Automatic Detection and Analysis of Encrypted Messages in Malware

Ruoxu Zhao(✉), Dawu Gu, Juanru Li, and Yuanyuan Zhang

Department of Computer Science and Engineering,
Shanghai Jiao Tong University, Shanghai, China
zhaoruoxu@gmail.com, dwgu@sjtu.edu.cn

Abstract. Encryption is increasingly used in network communications, especially by malicious software (malware) to hide its malicious activities and protect itself from being detected or analyzed. Understanding malware's encryption schemes helps researchers better analyze its network protocol, and then derive the internal structure of the malware. However, current techniques of encrypted protocol analysis have a lot of limitations. For example, they usually require the encryption part being separated from message processing which is hardly satisfied in today's malware, and they cannot provide detailed information about the encryption parameter such as the algorithm used and its secret key. Therefore, these techniques cannot fulfill the needs of today's malware analysis.

In this paper, we propose a novel and enhanced approach to automatically detect and analyze encryption and encoding functions within network applications. Utilizing dynamic taint analysis and data pattern analysis, we are able to detect encryption, encoding and checksum routines within the normal processing of protocol messages without prior knowledge of the protocol, and provide detailed information about its encryption scheme, including the algorithms used, secret keys, ciphertext and plaintext. We can also detect private or custom encryption routines made by malware authors, which can be used as signature of the malware. We evaluate our method with several malware samples to demonstrate its effectiveness.

Keywords: Network protocols · Encryption detection · Data analysis · Reverse engineering

1 Introduction

Today protocol reverse engineering is widely used in many security applications, especially in malware detection and analysis. To fully understand the intention and behavior of malware, security analysts usually have to obtain detailed network protocol information. However, current circumstance of the wider use of

Supported by the National Science and Technology Major Projects (Grant No.: 2012ZX03002011-002), the National Key Technology R&D Program (2012BAH426B02) and NSFC under Grant No. 61103040.

sophisticated encryption schemes in malware's network communication renders it very difficult to analyze the protocol directly. Several techniques were proposed to solve this problem [3,15,19], however these techniques have several critical weaknesses which make them inadequate in the analysis of current malware. First, these techniques usually require encryption or decryption to be in a separated phase from normal processing of protocol messages. This condition is hardly satisfied in today's malware, which usually uses several layers of different encryption or encoding schemes. Second, they usually detect encryption routines by the ratio of bitwise or arithmetic instructions. This condition also cannot be met when malware uses weaker but simpler encoding scheme instead of encryption, or with the existence of obfuscation. And third, these techniques only detect the existence of encryption routines but not the parameters such as the secret key. The lack of this information makes it very difficult to give a comprehensive view of the malware's internal structure.

We propose an in-depth approach to detect and analyze the encryption, encoding and checksum routines within a program using dynamic taint analysis [16] and dynamic data pattern analysis [23], and then uncover the complete structure of malware's protocol messages. First we construct the hierarchical structure of a protocol message using its procedure-level execution context. Then, we perform dynamic taint analysis on the possible procedures to discover encryption and encoding routines. After that, dynamic data pattern analysis is used to reveal the parameters of encryption or decryption, and to produce possible sub-messages. At last, we reconstruct sub-messages to the original protocol message to provide comprehensive analysis result of encrypted protocol content.

Some of our contributions are listed below.

– We use dynamic taint analysis as the primary tool to locate encryption or encoding within the message processing, eliminating the former requirement of the separation of encryption and message processing. We also propose methods to distinguish different layers of encryption, revealing the internal structure of encrypted message.
– Dynamic data pattern analysis is used to extract the high-value parameters of encryption, including the algorithm used, secret keys, ciphertext, plaintext, etc. This information is valuable to security analysts, and can be used as signature to malware detection and classification.
– We provide methods to detect non-public or custom made encryption or encoding routines used in malware's protocols automatically, with no prior knowledge of the malware. Some of the parameters used in custom algorithms can be extracted at the same time.
– We use entropy metrics and data characteristics as powerful supplements to distinguish encryption routines (focusing on confusion and diffusion) and encoding routines (focusing on transformation). The entropy metrics provide a convenient way to discover the underlying nature of detected algorithm.
– We evaluate our method with custom programs as well as several real world malware samples to show the effectiveness of our approach, including ZeuS P2P botnet, Mega-D botnet, Storm botnet, etc.

2 Background and Related Work

Protocol reverse engineering has gained significant attention in recent years. Polyglot [4] took the first step to automatically reverse engineer the message format using dynamic program analysis techniques. Utilizing dynamic taint analysis, it can discover different kinds of message fields such as direction fields and keywords. AutoFormat [14] took a step further to detect the structural information about protocol messages. The work by Wondracek et al. [20] combined multiple messages together to deduce the internal structure of messages. All these approaches worked on plain unencrypted messages. There are other systems [6,7] proposed to automatically infer the protocol state machine using program analysis techniques. The first effort to reverse engineer encrypted protocol content automatically was made by the system ReFormat [19], and then Dispatcher [3]. They took similar approaches with the assumption of the separation of decryption and message processing, and used instruction characteristics to detect the decryption function. The lack of flexibility and detailed information about cryptographic parameters limited their usages. There are other approaches [8,17,18] that used network traces to analyze protocols. These approaches are mostly probabilistic and require prior knowledge about the protocol, which are less accurate than the program analysis based approaches.

Automatic detection and analysis of cryptographic algorithms is also a hot topic of security research in recent years. Gröbert's work [11] took the first step to detect cryptographic primitives in software using dynamic data analysis techniques. Zhao et al. [22,23] extended this work using dynamic data pattern analysis, which is more effective and accurate. The system Aligot [5] focused on detecting cryptographic algorithms in obfuscated software using loop detection techniques. CipherXRay [13] used the avalanche effect of cryptographic algorithms and dynamic taint analysis to detect the input-output dependency of cryptographic algorithms.

