An Analysis of Gameover Zeus Network Traffic

GIAC (GCIA) Gold Certification

Author: Daryl Ashley, Ashley@utexas.edu
Advisor: Dominicus Adriyanto Hindarto

Accepted: January 24 2015

Abstract

Malware is evolving to use encryption techniques to obfuscate network communication to evade detection. This paper analyzes anomalies within network traffic generated by Gameover Zeus. The anomalies result from the encryption methods used to obfuscate network communications. However, even though the anomalies can be seen when manually inspecting the network packets, the obfuscation techniques pose difficulties when attempting to use signature based Intrusion Detection Systems (IDS) for detection. While the anomalies may not be useful for constructing IDS signatures, they may be useful in constructing custom detection algorithms.

1. Introduction

In September of 2011, a peer-to-peer variant of Zeus emerged on the internet (Symantec, 2014). This version of Zeus, also known as Gameover or P2P Zeus, is not susceptible to traditional takedown methods because the command and control infrastructure is no longer centralized. Detection of this variant is also made more difficult because communication between the peers is encrypted (Andriesse & Bos, 2014). Although the botnet has been significantly disrupted by a takedown effort (Symantec, 2014), analysis of the malware can provide useful insights into the effectiveness of signature based intrusion detection systems.

The protocol used for communication has been described in detail in research papers written by Andriesse & Bos (2014) and Cert Polska (2013). This paper uses the information from the research papers to decrypt and analyze the information in two separate packet captures. In the first packet capture, the Zeus infected hosts use a simple XOR based algorithm for encrypting its traffic. In the second packet capture, the RC4 algorithm is used for encryption. The two packet captures have interesting anomalies that differ due to the encryption algorithm that was used.

The IP addresses shown in the figures in this paper have been converted to private IP addresses. The packet captures are available on request from the author.

2. Zeus Communication Protocol

2.1. Overview

The protocol used by Gameover Zeus is described in detail in a research paper published by Andriesse and Bos (2014). The protocol includes mechanisms for exchanging binary and configuration updates, requesting peer lists, and requesting the IP address of special members of the botnet referred to as "proxy bots" (Andriesse & Bos, 2014). The following sections outline a portion of the research that was used when analyzing the packet captures.

Daryl Ashley, ashley@utexas.edu

2.2. Network Communication

Each infected host uses a unique UDP port for communication (Cert Polska, 2013). For hosts infected with a version of Zeus prior to June, 2013, the port was selected from the range 10,000 to 30,000. For hosts infected after June, 2013, the range was between 1024 and 10000 (Andriesse & Bos, 2014). Figure 1 shows the output of a packet capture that was generated using the command **tcpdump –nr zeus.pcap proto 17 and host 10.1.1.1**. The network traffic of a host infected with Gameover Zeus was captured to a file named zeus.pcap. The host that was infected with Gameover Zeus has an IP address of 192.168.1.1. Since the infected host sends UDP packets to a number of peers, the host filter was used to display traffic that was generated to a single peer, the peer at IP address 10.1.1.1. The filtered output makes it easier to focus on the network traffic generated between the two peers.

```
16:38:40.382191 IP 192.168.1.1.26609 > 10.1.1.1.10619: UDP, length 122
16:38:40.433485 IP 10.1.1.1.10619 > 192.168.1.1.26609: UDP, length 502
16:38:40.433883 IP 192.168.1.1.26609 > 10.1.1.1.10619: UDP, length 130
16:38:40.485140 IP 10.1.1.1.10619 > 192.168.1.1.26609: UDP, length 184
16:38:40.486484 IP 10.1.1.1.10619 > 192.168.1.1.26609: UDP, length 506
16:38:40.487234 IP 10.1.1.1.10619 > 192.168.1.1.26609: UDP, length 491
16:38:40.488214 IP 10.1.1.1.10619 > 192.168.1.1.26609: UDP, length 431
16:38:40.488694 IP 10.1.1.1.10619 > 192.168.1.1.26609: UDP, length 429
```

Figure 1: tcpdump Output of Zeus UDP Packets

The packet capture shows two hosts communicating with each other using the UDP protocol. But, there is nothing that appears to be malicious about this network traffic. The host with IP address 192.168.1.1 is using port 26609 for network communication. Since this port is within the range 10000 to 30000, the infected host is running a version of Zeus released prior to June, 2013. When the infected host sends a packet, the source port is set to 26609. When the host at IP address 10.1.1.1 receives the UDP packet, it will send replies to port 26609.

