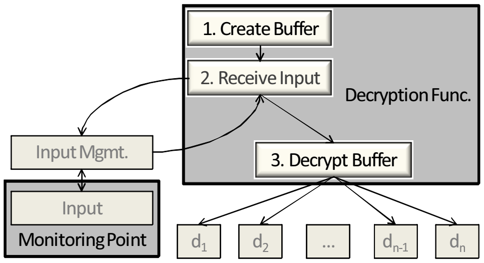


Fig. 1 Schematic flow of buffer creation and usage during the process of receiving encrypted data and decrypting the data. After the buffer is created, it is passed to the input routine. This returns the buffer filled with encrypted data. As a third step, the buffer is decrypted.

3.1 Assumptions

Our approach is based on some assumptions about the structure of the program. Of course, malware developers may adapt their programs to avoid meeting these assumptions. Implications are discussed in section 5.

We focus on the Windows operating system and x86 architectures because more than 95% of malware in-the-wild is developed for that platform[22].

Our most important assumption is that malware is using the standard I/O interfaces of the operating system (OS). This assumptions allows us to place monitoring points on these I/O interfaces. Malware authors, like authors of any other software, rely on the I/O functionality provided by the OS in order to be more independent from the system architecture. Malware with custom crafted file or network drivers would lack flexibility.

Additionally, we assume that buffers are created at the time they are needed. This reflects the intuitive behavior to allocate the buffer in the scope when it is required. It may be discarded after leaving their scope.

The encryption process is the last operation performed on the data before leaving the malware. Vice versa, the decryption process is the first operation performed on incoming data. We assume that the encryption functionality places its result in the output buffer and that this buffer is passed to the output interface. The same holds for the input buffer and decryption functionality.

A scenario, in which the encryption uses another buffer, which is later on copied into the output buffer, is not critical. In this case, the copy operation can be determined using other means, like e.g. copy signatures or using memory breakpoints. This allows for an iterative application of our approach.

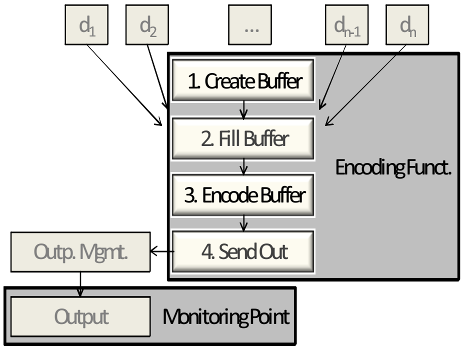
3.2 Buffer lifecycle

Buffers are used to transfer data in and out of the executable. Of course, there are different lifecycles for input and output operations. However, they show a similarity, which we exploit for finding the buffer origin.

Figure 1 shows the typical lifecycle of a buffer used for encrypted input. The buffer is created as part of the encryption initiation. It is then given to the input interface of the operating system. It may pass arbitrary management functions, which may perform error handling or add context information, like e.g. the socket descriptor. After the buffer has been filled with encoded data outside of the executable, it is returned. The buffer is then decrypted for extraction and usage of the original data.

Figure 2 displays the typical lifecycle of output buffers. In a first step, the buffer is created. It may be filled with the original, unencrypted data as an optional step. The buffer is then encrypted and the encryption result is passed to the output interface. Similar to the lifecycle of input data, it may pass arbitrary management functions for similar reasons.

Fig. 2 Lifecycle of output buffers for encrypted data. After the buffer is created, it may be filled with unencrypted data chunks. The data is then encrypted and sent out to the output interface. It may pass management functions, before. When its scope ends, the buffer may be discarded.



Both lifecycles have in common that the buffer creation is preceding the I/O operations. We have observed that the buffer creation function is often close to the encryption functionality or may even include this functionality. In these cases, we can locate the cryption routines from the buffer creation point.

3.3 Monitoring points

We are monitoring different I/O interfaces to gather information about buffer creation points and the context in which a buffer is used. The context includes information about data endpoints, like networking peers or files. The buffer creation functions and the context in which the buffer is used can automatically be determined when monitoring three different types of interfaces:

- Heap memory management functions
- I/O initialization
- I/O operations

Heap operations are monitored in order to detect the allocation of new buffers. The address and size of each allocated memory block is stored together with the function that initiated the allocation. This mapping is used to determine the creation point for heap buffers monitored in I/O operations.

The I/O initialization functions, like *connect()* or *OpenFile()*, are monitored to collect context information. The initialization functions provide information about the data endpoint, like filenames or IP addresses. Later, the collected information may be mapped to specific buffers. This eases the extraction of the desired functions for specific endpoints.