In this paper, we combine protocol reverse engineering techniques with cryptographic algorithm detection techniques to provide in-depth and comprehensive detection and analysis of encrypted protocol messages. We also extend the cryptographic algorithm detection to encoding functions and propose information entropy based metrics to distinguish encryption and encoding functions. Our approach requires no prior knowledge of the protocol or the algorithms, and tries to detect generic patterns and reveal complete structures of encrypted messages.

3 System Description

3.1 System Overview

Our system is designed to automatically extract the detailed internal structure and data protection schemes of an encrypted protocol message. Given a sample of a program, our system runs it within the execution monitor, and outputs the complete structure of encrypted network messages, with detailed information about its encryption, hashing and checksums used in different layers of the processing of the message. To achieve this goal, our system takes the following steps: (a) Run the program in the execution monitor (emulator) and obtain runtime traces containing low-level context data. (b) Upon receiving a message, our system analyzes the procedure call hierarchy and the message structure. (c) Analyze possible encryption (or decryption), encoding (or decoding) and checksum routines with the message processing. The decrypted (or decoded) data is extracted as sub-messages. (d) We continue step (b) on all sub-messages and subsequent messages until the analysis is complete, and output all analysis result. An overview of the architecture of our system is shown in Fig. 1.

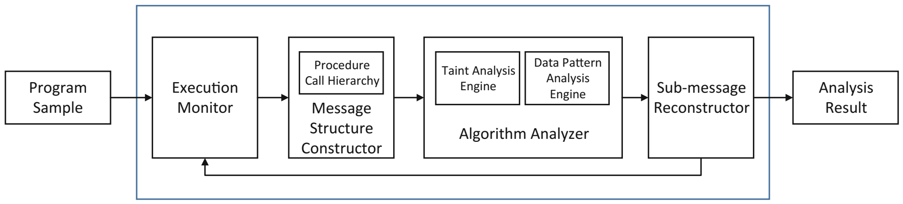

Fig. 1. System architecture

Dynamic Execution Monitoring. In order to obtain program runtime data, the program to be analyzed is run in a formerly available program emulation system [22]. Fine-grained information, including CPU instructions, register values, memory accesses and parameters of system API calls, can be visited conveniently. All networking APIs are hooked to notify our system when a network message is received, including the message data and context information. All subsequent processing of the message is monitored by our system to analyze possibly encrypted message content. The analysis and program execution are done simultaneously for better performance.

3.2 Message Structure Inference

The first step during analysis is to construct the internal hierarchical structure of a message into a tree structure. The message tree is later used to reconstruct the meaningful hierarchy of an encrypted message and its sub-messages. To achieve this, our system is based on the simple observation: Most functionality units of a program are implemented as procedures, especially encryption routines, hashing routines and checksums. This fact accords with the software engineering principle of modularized design, and is common in today's software even in malware.

During program execution, our message structure analyzer maintains a virtual call stack to track current procedure call hierarchy. Whenever message data is accessed (to byte granularity), we record its call stack context, and append it

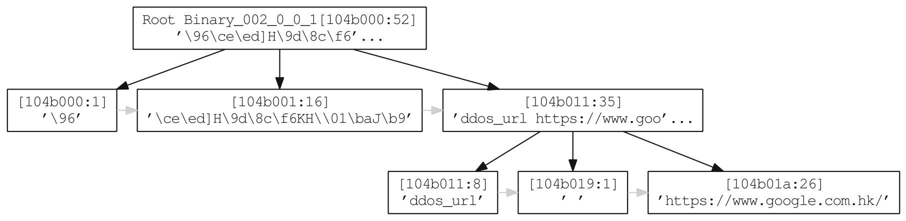
Root Binary_002_0_0_1[104b000:52]
'\96\ce\ed]H\9d\8c\f6'...
[104b000:1]
'\96'
[104b001:16]
'\ce\ed]H\9d\8c\f6KH\\01\baJ\b9'
[104b011:35]
'ddos_url https://www.goo'...
[104b011:8]
'ddos_url'
[104b019:1]
' '
[104b01a:26]
'https://www.google.com.hk/'

Fig. 2. A sample message tree

to the message tree properly. All message data with the same context are merged in the final tree. A sample of constructed message tree is shown in Fig. 2.

The sole purpose of message structure reconstruction is for the reconstruction of decrypted or decoded sub-messages. We can also analyze message field format in this step, which has been extensively studied [3,4,9,14,20]. So here we omit the analysis of message field format and focus on encrypted data.

3.3 Data-Oriented Analysis and Algorithm Detection

After the inference of message structure, we conduct data-oriented analysis on each of the possible procedures. The data analysis mainly includes dynamic data taint analysis for the detection of data dependency, and dynamic data pattern analysis for the detection of specific algorithms. We'll discuss this in detail in Sect. 4.

3.4 Sub-message Generation

A sub-message is an encrypted or encoded partial message which is embedded within its parent message. A sub-message often indicates a new layer of the original message, which usually has different semantic meanings and is processed in different routines. After each successful detection of an algorithm in the previous step, we spot the beginning of a sub-message starting at the completion point of the algorithm. We then analyze the sub-message recursively until all sub-messages are detected and analyzed.

3.5 Message Reconstruction

Upon the completion of analysis of each sub-message, we discard the original analysis of the sub-message in its parent message and append the newly generated sub-message. The final result is a tree-like structure where all layers of the processing of the original message and the conversions of message data are clear to analysts. Examples of our final result are shown in Sect. 5.

