Finding and Extracting Crypto Routines from Malware

Felix Leder, Peter Martini, Andre Wichmann
University of Bonn, Germany
Institute of Computer Science IV
Email: {leder, martini, wichmann}@cs.uni-bonn.de

Abstract

In this paper we present a new approach for identifying the crypto routines in different types of malware. In traditional malware analysis, like sandboxing, network data is examined as seen on the wire or data is collected as it is written to a file. The use of proprietary binary formats, obfuscation, or encryption hides important details, which are necessary for investigating malicious behavior. It is hardly possible to create decryptors just from monitored sandbox data. Our approach not only examines the data when leaving or entering the malware but also correlates it with information from inside the malware. By monitoring the data at I/O interfaces as well as data dependencies our approach automatically reveals the data origin. Knowing the data origin enables an analyst to easily find the crypto functions.

Using this approach, we were able to identify the encryption, decryption, and command parser in different malware samples each within minutes. In our evaluation, we present the results for the Kraken command&control protocol encryption and for the file encryption of the Srvcp trojan.

1. Introduction

An increase of malware has been observed all through the last years but has never reached the exponential growth currently observed [20]. There is a clear trend towards one-time binaries that tend to change from infection to infection. Along with the rising overall number, more and more stealth and hiding techniques can be found in malware [8], [20].

While sandboxing used to be enough to extract the relevant information for understanding the behavior of malware and for monitoring botnets [4] [22], modern bots often encrypt network traffic, files and data in order to avoid eavesdropping. Monitoring, as performed by [18], is essential for estimating the effects and infections of specific malware. Thus, it is necessary to extract the crypto functions and include those into existing monitoring tools, like the extension [11]

While sandboxing is an efficient means when network traffic and files are unencrypted, it is rather useless for encrypted data. Sandboxes are just monitoring the data passed along the OS interfaces, but encryption and decryption takes place inside the malware. The data leaving or entering the malware is just the result. Automated decryption and decoding is required in order to monitor and classify malware specimen.

The recovery of crypto routines has required a lot of effort for manual reverse engineering and analysis, in the past. We present an approach that automates the finding of parts inside a malware that contain possible crypto functions. The knowledge about these parts, enables analysts to extract the functionality and create decryption add-ons for monitoring tools.

Similar to sandboxes, we are monitoring the I/O interfaces to the OS. Instead of just collecting the data leaving the malware, we combine the information at the interfaces with information from within the malware. The main focus is to monitor buffers as they passed to I/O interfaces. We use the memory address of the buffer together with data dependecy information for locating the buffer origin. We have observed that the buffer origin is often close to the crypto routines. The reason for this is that buffers are usually created at the time they are needed and not far ahead.

Using our approach we were able to find the decryption functionality from different malware specimen within minutes.

The rest of the paper is structured as following. Section 2 gives an overview about related work. Section 3 describes our approach in more detail. The applicability is shown using a Kraken bot sample as well as the Srvcp trojan in section 4. The publication of our approach may invalidate it. Possible implications are discussed in section 5. Section 6 concludes and gives an overview about future work.

2. Related Work

Traditional malware analysis follows two different paradigms. One way is to perform *static analysis* on the raw binary. Another way is monitoring the malware as it being executed with *dynamic analysis*.

During static analysis, the binary is analyzed without executing it [7]. For that, it is typically disassembled to assembly instructions. Based on the disassembly, the control flow as well as information about the data usage is extracted. It is often faster than dynamic analysis [4]. The downside of static analysis is that different data may be executed than the instructions seen during the analysis. This is the case when packers [16], polymorphism [19], or obfuscation techniques [13] are used.

Dynamic analysis tools analyze the malware while it is executed. They normally monitor file and registry accesses as well as network traffic. Some tools, like [9], [22], make observe the malware from inside the system in which it runs. Other approaches emulate the full computer and observe the behavior of malware from outside the system [4]. It uses QEmu [5] for emulating x86 PCs. Systems that monitor botnets, e.g. [18], [21], often rely on those tools for extracting information about the command & control (c&c) protocol.

Tools, like the presented ones are used for the mass-analysis of malware. They are able to obtain information about malware using standard protocols. They are generally not usable for encrypted. They only observe the data leaving or entering through the OS interfaces but they don't take details from inside the program into consideration.