2.3. Message Header

Each UDP packet sent by a host infected with Zeus contains a Zeus message as its UDP payload. The Zeus message can be broken up into two parts, a 44 byte message header followed by a message payload. The message payload will vary in length

Daryl Ashley, ashley@utexas.edu

depending on the type of message being sent (Andriesse & Bos, 2014). Figure 2 shows the packet layout of a Zeus message, including the IP and UDP header sections.

<table>
<tr><td>ip header</td><td colspan="8">ip payload</td></tr>
<tr><td></td><td>udp header</td><td colspan="7">udp payload</td></tr>
<tr><td></td><td></td><td colspan="6">Zeus message header</td><td>Zeus message payload</td></tr>
<tr><td></td><td></td><td>rnd</td><td>ttl</td><td>lop</td><td>type</td><td>session ID</td><td>source ID</td><td></td></tr>
</table>

Figure 2: Zeus Packet Layout

Figure 3 summarizes the fields that are present in the Zeus message header, as well as their position within the header and the length of each field. One field that is particularly interesting is the lop field. Zeus appends a number of random bytes to each message that is sent. The peer that receives the message discards the randomly generated bytes after it decrypts the message. The lop field contains the number of random bytes that have been appended to the message. Since a random number of bytes are appended to each message, the length of each UDP packet sent between two infected hosts will usually differ. The variable length packets may help infected hosts evade detection by intrusion detection systems (Andriesse & Bos, 2014). Therefore, Zeus may use variable length packets as well as encryption to evade detection.

field	length	description
rnd	1	randomly generated byte
TTL	1	time to live
LOP	1	length of padding
type	1	message type
session ID	20	randomly generated to tag session
source ID	20	used as unique identifier of infected host

Figure 3: Zeus Protocol Header

Daryl Ashley, ashley@utexas.edu

The other header fields that are interesting are the type and source ID fields. The type field will be discussed in the following section. The source ID field is a unique identifier of the host sending the message. In versions of Zeus after June, 2013, RC4 is used to encrypt messages. The source ID field is used as the RC4 key when encrypting replies to the sending host (Andriesse & Bos, 2014).

2.4. Message Types

There are a number of different message types that are used by Gameover Zeus (Andriesse & Bos, 2014). Figure 4 provides a summary of some of the message types. The length of the payload of each message type is of interest because it can be used to calculate the expected length of the message. The message length should be equal to 44 bytes (header) + payload length + lop.

type	payload length	description
0x0	0 or 12	version request
0x1	22	version reply
0x2	28	peer list request
0x3	450	peer list reply
0x6	304	proxy reply
0x32	304	proxy announce

Figure 4: Zeus Message Types

3. Packet Analysis

3.1. tcpdump

The packet captures were created using tcpdump. The initial analysis of the encrypted packets was also performed with tcpdump. The –X flag can be used to display the packet in hexadecimal format, along with an ASCII conversion on the right hand side of the output. The output can be filtered using the host, port, and proto keywords ("Manpage of TCPDUMP", 2014). Use of these keywords to filter the output can significantly reduce the amount of data that needs to be analyzed.