4 Data Dependency and Data Pattern Analysis

On acquiring the procedure call hierarchy, we are able to conduct data flow analysis on each of the procedure call that possibly contains encryption, encoding or checksum. For any deterministic algorithm, we argue that the relationship between its input and output data is uniquely decided, which means the output of an algorithm is predetermined for any fixed input data. This is the key concept in the whole analysis phase.

For each trace of procedure call, we conduct dynamic taint analysis first to test if the input-output data dependency satisfies the characteristics of a certain algorithm. We use every byte of the received message as the taint source, and track the flow of the message in each procedure. For those whose output data is tainted, we perform further analysis on it to test if a specific algorithm exists. This approach has the limitation that not all input parameters of an algorithm can be tainted in the same procedure. This limitation can be resolved using dynamic data pattern analysis.

To certify the existence of an algorithm, we conduct dynamic data pattern analysis on selected procedures. For each procedure, all possible combinations of input and output data are iterated to see if the data pattern is satisfied. In the meantime, the parameters of an algorithm can be extracted as side products. We further discuss the analysis details using these techniques for different kinds of algorithms.

4.1 Detecting Block Ciphers

For block ciphers, we use the avalanche effect as theoretical basis for our analysis. The avalanche effect says that flipping a single bit in input results in about half of the output bits being flipped in a well-designed block cipher. In fact, if we consider the situation for each byte, the possibility of an output byte not being affected by any one of the input byte is so low that it cannot happen in one experiment, even for a short 64-bit block [13]. Hence, we argue that every byte in the output data of a block cipher is dependent on every byte of its input data. Dynamic taint analysis is just the right tool to detect this dependency. Whenever we found a block of output data being completely tainted by a block of input data, we further analyze this block using data pattern analysis to verify. A demonstration of the data dependency of block ciphers is shown in Fig. 3.

Key Scheduling. Before actual encryption of a block cipher, a program must perform key scheduling first to generate the sub-keys. In most of the malware, the secret key is embedded in its binary, thus won't be tainted in procedure input. We use the strategy of retainting the input using some special taint tags, and test if every byte of the possible sub-key is tainted by a subset of input key. Then data pattern analysis is performed to verify its belonging to an algorithm.

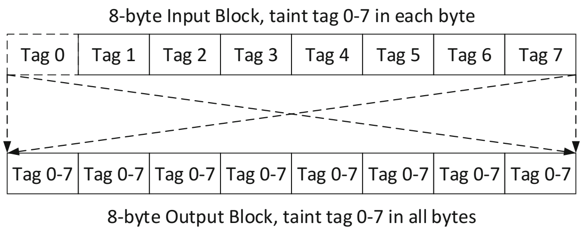

Fig. 3. Block cipher data dependency

Modes of Operation. With the power of taint analysis and data pattern analysis, detecting modes of operation of block ciphers is fairly straightforward. We demonstrate how some of modes of operation is detected.

Electronic Codebook (ECB). There's no initialization vector (IV) in ECB mode, so it can be detected using taint analysis only.

Cipher-Block Chaining (CBC). In CBC mode, the IV is first XORed with plaintext, and then encrypted to produce the ciphertext. We first detect the block encryption and get the actual input, which is the XORed result. We then perform XOR with the tainted message input, and get the IV used in encryption.

Cipher Feedback (CFB) and Output Feedback (OFB). In CFB and OFB mode, the IV is encrypted and then XORed with the plaintext to produce the ciphertext. For most of the malware implementations, the IV is embedded in the malware's binary, just like the secret key. So we detect the IV first using data pattern analysis in the first block, and detect subsequent encryptions using taint analysis.

Counter (CTR). In CTR mode, each block input can be untainted in our analysis, so we detect all blocks using data pattern analysis.

4.2 Detecting Stream Ciphers

Most stream ciphers don't have a strong data dependency like the block ciphers. The plaintext of stream ciphers is usually XORed with some value to produce the ciphertext. Hence, each byte of the ciphertext of stream ciphers must be tainted by at least the corresponding byte in plaintext. We then use data pattern analysis to verify the data dependency.

Key Scheduling. Stream ciphers like RC4 may also have a key scheduling process. This process is actually quite similar to the key scheduling of block ciphers where each byte of the scheduled key is tainted by a part of the secret key. As most of the secret keys are untainted in our analysis, we also detect them using data pattern analysis (Fig. 4).

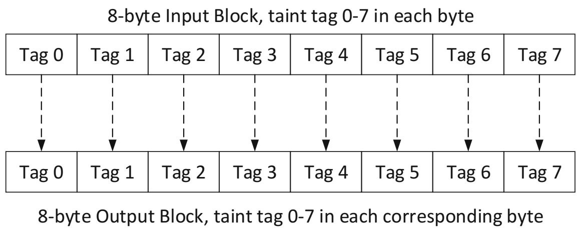
8-byte Input Block, taint tag 0-7 in each byte
Tag 0
Tag 1
Tag 2
Tag 3
Tag 4
Tag 5
Tag 6
Tag 7
Tag 0
Tag 1
Tag 2
Tag 3
Tag 4
Tag 5
Tag 6
Tag 7
8-byte Output Block, taint tag 0-7 in each corresponding byte

Fig. 4. Stream cipher data dependency

4.3 Detecting Hash Functions

Like block ciphers, well designed hash functions also have the property of the avalanche effect. Unlike block ciphers, they produce the same length of output regardless of the length of their input. The strong data dependency is a notable signature of hash functions. We take similar approaches as before to detect hash functions.