Monitoring points on the actual I/O interfaces, like *send()* or *ReadFile()*, are used to determine the actual buffer origin. Thus, they are essential for locating the cryption functions.

3.4 Determining the buffer origin

The primary goal of our approach is to find the creation point of buffers holding encrypted data. As we have observed, the creation point is often close to the encryption function for output buffers and respectively close to the decryption function for input buffers.

For this purpose, monitoring points are placed on relevant I/O interfaces. If a buffer is passed to a monitored I/O interface, three steps are performed:

1. Extraction of the buffer address
2. Mapping to type of memory
3. Mapping to function based on memory region

First, the buffer address is extracted from the call to the I/O interface. The location of the address depends on the calling convention of the interface but can be found at fixed positions. It is located either in registers or at fixed offsets on the call stack.

Based on the buffer address, a mapping to its memory region has to be performed because different methods have to be used for the mapping to a creation function. The choice of the method dependends on the memory region, which may be heap memory, stack memory, or global memory.

Figure 3 illustrates the mapping for buffers located on the stack. Each function on the stack has a dedicated stack frame. A stack frame is used for the return address, for call parameters as well as for local variables. Once the address of a stack buffer is known, the stack frame containing that address may be determined. The buffer is a local variable of the function that created this stackframe. The address of this function is the buffer creation point. In addition, the function address may be determined from the stackframe.

Heap buffers contain no information about the function that created them. In order to determine their creation function, we use monitoring points on heap management functions, like *RtlAllocateHeap()*. This way, we can create a mapping from the function using the heap management to the allocated memory space. If the address of the I/O buffer points to the heap, the list of mappings is examined for the

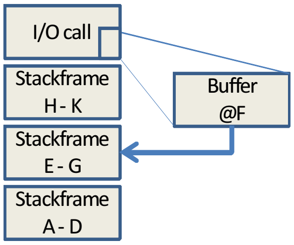


Fig. 3 A stack buffer passed to an I/O interface may be mapped to its creation function. The mapping is achieved by comparing the buffer address to the boundaries of different stack frames.

space containing the buffer. As the heap memory is non-overlapping, the creation function can be determined and is unambiguous.

A creation function for global memory cannot be determined because it is created at program start. As it is constantly occupying memory and more difficult to manage, it is hardly ever used for I/O buffers.

4 Application - Extracting Kraken Encryption

We illustrate the applicability of our approach using a Kraken botnet sample: We were able to identify the encryption and decryption function within minutes. Based on this, we were able to recover the full cryption process for the proprietary Kraken command & control (c&c) protocol.

The Kraken Botnet is said to be the largest botnet in the world [19]. Estimations of the botnet size range from 185.000 to 600.000 zombie hosts worldwide.

Its main purpose is to spread SPAM mail. Single infected hosts have been observed transmitting as much as 500.000 junk mails. Besides that, it harvests the windows address book and local files for email addresses and can install additional malware.

The bots contain a list of dynamic DNS hostnames for contacting the botnet master [17]. They subsequently try to contact each hostname via UDP until a response is received. After a successful handshake, the bots use a proprietary, encrypted c&c protocol for data exchange.

For our evaluation, we have analyzed a Kraken sample from early 2008. It uses the Kraken protocol version 311. For manual verification of the results, we unpacked the sample[7].

For use in our NGBPA implementation, we have used the original, packed sample and placed monitoring points on networking functions like *sendto()* and *recvfrom()*. After having started the sample, we observed connection attempts to different SMTP servers for 20 seconds, which we intentionally dropped. 20 seconds later, the first encrypted buffer sent via UDP to port 447 was captured. This buffer - which was passed to *sendto()* - was located on the stack. The buffer was contained in the stack frame of function *sub_1A832C*. Not answering those requests, we could see similar requests to different hosts every 10 seconds. The buffer origin stayed the same for all of these.