Debuggers that allow scripting [1], [12], [23] can be used to monitor API calls and other OS interfaces. They may be used for finding coding functions but this requires a lot of manual work because they don't include the functionality for data collection and correlation.

The automated reverse engineering framework PaiMei [3] is the method closest to our approach. It traces program execution and collects information at different breakpoints for posterior analysis. PaiMei is a generic reverse engineering framework with no specific focus. Therefore, it does not contain correlation functionality for finding buffer creation or crypto functions.

3. Methodology

Malware authors, like authors of any other software, rely on the I/O functionality provided by the operating system (OS) in order to be independent from the OS version.

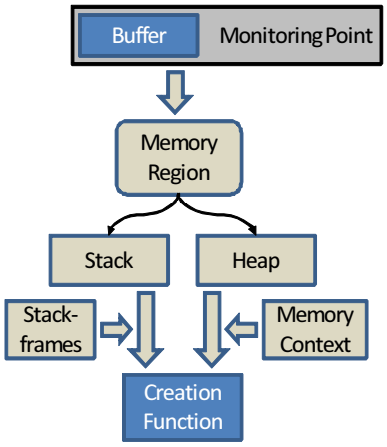

Figure 1. Determining creation function for buffer

In our approach, we exploit this to monitor data as it is passed to the I/O interfaces of the OS. We collect information about the memory, context information, like filenames or network endpoints, as well as buffers that contain encrypted data. For this, we are placing monitoring points at the relevant I/O interfaces, e.g the API function *send()*.

Whenever a buffer is observed at a monitoring point, the memory region that contains the buffer is determined. Combining the information about memory region, buffer address, and details about the data dependencies allows us to automatically determine the buffer origin. We have observed that the buffer origin is often close to the *crypto routine*. This is for two reasons. First, crypto is the last action performed on the data before it leaves the malware. Second, it is an common development paradigm to allocate buffers at the time they are needed and not long before.

The automated detection of the buffer origin is depicted in figure 1. It relies on the knowledge about data dependencies inside the malware. The data dependencies must be determined differently for stack and heap memory regions. For stack memory, this is achieved by examining the stack frames at the time of the interface call. For heap memory, information from allocations monitoring can be used.

The vast amount of I/O usually performed by malware requires efficient filtering in order to focus the analysts view for the crypto routines. Correlated information about data endpoints, like filenames, enable such filtering.

In the following, our approach is formally described before we present some aspects of our practical realization.

3.1. Formal description

Definition 1. Let ζ be the set of functions.

Definition 2. Furthermore, let

$$Enc(d_1 \circ \cdots \circ d_n)$$

be an encryption for the combined data units $d_1, \cdots, d_n$.

Definition 3. For a given $Enc(.)$, let

$$Dec(Enc(.)) := Enc^{-1}(Enc(d_1 \circ \cdots \circ d_n)) = d_1 \circ \cdots \circ d_n$$

be the decryption.

There may be a different decryption and ecnryption in the botnet controller and the zombie. If $Dec(.) \in \zeta$ and $Enc(.) \in \zeta$, the malware contains both. This may be because of the malware using symmetric encoding or because the malware can be both controller and zombie at the same time.

Definition 4. Let $B := \{b_1, ..., b_n\}$ be the set of all buffers used in the program. A buffer is a dedicated space in memory that can be used in the time between its creation and removal.

Definition 5. For a given buffer $b_i \in B$, let

$$D(b_i) := \{f \in \zeta | f \text{ defines } b_i\}$$

be the set of functions that define b_i. This means that all $f \in D(b_i)$ fill the buffer b_i with data. It will be named *definition set of* b_i in the following.

Definition 6. For a given buffer $b_i \in B$, let

$$U(b_i) := \{f \in \zeta, f \text{ uses } b_i\}$$

be the set of functions that use b_i. This means that all $f \in U(b_i)$ access data in buffer b. It will be named *usage set of* b_i in the following.

Definition 7.

$$C(f) = \{b_i \in B | f \text{ creates } b_i\}$$

$C(f)$ is the set of buffers created by function f.

Definition 8. An execution path ρ, is a sequence of functions $f_1, f_2, \cdots, f_m$ that reflects the order in which functions have called each other during execution.