4.4 Detecting Encoding Functions

The encoding functions we refer to are used to transform data into another format for network transmission, such as the widely used Base64. They don't have cryptographic characteristics and their encoded message can be decoded easily. However, they still remain some weak data dependency which is similar to stream ciphers, and can be used as a signature to detect these functions. Some of the encoding functions produce different length of output to the input, so we modify the method used in stream ciphers to handle variable length of output data.

Some of the malware authors don't care much about the security of their encryption functions, and just use a simpler encoding scheme instead. However, this approach is usually good enough to bypass most of the intrusion detection systems or black-box analysis [10]. We further describe this situation in Sect. 4.6.

4.5 Detecting Checksums

Malware usually uses checksums to detect errors or modifications of network data. Unlike hash functions, checksums usually have a small length and can be easily forged. Most of the checksums don't exceed 32-bit, and can fit into a register of x86 CPUs. Therefore, we detect checksum routines using register values as well as memory data, to see if a small-size datum is dependent on the whole input block. Data pattern analysis includes common checksum algorithms like CRC-32, Alder-32, bitwise XOR and arithmetic sum.

4.6 Inferring Private Algorithms

Apart from using standard algorithms, many malware authors choose to use custom or modified algorithms to avoid detection. Doing so further increases the difficult to reverse engineer the malware samples or to analyze network traffic. It's of great importance to detect these kinds of algorithms, since they provide valuable information to security analysts. Here we discuss the techniques we use to detect custom or private encryption and encoding algorithms.

Although the details of private algorithms remain unknown prior to our analysis, they do exhibit many of the features of standard algorithms mentioned earlier. We still use data analysis techniques as weapons to uncover the nature of these algorithms. Since many of the details are unavailable to our analysis, we have to introduce extra techniques to extract them. We introduce information entropy based metrics for algorithm classification, and discuss in detail about the detection of each kind of algorithms.

Entropy Metrics. In information theory, entropy is used to quantify the expected value of the information contained in a message. We use Shannon entropy here for the measurement of the randomness of data. Given a block of binary data d(length $n > 0$), and $c_i (0 \leq i < 256)$ denoting the total occurrences of byte i in d, we defined the normalized entropy $H(d)$ as:

$$H(d) = -\frac{\sum_{i=0}^{255} \frac{c_i}{n} \log_2 \frac{c_i}{n}}{\log_2 n} (0 < H(d) \leq 1)$$

Unencrypted messages or texts usually have a low $H(d)$ value, yet encrypted binary data tends to have a high (nearly 1) $H(d)$ value. In this way, we further define the quotient of the entropy of the output data d_o and the input data d_i for a procedure trace p as:

$$Q(p) = \frac{H(d_o)}{H(d_i)}$$

For short messages, the $H(d)$ or $Q(p)$ value may not be meaningful, as information entropy is a statistical concept. However, our experiments suggest that for medium length (tens or hundreds of bytes) messages, the $H(d)$ and $Q(p)$ value can be used to measure the randomness indeed. We'll discuss the usages of entropy metrics below, and show our experiment results in Sect. 5.

Block Ciphers. For the detection of private block ciphers, we use taint analysis to discover the data dependency. The key scheduling and modes of operation (other than ECB), however, cannot be easily detected because of the unavailability of data pattern analysis. For most of block cipher decryptions, the Q value is usually below 0.8, which is the lowest among all kinds of algorithms.

Stream Ciphers and Encoding Functions. Private stream ciphers and encoding functions exhibit almost the same features in our analysis, as they all show byte-to-byte mapping in our data dependency analysis. The first difference is that encoding (or decoding) functions may have input and output data with different lengths, yet in stream ciphers the length is always the same. Another difference is that the Q value of stream ciphers is generally lower than encoding functions. We use an empirical Q value of 0.9 as the boundary, where procedures of Q below 0.9 as stream ciphers and above as encoding functions.

Many malware authors use XOR-based encryption schemes. There are mainly two kinds of XOR-based encryptions: one is chained-XOR, where each byte is XORed with previous value to get the encrypted byte; the other one is keyed-XOR, where each byte is XORed with a short custom-scheduled key (similar to RC4). In our analysis, we treat the chained-XOR as encoding, whereas the keyed-XOR as encryption, which is supported by the experiment results of entropy metrics.

Hash Functions and Checksums. Hash functions have very distinguishable data characteristics. They have a strong data dependency, and their Q value is usually above 1.1. Checksum routines share the data characteristics of hash functions, except that their output is too short for entropy metrics.

5 Implementation and Evaluation

We implemented our system as a plugin module of the LochsEmu emulator [21,22], which can be used to analyze 32-bit Windows programs. This infrastructure enables us to conduct efficient and convenient data-oriented analysis. Other available frontend options include QEMU [2] and PIN [1], but they usually require tracing the intermediate result into hard disk first, which introduces considerable performance deduction. We implemented the taint analysis engine, the data pattern analysis engine and algorithm analyzers as loosely coupled submodules with about 10k lines of C++ code.

We chose some custom made programs for test purposes, and some variants or self-compiled versions of real-world malware (botnet) for validation and evaluation, including ZeuS P2P botnet, Mega-D, Storm, ZeroAccess, Festi and Mariposa. Most of the test samples are botnet clients which have extensive network communication. Our target algorithms include block cipher DES (ECB, CBC and CFB modes), stream cipher RC4 and chained XOR, hash function MD5, encoding function Base64 and checksum functions CRC32 and Alder32. We also detect private or custom algorithms which we call generic symmetric ciphers, generic stream ciphers and generic encoding/decoding algorithms. An overview of our evaluation result is in Table 1.