3.2. Automated Analysis of Packet Captures

Several python scripts are included in the appendix of this paper. The scripts were used to automate the decryption and decoding of the UDP packets used by Zeus for communication. The dpkt Python module can be used to read a packet capture that was produced by tcpdump (Oberheide, 2008). For example, the code snippet shown in figure 5 will open a packet capture file and iterate through each of the packets, printing the source port of any UDP packets in the packet capture.

```
import dpkt
filename = "infected.pcap"
def main():
    for ts, pkt in dpkt.pcap.Reader(open(filename, 'r')):
        eth = dpkt.ethernet.Ethernet(pkt)
        if eth.type!=dpkt.ethernet.ETH_TYPE_IP:
            continue
        ip = eth.data
        if ip.p == dpkt.ip.IP_PROTO_UDP:
            udp = ip.data
            print "UDP source port", udp.sport

if __name__=="__main__":
    main()
```

Figure 5: Using dpkt to Parse Packet Capture

3.3. XOR Decryption

Prior to June of 2013, Gameover Zeus used a "rolling XOR" algorithm to encrypt its messages (Andriesse & Bos, 2014). An example of the rolling XOR algorithm is as follows. Suppose the message payload is the sequence of bytes "0x11 0x2e 0x54 0x9d". The first byte is left as is in the cipher text. The second byte is encrypted by XORing the second byte of the original message with the first byte of the cipher text: 0x11 XOR 0x2e = 0x3f. This is the second byte of the cipher text. The third byte is encrypted by XORing the unencrypted third byte of the original message with the second byte of the cipher text: 0x3f XOR 0x54 = 0x6B. Finally, 0x6B, the third byte of the cipher text, is XORed with the last byte of the original message. The cipher text is "0x11 0x3f 0x6B 0xF6".

Decryption is the opposite of encryption. First, the last byte of the cipher text is XORed with the preceding byte of the cipher text: 0xF6 XOR 0x6B = 0x9d. This recovers the last byte of the original message. This process is repeated for all the remaining bytes in the cipher text except the first byte, which was not encrypted.

There are a couple of interesting observations about this algorithm. First, in order to determine the value of a specific byte in the original message, it is not necessary to decrypt the entire message. For example, to determine the original value of the second byte, XOR it with the preceding byte: 0x3F XOR 0x11 = 0x2E. It is not necessary to decrypt the third and fourth bytes before jumping to this step.

The second observation is that any value XORed with 0 is equal to the value itself. For example 0x4 XOR 0x0 = 0x4. Suppose that the rolling XOR algorithm is used to encrypt the message "0x11 0x2e 0x00 0x00 0x00 0x00". It can be shown that the corresponding cipher text is "0x11 0x3F 0x3F 0x3F 0x3F 0x3F". Note that the encrypted byte at position 2 is repeated each time it is XORed with 0x00. This observation will be used when analyzing Zeus messages encrypted using the rolling XOR algorithm.

The python snippet shown below was used to decrypt messages that were encrypted using the rolling XOR algorithm.

```
def xordecrypt(payload):
    decrypted = []
    decrypted.append(ord(payload[0]))
    for i in range(1, len(payload)):
        decrypted.append(ord(payload[i]) ^ ord(payload[i-1]))
    return decrypted
```

Figure 6: Python Subroutine to Decrypt Rolling XOR

Daryl Ashley, ashley@utexas.edu

3.4. RC4 Decryption

After June of 2013, Zeus started using RC4 to encrypt its traffic (Andriesse & Bos, 2014). RC4 is a widely used software stream cipher. The cipher generates a pseudo random sequence of bytes that is XORed with the message to produce a cipher text. The same pseudo random sequence of bytes is needed to decrypt the cipher text. The cipher text is XORed with the pseudo random sequence of bytes to recover the original message (Paul, 2012).

A software based stream cipher has two components. The first is a key scheduling component that uses a secret key to initialize the internal state of the RC4 instance. Once initialized, a pseudo-random generation algorithm is used to generate the sequence of bytes that is used for encryption and decryption (Paul, 2012). If a different key is used to initialize the RC4 instance, a different stream of bytes will be generated, and decryption of the cipher text will not succeed.