3.1.1. Assumptions. We assume that the encryption inside the malware is conducted in the definition sets $D(b_i)$ for a buffer b_i, which is used at a monitoring point, later on. This situation is illustrated in figure 2. Furthermore, we assume

$$D'(b_i) := D(b_i) \cap Enc(.) \neq \emptyset$$

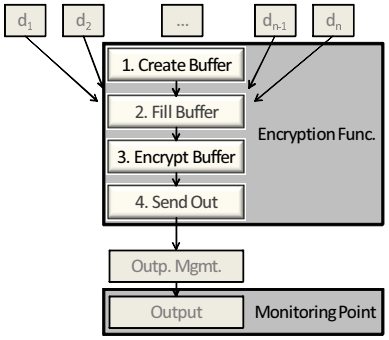

Figure 2. Encryption and output of a buffer

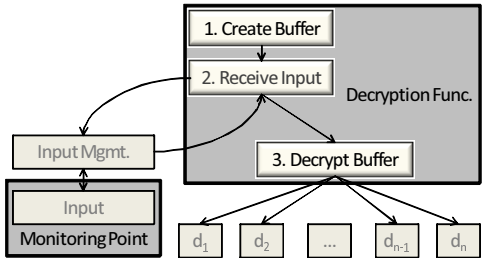

Figure 3. Input and decoding of a buffer

$D'(b_i)$ is that part of the definition set of b_i that ecrypts into b_i. The crypto consists of four steps that are combined in one function. First, the buffer is created. Second, as an optional step, the buffer is filled with data units. The units may already be combined and just copied into the buffer. The third and important step is to encrypt the data into the buffer. Fourth, the buffer is passed out of the crypto function. After that, it may pass a number of functions related to managing the output. Finally, it is passed to the output interface. Montoring points are placed on the output interface.

The decryption is performed in the usage sets $U(b_i)$ for a buffer b_i. We assume:

$$U'(b_i) := U(b_i) \cap Dec(.) \neq \emptyset$$

$U'(b_i)$ decrypts the data contained in b_i. This is illustrated in figure 3. The decoding functionality consists of 3 steps. First, the buffer is created. Second, the buffer is passed to the input interface. A monitoring point is placed on that input interface. The buffer may pass an arbitrary number of input management functions. The third and important step is the decryption of the buffer data.

We assume that

$$\forall b_i \in B : \exists f \in \zeta | f \text{ creates } b_i$$

Every buffer is created by a function in the malware

program. This does not hold for global buffers. Implications are discussed in sections 3.1.2, 5.

More specific, we assume that every buffer b is created by a *single* function $\widehat{f}_b \in \zeta$. It is either contained in the stack frame of that function or in a heap buffer allocated from that function.

Let

$$M : b_i \mapsto \{Stack, Heap, GlobalMemory\}$$

be a function that maps a buffer b_i to the memory region in which it is located. We assume that such a mapping exists and that buffers exist either on the stack, heap or in global memory.

3.1.2. Methodology. We have observed that the creation function of a buffer is very often close to the crypto routines using it. Therefore, the goal is to find the creation function $\widehat{f}_b$ for buffer b.

This is achieved in a three stages.

1) Placing a monitoring point MP on the relevant I/O interface
2) Determine the memory region containing buffer b_i, which is observed at MP
3) Find creation function $\widehat{f}$ based on the memory region and information about b_i

Let f^* be the I/O interface function. In a first step f^* is monitored. During execution, buffer b_i is observed when being passed to f^*.

In a second step

$$M(b_i) \rightarrow \{Stack, Heap, GlobalMemory\}$$

is used to determine the memory region, in which b_i is located. Buffers can reside in three different types of memory: Stack, Heap, and Global Memory. The buffer creation function $\widehat{f}_b = C^{-1}(b)$ is determined depending in the memory type.

Since each buffer $b_i \in B$ is created by a single function, C^{-1} exists with

$$C^{-1}(b_i) = f \in \zeta | f \text{ created } b_i$$

Stack buffers must exist inside the stackframe of one functions in ρ. Each $f \in \rho$ can uniquely be identified by its stackframe λ_f and its return address.

$$\widehat{f}_b = C^{-1}_{stack}(b) := f \in \rho | b_i \text{ is part of } \lambda_f$$

is the function f that created and contains b_i in its stackframe λ_f.