Table 1. Evaluation result overview

Sample	Result
Mega-D	DES key schedule; DES-ECB decryption
ZeuS P2P	Chained XOR; MD5; RC4 key schedule; RC4 decryption
Storm	Generic decoding; Checksum (XOR 8-bit); Checksum (ADD 8-bit)
ZeroAccess	Generic stream decryption; Checksum (CRC32)
Festi	Generic stream decryption
Mariposa	Generic stream decryption
Test Sample 1	DES key schedule; DES-CBC decryption; DES-CFB decryption
Test Sample 2	Base64 decoding

5.1 Entropy Metrics

We use entropy metrics for the distinguishing of encryption and encoding. The Q values of the samples above are shown in Fig. 5.

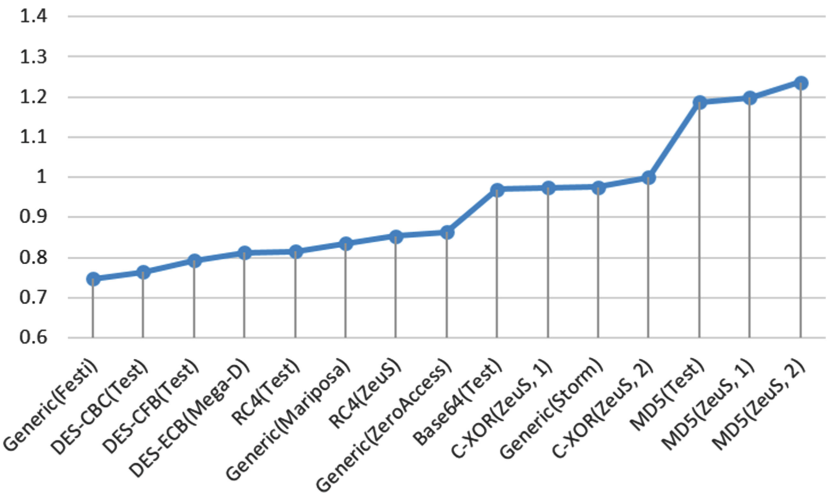

Fig. 5. Values of Q function

There's a huge gap between MD5 hash function and other algorithms because MD5 acts like encryption which makes entropy higher, while others are decryption or decoding which reduces entropy. Generally, we treat procedures with $Q > 1.1$ as encryption functions or hash functions. We set the Q range $0.9 < Q \leq 1.1$ for encoding and decoding. All detected decoding routines fell into this range with Q values near 1.0. All decryption routines met the condition that $Q \leq 0.9$, generally within the range from 0.75 to 0.85. These decryption routines include both symmetric ciphers and stream ciphers.

5.2 Case Study

ZeuS P2P Botnet. The C2 message (type 0xCC) is the most complicated encrypted message we analyzed. It contains three layers of encryption or encoding, and two MD5 hash data blocks used as checksums. The layout of a ZeuS C2 message is shown in Fig. 6, and the complete structure of a ZeuS message is in Appendix A.

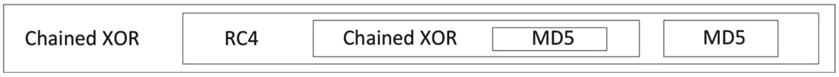

Fig. 6. ZeuS message layout

The chained XOR algorithm is a fundamental algorithm used in ZeuS botnet, and the outmost layer of every ZeuS message is encoded using chained XOR. This algorithm uses a single fixed byte as initialization byte, and each byte is XORed with the previous byte to get the encoded byte, as shown below.

```
void _visualEncrypt (void *buffer, DWORD size)
{
    for (DWORD i = 1; i < size; i++)
        ((LPBYTE)buffer)[i] ^= ((LPBYTE)buffer)[i - 1];
}
```

The result of entropy metrics of the XOR algorithm shows that it's more of an encoding algorithm rather than encryption, which is predictable since chained XOR only introduces very limited security. However, this simple scheme is enough to evade most black-box network trace based analysis.

The C2 message also has a layer of RC4 encryption. It uses standard RC4 algorithm, and we're able to successfully detect the RC4 key schedule as well as the encryption. The Q function value of the RC4 procedure is about 0.85, which is a little bit high for the reason that the decrypted message still contains encoded binary data. Two MD5 hashes are used in the message to check the message's integrity. They're both successfully detected and their occurrences are linked with MD5's output, shown in Appendix A.

With the information above, security analysts can easily grasp the high-level structure of a large complicated message, and focus on an interesting point to do further manual analysis. This information can also be used to study the evolvement of a particular malware.

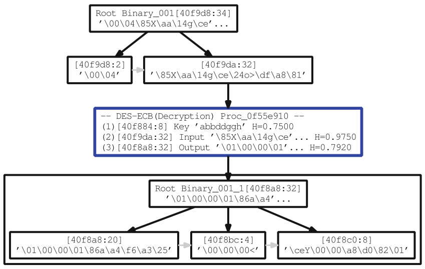
Root Binary_001[40f9d8:34]
'\00\04\85X\aa\14g\ce'...
[40f9d8:2]
'\00\04'
[40f9da:32]
'\85X\aa\14g\ce\24o>\df\a8\81'
-- DES-ECB(Decryption) Proc_0f55e910 --
(1)[40f884:8] Key 'abbddggh' H=0.7500
(2)[40f9da:32] Input '\85X\aa\14g\ce'... H=0.9750
(3)[40f8a8:32] Output '\01\00\00\01'... H=0.7920
Root Binary_001_1[40f8a8:32]
'\01\00\00\01\86a\a4'...
[40f8a8:20]
'\01\00\00\01\86a\a4\f6\a3\25'
[40f8bc:4]
'\00\00\00<'
[40f8c0:8]
'\ceY\00\00\a8\d0\82\01'