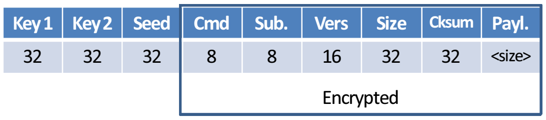
```
.text:001A83CA mov dword ptr [esp+80h+buf], eax
.text:001A83CE lea eax, [esp+80h+buf] ; key1
.text:001A83D2 mov [esp+80h+var_2C], edx ; key2
.text:001A83D6 mov [esp+80h+var_28], ebx ; seed
.text:001A83DA mov [esp+80h+var_24], 1 ; cmd 1
.text:001A83DF mov [esp+80h+var_23], bl ; subcmd
.text:001A83E3 mov [esp+80h+version], 137h ; vers.
.text:001A83EA mov [esp+80h+var_20], ebx ; size
.text:001A83EE mov [esp+80h+var_1C], ebx ; chksum

.text:001A83F2 call encryptHeader <----------

.text:001A83F7 call create_new_udp_sock
...
.text:001A8422 lea eax, [esp+90h+buf]
.text:001A8426 push eax ; buf
...
.text:001A842B call ds:sendto  <-------------
```

Fig. 4 Kraken encryption origin

A closer look at the creation function revealed the code block shown in figure 4[1]. The excerpt shows, how different fields in the buffer are filled with keys, some seed, commands, protocol version, size, and a checksum. Looking at the two functions following this block, reveals suspicious mathematical operations in the first function (001A83F2) while the second (001A83F7) creates a UDP socket.

Having a candidate for the encryption, we loaded the binary in a debugger and placed a breakpoint on that function. Running the candidate functions shows that the buffer is modified. The result was the data sent out via UDP, afterwards. A manual investigation and a dissection from C. Pierce [17] verified this function to contain the encryption.

Based on these results, we were able to reconstruct the decryption and encryption functions used in the kraken botnet. A re-implementation in C can be found in the appendix. The protocol is shown in figure 5. The first three fields are two keys and a seed, which are used for encryption and decryption. The other fields are symmetrically encrypted before transmission. As the encryption is symmetric and keys are included in each payload, it is encryption by obfuscation but not secure in any way.



Key 1	Key 2	Seed	Cmd	Sub.	Vers	Size	Cksum	Payl.
32	32	32	8	8	16	32	32	<size>
			Encrypted					

Fig. 5 The Kraken protocol. Shown is the protocol header including the number of bits for each field. Only the keys and seed are unencrypted.

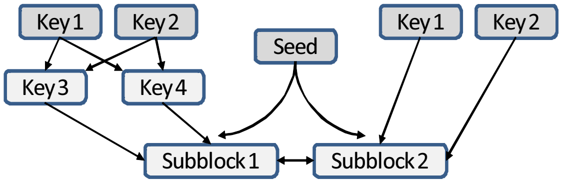
The two keys are derived from information about the host hardware. The derivation of the two keys together as well as of the checksum is described in more detail

[1] The annotations and comments were added later

in [17]. We found the creation of the seed in the encryption function. It is different for each data packet. The seed is based on the processor tick count and computed by adding the 32 high-bits to the 32 low-bits.

Figure 6 illustrates the data dependency used in encryption and decryption. Details may be studied in our C re-implementation included in the appendix. As illustrated in figure 5, all fields except for the keys and seed are encrypted together with the c&c payload. The encryption algorithm can be applied in 8-byte-blocks or bytewise. The kraken sample studied uses block-encryption. The data is split into 8-byte-blocks, which are divided into two subblocks. Each subblock is used to encrypt the other in combination with the seed and the two keys. If the data size is not a multiple of 8 bytes, the last bytes are encrypted bytewise.

Fig. 6 The data dependencies of the Kraken encryption. Keys 3 and 4 are derived by bitshift operations of keys 1 and 2. Each block is divided into two subblocks. These are encrypted using the seed, two keys and the other subblock.



By spoofing UDP answers, we were able to locate the decryption function with our approach, too. The buffer for the *recvfrom()* call was created in the same function as the send buffer. Figure 7 shows the excerpt related to data reception. The decryption function is located right after the call to *recvfrom*

```
.text:001A8464 lea eax, [esp+90h+recvbuf]
.text:001A8468 push eax ; buf
...
.text:001A846D call ds:recvfrom  <--------------
...
.text:001A8478 lea esi, [esp+80h+recvbuf]

.text:001A847C call decryptHeader <-------------
```

Fig. 7 Kraken decryption origin

The keys and seed for the decryption are contained in the c&c protocol data. Thus, it is possible to decrypt all c&c traffic using the information transmitted over the network. Monitoring applications can make use of this to create their own decryption stub. A C re-implementation of the decryption is included in the appendix.

Identifying both encryption and decryption took us only a few minutes: Running the kraken botnet sample, our NGBPA tool took 20 seconds before the first packet was sent out, which immediately revealed the origin of the buffer. Around 5-10 minutes of manual investigation were needed afterwards to identify the encryption and decryption functions.

This example illustrates both the applicability and performance of our approach. The application of our approach to other malware samples, not mentioned here, showed a similar efficiency.