$C(f)$ can be determined for heap buffers by monitoring all heap allocations. Since buffers are unique and created by a single function, C^{-1}_{heap} can be constructed and

$$\widehat{f}_b = C^{-1}_{heap}(b)$$

Global buffers are created at program start by the OS and not by a function inside the malware. Therefore, global buffers are left out of scope. It is not known to us that any malware is using global buffers for I/O operations, but it would theoretically be possible. This is duscussed in section 5.

Thus, based on $M(b_i)$ the buffer creation function

$$\widehat{f}_b = C^{-1}(b)$$

for buffer b can be determined:

$$C^{-1}(b_i) := \begin{cases} C^{-1}_{stack}(b_i) & : \quad M(b_i) = Stack \\ C^{-1}_{heap}(b_i) & : \quad M(b_i) = Heap \end{cases}$$

3.1.3. Finding the crypto routine. The final goal is to determine the crypto routine. The identification and knowledge about the crypto routine allows an analyst to reveal the coding logic and to integrate it into existing monitoring tools.

For encryption, the definition set $D'(b_i)$ is of relevance. For decryption, the usage set $U'(b_i)$ is of interest. The finding is similar in both cases. We will focus on the encryption in the following.

Let $f^\circ \subseteq D'(b_i)$ be a single function that either performs the full crypto or initiates and coordinates it. Based on our observations very often either the crypto routine is the same as the buffer creation function or it is called from the buffer creation function:

$$f^\circ = \widehat{f}$$

or

$$\exists \rho^\circ = f_1, ..., f_i, f_{i+1}, ... | f_i = \widehat{f}, f_{i+1} = f^\circ$$

From a software design perspective this is a very intuitive behavior. A software author creates a buffer at the time it is needed and does not unnecessarily block memory. Soon after the buffer is created, it is used for crypto. Then, it is passed to the I/O interface. Thus, the crypto can be found by investigating the buffer creation function.

3.2. Practical realization

The implementation is presented and evaluated using Microsoft Windows malware. The general approach is not dependent on the platform.

As described in the previous section, we locate the crypto by finding the creation function for I/O buffers.

The practical implementation is based on the concept of *monitoring points* and run-time analysis. Monitoring points are used for observing relevant API calls. They are placed on three types of interface functions:

1) Data endpoint and context information

2) Heap operations
3) input and output

Context information and information about the data endpoint are used to filter out the results for a specific context. This information can hold filenames or network connection addresses. Examples for monitored context API calls are *connect()* or *CreateFile()*. The handles returned can later be used to map a specific buffer origin to a given context. With this information it is possible to differentiate buffer origins by their context, i.e. different files or network endpoints, which may be used simultaneously.

Heap operations are monitored to create a list of heap sections and a mapping to the function that created those regions. This mapping is used later on to determine the origin of heap buffers.

The most important monitoring points are the ones for input and output API, like *send()* or *WriteFile()*. The buffer origin is determined whenever such a monitoring point is triggered. In addition, the context computed by comparing the handles. This is used for filtering for specific data endpoints, like file names. The buffer passed to the I/O API is examined in order to find its origin. Based on the memory address of the buffer, the memory region containing the buffer is determined.

The memory region can be determined by comparing the buffer address to the start and end of each memory region inside the malware.

If the buffer is located on the stack, the stack is unrolled and split into the stackframes. In addition to the boundaries of each stackframe, the functions that created the frame is extracted. Then, the buffer address is compared with each stackframe. The stackframe that contains the buffer belongs to the creation function of the buffer.

In case the buffer is located in the heap, the information from *heap monitoring points* is used to determine the buffer creation function. This is performed using a mapping of heap sections to their creation functions and matching them ti the buffer address.

4. Evaluation

We will show the applicability of our approach using two different malware samples. For all of these samples, as well as the Storm worm and SdBot variants that are not mentioned here, we were able to identify the relevant functions within minutes.

The malware samples we present in the following are a sample of the Kraken botnet and the Srvcp trojan. The Kraken sample is used to show the general applicability of our approach based on an up-to-date example. The Srvcp trojan illustrates the challenge to find the right I/O points and how our approach can be used iteratively to find it.

4.1. Kraken Botnet

The Kraken Botnet was the largest spamming botnet in 2008 [17]. Single infected hosts have been observed of sending as much as 500.000 junk mails. Besides that, it harvests the windows address book as well as local files for email addresses and can download and execute additional programs. The bots contain a list of dynamic DNS hostnames for contacting the botnet master [15]. They subsequently try to contact each hostname via UDP and continue with the next hostname if no response is received. After a successful handshake, the a proprietary, encrypted command&control protocol is used between the infected host and the botnet controller.