Fig. 7. The Mega-D message

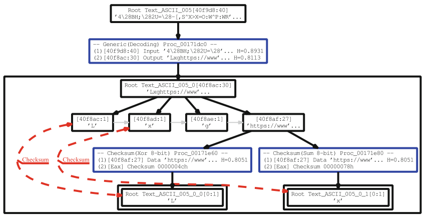
Root Text_ASCII_005[40f9d8:40]
'4\28BH;\282U=\28-[,S^X>X=O:W^P:WR'...
-- Generic(Decoding) Proc_00171dc0 --
(1)[40f9d8:40] Input '4\28BH;\282U=\28'... H=0.8931
(2)[40f8ac:30] Output 'Lxghttps://www'... H=0.8113
Root Text_ASCII_005_0[40f8ac:30]
'Lxghttps://www'...
[40f8ac:1]
'L'
[40f8ad:1]
'x'
[40f8ae:1]
'g'
[40f8af:27]
'https://www'...
Checksum
Checksum
-- Checksum(Xor 8-bit) Proc_00171e60 --
(1)[40f8af:27] Data 'https://www'... H=0.8051
(2)[Eax] Checksum 0000004ch
-- Checksum(Sum 8-bit) Proc_00171e80 --
(1)[40f8af:27] Data 'https://www'... H=0.8051
(2)[Eax] Checksum 00000078h
Root Text_ASCII_005_0_0[0:1]
'L'
Root Text_ASCII_005_0_1[0:1]
'x'

Fig. 8. The storm message

Mega-D Botnet. The Mega-D botnet uses DES-ECB encryption to protect its network communication. A sample analysis result of Mega-D message is shown in Fig. 7.

The Mega-D messages begin with a two-byte field specifying the number of DES blocks. The example above shows that there're 4 blocks in this message. The decryption is detected including its secret key, ciphertext and plaintext. The decrypted plaintext is further divided into several parts, and in this example they indicate some ID fields.

One thing worth mentioning is that Mega-D encrypts its message with embedded secret key 'abcdefgh', however the detected secret key is 'abbddggh'. That's because 'abbddggh' is the parity-fixed value of 'abcdefgh', and obviously they produce the same S-box for DES decryption after fixing parity.

Storm Botnet. The Storm botnet uses a encoding algorithm which is similar to Base64 [12]. The plaintext is first padded and separated into 6-bit units and then each unit is added with 0x21 to get the encoded byte. The decoded data contains two bytes for checksum, one is 8-bit sum modulo 256 and the other is 8-bit bitwise XOR, as shown in Fig. 8.

ZeroAccess, Festi and Mariposa. The encryption schemes for these three botnet samples are all custom XOR-based stream ciphers. ZeroAccess uses a custom scheduled 256-byte S-box, which is similar to RC4. It also uses a CRC32 checksum within the decrypted message to validate integrity, shown in Fig. 9.

Festi uses a 4-byte embedded key, and performs bitwise XOR of the plaintext and the key every 4 bytes to get the ciphertext. Mariposa uses a 2-byte key derived from the plaintext. It's easy to see that these encryption algorithms are not cryptographically secure, but very easy to implement and use. There's really no point in using the encryption algorithms that are proven to be secure under the circumstance that the software binary can be obtained and analyzed, so this kind of simple encryption schemes is widely used by malware authors.

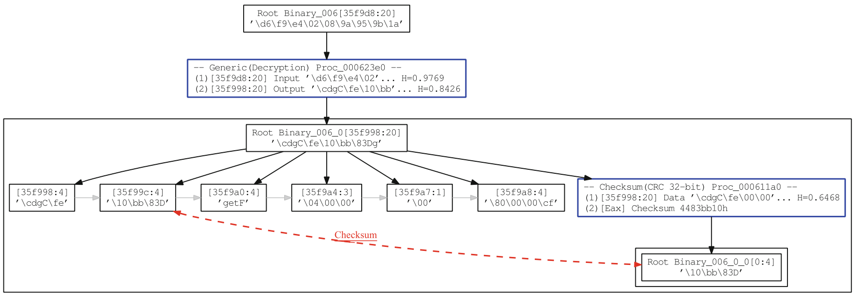
Root Binary_006[35f9d8:20]
'\d6\f9\e4\02\08\9a\95\9b\1a'
-- Generic(Decryption) Proc_000623e0 --
(1)[35f9d8:20] Input '\d6\f9\e4\02'... H=0.9769
(2)[35f998:20] Output '\cdgC\fe\10\bb'... H=0.8426
Root Binary_006_0[35f998:20]
'\cdgC\fe\10\bb\83Dg'
[35f998:4]
'\cdgC\fe'
[35f99c:4]
'\10\bb\83D'
[35f9a0:4]
'getF'
[35f9a4:3]
'\04\00\00'
[35f9a7:1]
'\00'
[35f9a8:4]
'\80\00\00\cf'
-- Checksum(CRC 32-bit) Proc_000611a0 --
(1)[35f998:20] Data '\cdgC\fe\00\00'... H=0.6468
(2)[Eax] Checksum 4483bb10h
Checksum
Root Binary_006_0_0[0:4]
'\10\bb\83D'

Fig. 9. The ZeroAccess message

6 Conclusion

In this paper, we present a novel encrypted protocol analysis technique that can be used to reveal the complicated encrypted message structure of today's malware. We first infer the message structure using runtime context, and reconstruct the message into a tree hierarchy. We then use data-oriented analysis techniques including taint analysis and data pattern analysis to detect encryption, encoding and checksum routines and extract possible sub-messages. At last, we analyze recursively on all sub-messages to uncover the complete structure of an encrypted message.