Figure 7 shows RC4 encryption of the plain text "john smith" when the RC4 instance is initialized with the key "darylashley". The figure shows the byte stream produced by the pseudo-random generation algorithm. The figure also shows the hexadecimal representation of "john smith". Each byte of the plain text is XORed with the corresponding byte in the byte stream. Figure 8 shows the RC4 encryption of the plain text "abcdesmith" using the same key.

byte stream		0x55	0x5c	0xa6	0xf9	0x1c	0x55	0xaa	0x7c	0x4d	0xee
plain text	⊕	0x6a	0x6f	0x68	0x6e	0x20	0x73	0x6d	0x69	0x74	0x68
cipher text		0x3f	0x33	0xce	0x97	0x3c	0x26	0xc7	0x15	0x39	0x86

Figure 7: RC4 Encryption of "john smith"

Daryl Ashley, ashley@utexas.edu

byte stream		0x55	0x5c	0xa6	0xf9	0x1c	0x55	0xaa	0x7c	0x4d	0xee
plain text	⊕	0x61	0x62	0x63	0x64	0x65	0x73	0x6d	0x69	0x74	0x68
cipher text		0x34	0x3e	0xc5	0x9d	0x79	0x26	0xc7	0x15	0x39	0x86

Figure 8: RC4 Encryption of "acbdesmith"

If the same key is used to encrypt multiple messages, the RC4 algorithm is susceptible to cryptographic attacks. Even though each message has been encrypted, the last 5 bytes of each cipher text are identical because the 5 bytes at offset 6 of each message is "smith". Since each RC4 instance was in an identical state when encrypting the messages, the same pseudo random byte sequence was used to encrypt each of the messages at this offset. Although it may not be possible to recover the original messages from the above cipher text, an attacker would know that the two messages contained identical data in the last 5 bytes of each message. This observation will be used when analyzing Zeus packets encrypted using the RC4 algorithm.

The Python Cryptography Toolkit is a python package that contains various cryptographic functions. It is available at https://www.dlitz.net/software/pycrypto. The package provides an ARC4 module that can be used to perform RC4 encryption of a message. The new() function can be passed a key parameter that can be used to initialize the internal state of the RC4 instance. The encrypt and decrypt functions can be used to encrypt and decrypt messages after the RC4 instance has been created and initialized (Litzenberger, 2012). Figure 9 shows a code snippet that uses the package to encrypt the plain text "abcdesmith" after initializing the RC4 instance with the key "darylashley".

Daryl Ashley, ashley@utexas.edu

```
from Crypto.Cipher import ARC4 as rc4

# Secret key used to initialize RC4 state
KEY = "darylashley"

# Messages to encrypt
msg = "abcdesmith"

# Create RC4 Instance
r = rc4.new(KEY)
# Encrypt message 1
cipher = r.encrypt(msg)

# Display the message
print " ".join(hex(ord(n)) for n in cipher)
```

Figure 9: RC4 Python Snippet

4. XOR Packet Capture

Figure 10 shows a UDP packet for a host infected with a version of Zeus that uses the rolling XOR algorithm to encrypt its traffic. The UDP ports used to communicate are between 10000 and 30000, so this is a version of Zeus prior to June 2013.