For our evaluation, we have used a sample that uses the Kraken protocol version 311. Monitoring points were placed on networking function, e.g. *sendto()*, *send()*, and *recvfrom()*.

During the first 20 seconds, different TCP connection attempts were observed. After 20 seconds, the first handshake packet was sent to UDP port 447 using *sendto()*. The buffer at the *sendto()* monitoring point was located on the stack. It was contained in the stack frame located of the function that mapped to address *0x1A832C* in our dumped sample. Not answering those requests, we saw similar requests to different hosts every 10 seconds. All buffers observed originated from the same function.

The function at this address (*sub_1A832C*) contained the code excerpt that is displayed in figure 4. Function names and comments were added for presentation purposes, afterwards. The code block shows how different fields in the buffer are filled with data. The data contains keys, a seed based on processor ticks, command and subcommand, protocol version, size, and some kind of checksum. There are two functions following this block. The first function (encryptHeader), which is called at address 0x1A83F2, and its subfunctions contain some suspicious operations that are often found in encryption functions. The second function (create_new_udp_sock), which is called at address 0x1A83F7, creates a UDP socket.

With the first function being a candidate for an encryption function, we ran the botnet sample in a debugger. Stepping over the presented code shows how the data in the buffer is changed by the *encryptHeader* function. The result is then sent out using *sendto()*. A detailed manual investigation as well as a dissection

```
.text:001A83CA mov dword ptr [esp+80h+buf], eax
.text:001A83CE lea eax, [esp+80h+buf]    ; key 1
.text:001A83D2 mov [esp+80h+var_2C], edx ; key 2
.text:001A83D6 mov [esp+80h+var_28], ebx ; seed
.text:001A83DA mov [esp+80h+var_24], 1   ; cmd
.text:001A83DF mov [esp+80h+var_23], bl  ; cmd2
.text:001A83E3 mov [esp+80h+version], 137h ; ver.
.text:001A83EA mov [esp+80h+var_20], ebx ; size
.text:001A83EE mov [esp+80h+var_1C], ebx ; chks.

.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  <-------------
```

Figure 4. Kraken encryption function

from C. Pierce [15] verified this function to be the encryption.

In a second step, we modified our lab setup and spoofed a UDP response with dummy data. With a monitoring point on *recvfrom()*, we were able to find the decryption function, as well. The buffer was located in the same stack frame as the send buffer but the buffers were not identical. It was created by the same function *sub_1A832C*. Kraken is using symmetric encryption.

It took us only some minutes to identify both encryption and decryption functions with our approach. The Kraken sample ran for 20 seconds before the first UDP monitoring point was triggered. The routine related to the trigger immediately revealed the buffer creation function. Afterwards, we needed around five to ten minutes for manual investigation to identify and verify encryption and decryption candidates.

4.2. Srvcp Trojan

Srvcp is a trojan for Windows OS. It received its commands via IRC from a decicated server. It contains commands for downloading and executing arbitrary programs on the infected machine as well as commands for possible network scans and DoS attacks.

The list of IRC servers and other configuration details are stored in the encrypted file *gus.ini* in the Windows system folder. We came across the file using *FileMon* [9]. Its encrypted contents is shown in figure 5. As the characters in the file are all printable characters but do not make any sense on first sight, it is very likely to be encrypted and encoded.

Being suspicious about that file, we placed monitoring points on several Windows file access functions,

```
JexO215WuK60H7HgI.j11vh1
HBtJI.zWZtP/e1zcT/nCMAf0Osi.K.vC3lT1
ZC8YD.MBoxJ.wtPW61fAKYi1Vnu6H/yPVda.
YxPgS13wXdq0m4SMh/4NhJj0hN2gw/J/L.W1
...
```

Figure 5. Encrypted Srvcp config file

```
;;; - fscanf arguments -
.text:00403732 lea eax, [ebp+input_buf]
.text:00403738 push eax
.text:00403739 push offset "%[^\n]\n"
.text:0040373E push ebx ; File

.text:0040373F call fscanf     <----------
...
;;; - decryption arguments -
.text:004036B2 lea eax, [ebp+input_buf]
.text:004036B8 push eax
.text:004036B9 push esi

.text:004036BA call decryption <---------
```

Figure 6. Srvcp trojan decryption origin

like *ReadFile()*. Running the trojan in our tool immediately revealed a heap buffer that had been created in the function starting at *0x73D9C489*. Unfortunately, this address is outside of the address space of the considered trojan.