5 Discussion

Publishing our approach may invalidate it because malware authors may design new specimen specifically to not meet our assumptions. In this section, we discuss implications.

We assume that malware is using the OS for I/O. Our approach fails for custom I/O drivers directly accessing the hardware. However, for the malware author custom drivers increase development complexity and reduce flexibility.

Another assumption is that buffers are created at the time they are needed. Allocation long before is a rather unintuitive development strategy and complicates the design. The encryption can still be found using memory monitoring with breakpoints or emulator extension.

Other possibilities to break our approach are the use of global buffers or implementation of custom designed memory management functions. For malware authors, this complicates the software design and therefore maintainability, increases the risk for bugs, and may break modularity. This has an impact on the overall architecture and development efficiency. Since malware and especially botnet development is becoming more and more professional with a standard “testing and revision process” [12], it has to be efficient. It is questionable whether malware developers would take this step.

While malware authors probably stick with regular software design for reasons named before, our approach may be beneficial for a whole group of malware researchers. We therefore decided to publish even though there is a risk of limiting the lifespan of our approach this way.

6 Conclusions and Future Work

We were able to demonstrate the applicability of our approach. With a practical implementation, we were able to identify the encryption and decryption routines of the *Kraken* botnet within minutes. We were able to extract and re-implement the encryption and decryption logic, which is included in the appendix and can be integrated into botnet monitoring tools. We therefore conclude it to be a valuable component in the malware analysis toolchain.

One example is not enough to show a general usability. For that reason, our approach has to be evaluated with a representative set of malware samples. A major question in this context is how many samples from which sources are required to be representative for the malware in-the-wild.

In addition, an easy to configure interface to our implementation would be beneficial to speed up analysis. This includes the selection of typical I/O interfaces. Another future feature is the integration of additional buffer monitoring in case the considered malware violates our current assumptions.

Appendix

C re-implementation of the Kraken protocol decryption function.

```
void decode(uint8_t* buffer, uint32_t buffer_size, uint32_t key1, uint32_t key2,
            uint32_t seed, uint32_t blockwise_flag) {
  int  i;
  uint32_t buffer_pos = 0;
  uint32_t keys[] = {key1, key2, (key2 >> 0x13) | (key1 << 0x0d),
                                 (key2 << 0x0d) | (key1 >> 0x13)};
  if (blockwise_flag) {
      while (buffer_size - buffer_pos >= 8) {
          uint32_t* data1 = (uint32_t*) &buffer[buffer_pos];
          uint32_t* data2 = (uint32_t*) &buffer[buffer_pos+4];
          uint32_t round_key = seed + seed;

          for (i=0; i<2; ++i) {
              *data2 -= (*data1 << 4) + keys[2] ^ (*data1 >> 5) + keys[3]   \
                        ^ (round_key + *data1);
              *data1 -= (*data2 << 4) + keys[0] ^ (*data2 >> 5) + keys[1]    \
                        ^ (round_key + *data2);
              round_key -= seed;
            }
          buffer_pos += 8;
      }
  } /* the rest is decrypted bytewise */
  buffer = &buffer[buffer_pos];
  buffer_size -= buffer_pos;
  for (i = 0; i < buffer_size; ++i) {
      uint8_t seedbyte = (seed >> 8 * (3 - i%4) ) & 0xff;
      buffer[i] ^= ((uint8_t*)keys)[i] + seedbyte;
  }
}
```

C re-implementation of the Kraken protocol encryption function.

```
void encode(uint8_t* buffer, uint32_t buffer_size, uint32_t key1, uint32_t key2,
            uint32_t seed, uint32_t blockwise_flag) {
  int  i;
  uint32_t buffer_pos = 0;
  uint32_t keys[] = {key1,key2,(key2 >> 0x13) | (key1 << 0x0d),
                               (key2 << 0x0d) | (key1 >> 0x13)};
  if (blockwise_flag) {
      while (buffer_size - buffer_pos >= 8) {
          uint32_t* data1 = (uint32_t*) &buffer[buffer_pos];
          uint32_t* data2 = (uint32_t*) &buffer[buffer_pos+4];
          uint32_t round_key = 0;

          for (i=0; i<2; ++i) {
              round_key += seed;

              *data1 += (*data2 << 4) + keys[0] ^ (*data2 >> 5) + keys[1] \
                         ^ (round_key + *data2);
              *data2 += (*data1 << 4) + keys[2] ^ (*data1 >> 5) + keys[3] \
                         ^ (round_key + *data1);
          }
          buffer_pos += 8;
      }
  }  /* the rest is encrypted bytewise */
  buffer = &buffer[buffer_pos];
  buffer_size -= buffer_pos;
  for (i = 0; i < buffer_size; ++i) {
      uint8_t seedbyte = (seed >> 8 * (3 - i%4) ) & 0xff;
      buffer[i] ^= ((uint8_t*)keys)[i] + seedbyte;
  }
}
```