The –X tcpdump flag was used to generate a hexadecimal output of the packet payload. Based on the IP header length field, the length of the IP header is 20 bytes. Since the protocol field is set to 0x11, this is a UDP packet. So, there will also be a UDP header which is 8 bytes in length. The UDP payload should start at offset 0x1C of the packet. The first four bytes at this offset are circled in figure 10. These represent the encrypted rnd, ttl, lop, and type field of the Zeus header. The length of the UDP payload is 378 bytes, and is also circled in the figure.

```
14:05:09.768633 IP 10.1.1.1.16503 > 192.168.1.1.17973: UDP, length 378
        0x0000:  4500 0196 fce3 0000 7311 9fc7 0a01 0101  E.......s......0
        0x0010:  c0a8 0101 4077 4635 0182 2792 94c6 d8de  ....@wF5..'.....
        0x0020:  e2cf 4e5b 6c4a ecf2 7ff4 1e2d 4c14 1434  ..N[lJ.....-L..4
        0x0030:  d1d1 815d da02 500a ec45 e6be 7cc4 ad8d  ...]..P..E..|...
        0x0040:  82b5 1ae4 f742 190a 0a0a 0a0a 894c 4c1e  .....B.......LL.
        0x0050:  9424 10fd c98b 9605 57f2 afd0 a410 d0fe  .$......W.......
        0x0060:  4c16 70e1 2352 5252 5252 5252 5252 5252  L.p.#RRRRRRRRRRR
        0x0070:  5252 5252 5252 5252 60c9 f8f3 46ec 8a9d  RRRRRRRR`...F...
        0x0080:  5a79 a78c 3c76 6460 342f 15dc 0309 6c5b  Zy..<vd`4/....l[
        0x0090:  1d28 8797 d29c 6138 0179 3cc6 d7b6 40a1  .(....a8.y<...@.
        0x00a0:  e9b2 1f62 11fe 0ea4 05e8 0325 e4de 1149  ...b.......%...I
        0x00b0:  d842 94cc 316e 6f76 c453 7859 c65b 8898  .B..1nov.SxY.[..
        0x00c0:  ec0f f907 a4e9 a85f 5880 7aed a3eb cf3a  ......._X.z....:
        0x00d0:  cc4c 5f91 4922 4bf7 28e7 1e0d 9d02 b332  .L_.I"K.(......2
        0x00e0:  9676 11a4 c00a 349b 6d22 4020 cedf 56f0  .v....4.m"@...V.
        0x00f0:  0b50 fd25 3cf6 55dc 620b a0c4 e331 3eff  .P.%<.U.b....1>.
        0x0100:  6912 f90e ed32 697d 61fe 6586 0714 dcab  i....2i}a.e.....
        0x0110:  ba76 9c8a 54a1 8f2f 3c61 c7fa 11e1 323e  .v..T../<a....2>
        0x0120:  2443 c170 121a 07cd 07af 99bb dddf 4adf  $C.p..........J.
        0x0130:  9035 ce2b 505c e109 f675 e4ba 7710 11ed  .5.+P\...u..w...
        0x0140:  da1b 317f 85ef a097 6b02 ae0f 4dc6 5207  ..1.....k...M.R.
        0x0150:  2311 c525 a117 d9f6 5002 b79b b3c0 068c  #..%....P.......
        0x0160:  e73c f7f5 44ff 8ac3 b803 aa2b 8066 5ea8  .<..D......+.f^.
        0x0170:  3809 4bee 4c51 5efc d5e4 1e87 8230 a976  8.K.LQ^......0.v
        0x0180:  2d77 9acc 3686 8f14 b61e 69a7 2d2b 4603  -w..6.....i.-+F.
        0x0190:  2b10 0b81 8f23                           +....#
~
~
```

Figure 10: First 4 bytes of Message

The lop field is located at offset 0x1e of the packet, and the type field is located at offset 0x1f of the packet. The fields can be decrypted by XORing them with the preceding byte in the UDP payload. The lop = 0xd8 XOR 0xc6 = 0x1e. This means that the number of random bytes appended to this message was 30 bytes. The type = 0xde XOR 0xd8 = 0x6. Based on the summary of message types shown in figure 3, this is a proxy reply packet and should have a payload of length 304 bytes. The expected length of the packet is 44 (header bytes) + 304 (payload bytes) + 30 (lop) = 378 bytes. This matches the payload length displayed by tcpdump.