With the power of dynamic taint analysis and dynamic data pattern analysis, we're able to detect public encryption and encoding algorithms such as DES, RC4 and Base64. We can also locate possible custom or private algorithms, which are widely used by malware. The use of entropy metrics makes it possible to find out the data characteristics and distinguish encryption and encoding functions. We evaluate our technique using 6 malware samples as well as some custom made test programs. The evaluation result shows that our technique is reliable and accurate to detect both public and private algorithms, and to extract the complete structure of messages with complicated encryption and encoding schemes.

A ZeuS Botnet Message Format

See Fig. 10.

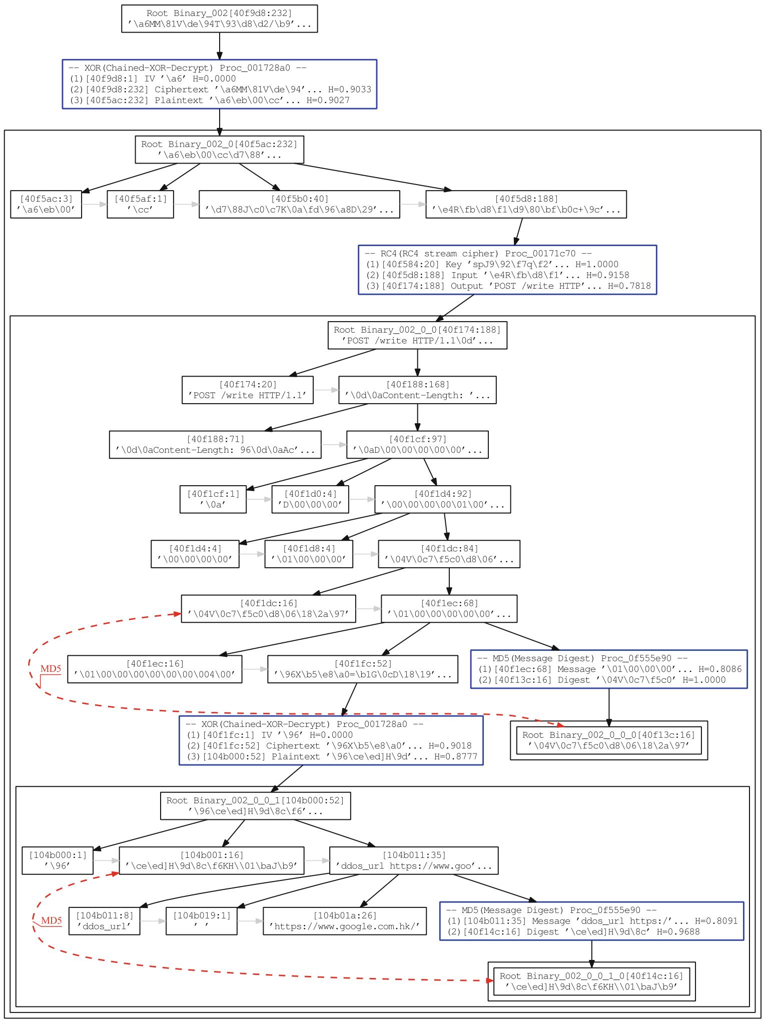
Root Binary_002[40f9d8:232] '\a6MM\81V\de\94T\93\d8\d2/\b9'...
-- XOR(Chained-XOR-Decrypt) Proc_001728a0 --
(1)[40f9d8:1] IV '\a6' H=0.0000
(2)[40f9d8:232] Ciphertext '\a6MM\81V\de\94'... H=0.9033
(3)[40f5ac:232] Plaintext '\a6\eb\00\cc'... H=0.9027
Root Binary_002_0[40f5ac:232] '\a6\eb\00\cc\d7\88'...
[40f5ac:3] '\a6\eb\00'
[40f5af:1] '\cc'
[40f5b0:40] '\d7\88J\c0\c7K\0a\fd\96\a8D\29'...
[40f5d8:188] '\e4R\fb\d8\f1\d9\80\bf\b0c+\9c'...
-- RC4(RC4 stream cipher) Proc_00171c70 --
(1)[40f584:20] Key 'spJ9\92\f7q\f2'... H=1.0000
(2)[40f5d8:188] Input '\e4R\fb\d8\f1'... H=0.9158
(3)[40f174:188] Output 'POST /write HTTP'... H=0.7818
Root Binary_002_0_0[40f174:188] 'POST /write HTTP/1.1\0d'...
[40f174:20] 'POST /write HTTP/1.1'
[40f188:168] '\0d\0aContent-Length: '...
[40f188:71] '\0d\0aContent-Length: 96\0d\0aAc'...
[40f1cf:97] '\0aD\00\00\00\00\00'...
[40f1cf:1] '\0a'
[40f1d0:4] 'D\00\00\00'
[40f1d4:92] '\00\00\00\00\01\00'...
[40f1d4:4] '\00\00\00\00'
[40f1d8:4] '\01\00\00\00'
[40f1dc:84] '\04V\0c7\f5c0\d8\06'...
[40f1dc:16] '\04V\0c7\f5c0\d8\06\18\2a\97'
[40f1ec:68] '\01\00\00\00\00\00'...
MD5
[40f1ec:16] '\01\00\00\00\00\00\00\004\00'
[40f1fc:52] '\96X\b5\e8\a0=\b1G\0cD\18\19'...
-- MD5(Message Digest) Proc_0f555e90 --
(1)[40f1ec:68] Message '\01\00\00\00'... H=0.8086
(2)[40f13c:16] Digest '\04V\0c7\f5c0' H=1.0000
-- XOR(Chained-XOR-Decrypt) Proc_001728a0 --
(1)[40f1fc:1] IV '\96' H=0.0000
(2)[40f1fc:52] Ciphertext '\96X\b5\e8\a0'... H=0.9018
(3)[104b000:52] Plaintext '\96\ce\ed]H\9d'... H=0.8777
Root Binary_002_0_0_0[40f13c:16] '\04V\0c7\f5c0\d8\06\18\2a\97'
Root Binary_002_0_0_1[104b000:52] '\96\ce\ed]H\9d\8c\f6'...
[104b000:1] '\96'
[104b001:16] '\ce\ed]H\9d\8c\f6KH\01\baJ\b9'
[104b011:35] 'ddos_url https://www.goo'...
MD5
[104b011:8] 'ddos_url'
[104b019:1] ' '
[104b01a:26] 'https://www.google.com.hk/'
-- MD5(Message Digest) Proc_0f555e90 --
(1)[104b011:35] Message 'ddos_url https:/'... H=0.8091
(2)[40f14c:16] Digest '\ce\ed]H\9d\8c' H=0.9688
Root Binary_002_0_0_1_0[40f14c:16] '\ce\ed]H\9d\8c\f6KH\01\baJ\b9'