References

1. P. Amini, *PyDbg - A pure Python win32 debugging abstraction class*, http://pedram.redhive.com/PyDbg/, last visit (l.v.): Oct. 2008
2. P. Amini, *Kraken Botnet Infiltration*, Blog on DVLabs, http://dvlabs.tippingpoint.com/blog/2008/04/28/kraken-botnet-infiltration, Apr. 2008
3. P. Amini, *PaiMei - Reverse Engineering Automization*, http://pedram.redhive.com/research/reverse_engineering_automation/, l.v.: Oct. 2008
4. Archer and FEUERRADER, *QuickUnpack*, http://reversengineering.wordpress.com/2007/10/06/quick-unpack-v20-final/, l.v.: Aug. 2008
5. U. Bayer and C. Kruegel and E. Kirda, *TTAnalyze: A Tool for Analyzing Malware*, In 15th Annual Conference of the European Institute for Computer Antivirus Research (EICAR), 2006
6. F. Bellard, *QEMU, a Fast and Portable Dynamic Translator*, USENIX Annual Technical Conference, 2005
7. N. Brulez, *Unpacking Storm Worm*, http://securitylabs.websense.com/content/Blogs/3127.aspx, l.v.: Aug. 2008
8. M. Christodorescu et al., *Semantics-aware malware detection*, IEEE Symposium on Security and Privacy, 2005
9. G. Combs, *Wireshark - network protocol analyzer*. http://www.wireshark.org, l.v.: Oct. 2008
10. D. Dittrich and S. Dietrich, *Command and control structures in malware*, Usenix magazine, Vol. 32, No. 6, Dec. 2007
11. R. Russinovich and B. Cogswell, *Windows Sysinterals*, http://technet.microsoft.com/en-us/sysinternals/default.aspx, l.v.: Oct. 2008
12. D. Fisher, *Storm, Nugache lead dangerous new botnet barrage*, Article, http://searchsecurity.techtarget.com/news/article/0,289142,sid14_gci1286808,00.html, l.v.: Oct. 2008
13. G. Hoglund and J. Butler, *Rootkits*, Addison Wesley, Dec. 2005
14. Holy Father, *Hooking Windows APITechnics of Hooking API Functions on Windows*, Code-Breakers Journal, vol. 1, no. 2, 2004
15. Immunity Inc., *Immunity Debugger*, http://www.immunitysec.com/products-immdbg.shtml, l.v.: Oct. 2008
16. C. Linn and S. Debray, *Obfuscation of executable code to improve resistance to static disassembly*, Proceedings of the 10th ACM conference on Computer and communications security, 2003
17. C. Pierce, *Owning Kraken Zombies, a Detailed Dissection*, Blog on DVLabs,http://dvlabs.tippingpoint.com/blog/2008/04/28/owning-kraken-zombies, l.v.: Oct. 2008
18. P. Royal and M. Halpin and D. Dagon and R. Edmonds and W. Lee, *PolyUnpack: Automating the Hidden-Code Extraction of Unpack-Executing Malware*, ACSAC '06: Proceedings of the 22nd Annual Computer Security Applications Conference on Annual Computer Security Applications Conference, 2006
19. P. Royal, *On the Kraken and Bobax Botnets*, Whitepaper, Damball, Apr. 2008
20. Shadowserver Foundation, *ShadowServer Homepage*, http://shadowserver.org, l.v.: Oct. 2008
21. P. Szor, *The Art of Computer Virus Research and Defense*, Addison-Wesley, Feb. 2005
22. Symantec Coorp. *Symantec Internet Security Threat Report Volume XIII*, Whitepaper, Apr. 2008
23. G. Wicherski, *botsnoopd - Sniffing on Botnets*, Blog, http://blog.oxff.net/2006/10/botsnoopd-sniffing-on-botnets.html, l.v.:Oct. 2008
24. C. Willems and T. Holz and F. Freiling, *Toward Automated Dynamic Malware Analysis Using CWSandbox*, Ieee Security & Privacy, 2007
25. O. Yuschuk, *OllyDbg Debugger*, http://www.ollydbg.de/, l.v.: Oct. 2008