Examining the memory regions and call stack of the malware showed that the function was part of the *CRTDLL.dll*. This DLL implements its own, Posix-based I/O routines. Thus, the monitoring point had not revealed the data destination in the trojan, but a different I/O module that implements its own buffered input. Our tool revealed the buffer used for the implemented buffered input. The call stack contained the functions *_readbuf()* and *_fscanf()*, which are exported by the dll.

After adapting the monitoring points to the I/O routines exported by *crtdll.dll* and especially *_fscanf()*, our tool located a stack buffer in function *sub_40363D* of the trojan for being the destination of the input operation. An excerpt from this function is shown in figure 6. After the call to *_fscanf()* the buffer is immediately passed to another function, which proved to be the decryption function.

For verification we ran the binary in a debugger. The decryption candidate returned cleartext strings containing the configuration details. The number of lines in the file matched the number of different decryption results. Differences in length exist because of padding. The decrypted file shows configuration details for command & control as well as a list of 34 IRC servers to connect to. The decrypted text of the file is shown in figure 7.

The time needed to determine the decryption routine

```
NICK=mikey
CHANNEL=mikag soup
SOUPCHANNEL=alphasoup ah
SERVER0=irc.mcs.net:6666
...
```

Figure 7. Decrypted Srvcp config file

in this sample took longer than that of the Kraken botnet. The main reason is the intermediate, buffering I/O. Placing a monitoring point at the Windows API functions immediately revealed a buffer created from a function outside of the trojan code. The manual investigation of the call stack and intermediate Dll took around 5 minutes. Placing a new monitoring point at the I/O interface took another 10 minutes. Running our tool on the intermediate I/O interface immediately located the function containing the call to the decryption routine. The overall time for finding and verification was approximately 20 to 30 minutes. Knowing the right I/O interface, it took less than 5 minutes.

4.3. Evaluation Summary

We used the two examples above to show the applicability of our approach for finding crypto functions in malware. We were able to identify the buffer creation in less than a minute each, which we assume is hardly possible with manual investigation. As crypto functions can be of arbitrary structure, manual verification has to be performed on the identified functions, afterwards. This step takes far more time.

A challenge is to identify the right I/O interface for setting monitoring points. Our own implementation uses standard API functions by default and can easily be extended to other I/O interfaces. The finding of intermediate I/O interfaces can also be achieved using our approach, as described with the second sample.

The software design used in malware is very heterogeneous. We have successfully applied our approach to two different examples. Not mentioned here is the application to the infamous Storm worm and SdBot variants because they were similar efficient as the application to Kraken. We therefore conclude that our approach is a valuable means in the toolbox for malware analysts.

5. Discussion

When an approach as the presented becomes public it may be invalidated. This is normal in the ongoing arm's race. Still, we see a large benefit for malware analysts in the publication. In the following, we will discuss strengths and weaknesses as well as implications of intentional assumption breaches by malware authors. We also present alternative ways for detecting crypto routines.

5.0.1. Dependence on I/O interface. We have shown how easy and fast crypto routines can be found with our approach if the right I/O routines are monitored. Our assumption that malware is using the OS for I/O does not always have to be true. Malware may use custom system calls. Our approach fails for such custom tailored I/O. The results of such an architecture is much larger binaries and is architecture dependence. Therefore, such malware will be less generic and may not spread well.

5.0.2. Buffer creation and coding functions. Our approach is based on the observation that coding is performed close the buffer creation. After reading this, a malware author may choose to specifically create buffers far away from the coders that make use of the buffers. This is a possible but rather unintuitive development strategy, which complicates the software design and may increase the memory usage.

Such a software design complicates maintainability, increases the risk for bugs, and discards modularity. Since malware and especially botnet development is becoming more and more professional [10], it has to be efficient. It is questionable whether malware developers would take this step instead of improving packers and obfuscation techniques.

In case the change in design is taken, it would still be possible to determine usage sets and definition sets. This results in a larger set of candidates to analyze but gives direct hints on which functions to investigate.