Fig. 10. ZeuS message format

References

1. PIN - a dynamic binary instrumentation tool. http://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool
2. QEMU open source processor emulator. http://wiki.qemu.org/Main_Page

3. Caballero, J., Poosankam, P., Kreibich, C., Song, D.: Dispatcher: enabling active botnet infiltration using automatic protocol reverse-engineering. In: Proceedings of the 16th ACM Conference on Computer and Communications Security, pp. 621–634. ACM (2009)
4. Caballero, J., Yin, H., Liang, Z., Song, D.: Polyglot: automatic extraction of protocol message format using dynamic binary analysis. In: Proceedings of the 14th ACM Conference on Computer and Communications Security, pp. 317–329. ACM (2007)
5. Calvet, J., Fernandez, J.M., Marion, J.Y.: Aligot: cryptographic function identification in obfuscated binary programs. In: Proceedings of the 2012 ACM Conference on Computer and Communications Security, pp. 169–182. ACM (2012)
6. Cho, C.Y., Shin, E.C.R., Song, D., et al.: Inference and analysis of formal models of botnet command and control protocols. In: Proceedings of the 17th ACM Conference on Computer and Communications Security, pp. 426–439. ACM (2010)
7. Comparetti, P.M., Wondracek, G., Kruegel, C., Kirda, E.: Prospex: protocol specification extraction. In: 2009 30th IEEE Symposium on Security and Privacy, pp. 110–125. IEEE (2009)
8. Cui, W., Kannan, J., Wang, H.J.: Discoverer: automatic protocol reverse engineering from network traces. In: Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium, pp. 1–14 (2007)
9. Cui, W., Peinado, M., Chen, K., Wang, H.J., Irun-Briz, L.: Tupni: automatic reverse engineering of input formats. In: Proceedings of the 15th ACM Conference on Computer and Communications Security, pp. 391–402. ACM (2008)
10. Elisan, C.: The XOR bypass (2012). https://blog.damballa.com/archives/tag/malware-dropper
11. Gröbert, F.: Automatic identification of cryptographic primitives in software. Diploma thesis, Ruhr-University Bochum, Germany (2010)
12. Lee, C.P.: Framework for botnet emulation and analysis. ProQuest (2009)
13. Li, X., Wang, X., Chang, W.: CipherXRay: exposing cryptographic operations and transient secrets from monitored binary execution (2012)
14. Lin, Z., Jiang, X., Xu, D., Zhang, X.: Automatic protocol format reverse engineering through context-aware monitored execution. In: NDSS, vol. 8, pp. 1–15 (2008)
15. Lutz, N.: Towards revealing attackers intent by automatically decrypting network traffic. Master's thesis, ETH, Zürich, Switzerland, July 2008
16. Newsome, J., Song, D.: Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In: NDSS (2005)
17. Rossow, C., Dietrich, C.J.: ProVeX: detecting botnets with encrypted command and control channels. In: DIMVA (2013)
18. Wang, Y., Zhang, Z., Yao, D.D., Qu, B., Guo, L.: Inferring protocol state machine from network traces: a probabilistic approach. In: Lopez, J., Tsudik, G. (eds.) ACNS 2011. LNCS, vol. 6715, pp. 1–18. Springer, Heidelberg (2011)
19. Wang, Z., Jiang, X., Cui, W., Wang, X., Grace, M.: ReFormat: automatic reverse engineering of encrypted messages. In: Backes, M., Ning, P. (eds.) ESORICS 2009. LNCS, vol. 5789, pp. 200–215. Springer, Heidelberg (2009)
20. Wondracek, G., Comparetti, P.M., Kruegel, C., Kirda, E., Anna, S.S.S.: Automatic network protocol analysis. In: NDSS, vol. 8, pp. 1–14 (2008)
21. Zhao, R.: Lochsemu process emulator for windows x86. https://github.com/zhaoruoxu/lochsemu

22. Zhao, R., Gu, D., Li, J., Liu, H.: Detecting encryption functions via process emulation and IL-based program analysis. In: Chim, T.W., Yuen, T.H. (eds.) ICICS 2012. LNCS, vol. 7618, pp. 252–263. Springer, Heidelberg (2012)
23. Zhao, R., Gu, D., Li, J., Yu, R.: Detection and analysis of cryptographic data inside software. In: Lai, X., Zhou, J., Li, H. (eds.) ISC 2011. LNCS, vol. 7001, pp. 182–196. Springer, Heidelberg (2011)