5.0.3. General assumptions. A general problem is that the creation functions of global buffers does not reveals where it is used. Using global buffers has implications on the software design and memory usage as discussed above. In addition, it significantly complicates the organization and synchronization of multiple threads. In this case this step is taken, memory monitoring can be used as decribed above, too.

Our approach assumes that a single function is responsible for performing or initiating the coding. This could be split into multiple functions. It is likely that those functions would still be close but complicates the analysts' manual verification of the coding function.

Monitoring memory accesses may be used in cases, in which our assumptions do not hold. For now, malware breaking with our assumptions must be very

sophisticated. Such a sample would be a very interesting study object for malware analysts.

6. Conclusions and Future Work

We have shown the applicability of our approach. With a practical implementation, we were able to identify the crypto routines of the *Kraken* botnet and the decryption function of the *Srvcp trojan*, as well as others not shown in detail. We therefore conclude it to be a valuable means in the malware analysis toolchain.

The examples presented are not necessarily representative for all malware in-the-wild. A more comprehensive evaluation is needed in order to harden the assumptions and to verify the wider usability of our approach. This is includes larger numbers of malware.

Furthermore, it would be beneficial to automate more of the analysis process in order to reduce manual work. This includes a pre-analysis for finding possible I/O interfaces and adding monitoring points for tracking accesses on buffers.

Acknowledgments

The authors would like to thank the anonymous reviewers of this paper for discussions and comments. We are also thankful for the people who supported us and gave valuable suggestions for our work.

References

[1] P. Amini, *PyDbg - A pure Python win32 debugging abstraction class*, http://pedram.redhive.com/PyDbg/, last visit: Oct. 2009

[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/, last visit: Oct. 2009

[4] 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

[5] F. Bellard, *QEMU, a Fast and Portable Dynamic Translator*, USENIX Annual Technical Conference, 2005

[6] N. Brulez, *Unpacking Storm Worm*, http://securitylabs.websense.com/content/Blogs/3127.aspx, last visit: Aug. 2009

[7] M. Christodorescu et al., *Semantics-aware malware detection*, IEEE Symposium on Security and Privacy, 2005

[8] D. Dittrich and S. Dietrich, *Command and control structures in malware*, Usenix magazine, Vol. 32, No. 6, Dec. 2007

[9] R. Russinovich and B. Cogswell, *Windows Sysinterals*, http://technet.microsoft.com/en-us/sysinternals/default.aspx, last visit: Oct. 2009

[10] D. Fisher, *Storm, Nugache lead dangerous new botnet barrage*, Article, http://searchsecurity.techtarget.com/news/article/0,289142, sid14_gci1286808,00.html, last visit: Oct. 2009

[11] M. Hale Ligh, *Kraken Encryption Algorithm*, Blog, http://mnin.blogspot.com/2008/04/kraken-encryption-algorithm.html, last visit: Oct. 2009

[12] Immunity Inc., *Immunity Debugger*, http://www.immunitysec.com/products-immdbg.shtml, last visit: Oct. 2009

[13] 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

[14] M. Oberhumer and L. Molnar. The Ultimate Packer for eXecutables (UPX), http://upx.sourceforge.net/. Last visit: Oct. 2009

[15] C. Pierce, *Owning Kraken Zombies, a Detailed Dissection*, Blog on DVLabs,http://dvlabs.tippingpoint.com/blog/2008/04/28/owning-kraken-zombies, last visit: Oct. 2009

[16] 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

[17] P. Royal, *On the Kraken and Bobax Botnets*, Whitepaper, Damball, Apr. 2008

[18] Shadowserver Foundation, *ShadowServer Homepage*, http://shadowserver.org, last visit: Oct. 2009

[19] P. Szor, *The Art of Computer Virus Research and Defense*, Addison-Wesley, Feb. 2005

[20] Symantec Coorp. *Symantec Internet Security Threat Report Volume XIII*, Whitepaper, Apr. 2008

[21] G. Wicherski, *botsnoopd - Sniffing on Botnets*, Blog, http://blog.oxff.net/2006/10/botsnoopd-sniffing-on-botnets.html, last visit:Oct. 2009

[22] C. Willems and T. Holz and F. Freiling, *Toward Automated Dynamic Malware Analysis Using CWSandbox*, Ieee Security & Privacy, 2007

[23] O. Yuschuk, *OllyDbg Debugger*, http://www.ollydbg.de/, last visit: Oct. 2009