This approach to identifying a potential Zeus UDP packet is fairly straightforward. However, creating a rule to detect this type of packet for a signature based IDS, such as Snort, may not be possible. Instead, this approach could possibly be implemented as a dynamic preprocessor in Snort because a dynamic preprocessor can be used to perform more complex analysis of the packets inspected by Snort (Ashley, 2008).

Daryl Ashley, ashley@utexas.edu

However, this approach is more time consuming than writing a signature because custom code must be written.

Figure 11 shows the first 92 bytes of the Zeus payload after it has been decrypted and decoded. The python scripts used to decrypt and decode the packet are included in the appendix. The information used to decode the packet is based on the proxy struct describe in (Andriesse & Bos, 2014).

```
rnd:          148
ttl:          82
lop:          30
type:         6
session id:   0x3c 0x2d 0x81 0x15 0x37 0x26 0xa6 0x1e 0x8d 0x8b 0xea 0x33 0x61 0x58 0x0 0x20 0xe5 0x0 0x50 0xdc
source id:    0x87 0xd8 0x52 0x5a 0xe6 0xa9 0xa3 0x58 0xc2 0xb8 0x69 0x20 0xf 0x37 0xaf 0xfe 0x13 0xb5 0x5b 0x13
ip type:      0x0 0x0 0x0 0x0
peer id:      0x83 0xc5 0x0 0x52 0x8a 0xb0 0x34 0xed 0x34 0x42 0x1d 0x93 0x52 0xa5 0x5d 0x7f 0x74 0xb4 0xc0 0x2e
ipv4 address: 10.2.2.2
ipv4 port:    29122
ipv6 address: 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0
ipv6 port:    0x0 0x0
```

Figure 11: Decrypted Packet Contents

The portion of the decoded packet that is of interest is the ipv6 address and ipv6 port. The ipv6 address contains a sequence of sixteen 0x00 values, and the ipv6 port contains a sequence of two 0x00 values. This sequence of bytes produces the anomaly shown in the figure 12. The eighteen bytes in the UDP payload starting at offset 0x66 are identical to the byte located at offset 0x65 of the packet. The reason for the anomaly is described in section 3.3 of this paper. Code to check for this anomaly could potentially be added as a sanity check when writing the dynamic preprocessor.

Daryl Ashley, ashley@utexas.edu

```
14:05:09.768633 IP 10.1.1.1.16503 > 192.168.1.1.17973: UDP, length 378
        0x0000:  4500 0196 fce3 0000 7311 9fc7 0a01 0101  E.......s...l..0
        0x0010:  c0a8 0101 4077 4635 0182 2792 94c6 d8de  ....@wF5..'.....
        0x0020:  e2cf 4e5b 6c4a ecf2 7ff4 1e2d 4c14 1434  ..N[lJ.....-L..4
        0x0030:  d1d1 815d da02 500a ec45 e6be 7cc4 ad8d  ...]..P..E..|...
        0x0040:  82b5 1ae4 f742 190a 0a0a 0a0a 894c 4c1e  .....B.......LL.
        0x0050:  9424 10fd c98b 9605 57f2 afd0 a410 d0fe  .$......W.......
        0x0060:  4c16 70e1 2352 5252 5252 5252 5252 5252  L.p.#RRRRRRRRRR
        0x0070:  5252 5252 5252 5252 60c9 f8f3 46ec 8a9d  RRRRRRRR`...F...
        0x0080:  5a79 a78c 3c76 6460 342f 15dc 0309 6c5b  Zy..<vd`4/....l[
        0x0090:  1d28 8797 d29c 6138 0179 3cc6 d7b6 40a1  .(....a8.y<...@.
        0x00a0:  e9b2 1f62 11fe 0ea4 05e8 0325 e4de 1149  ...b.......%...I
        0x00b0:  d842 94cc 316e 6f76 c453 7859 c65b 8898  .B..1nov.SxY.[..
        0x00c0:  ec0f f907 a4e9 a85f 5880 7aed a3eb cf3a  ......._X.z....:
        0x00d0:  cc4c 5f91 4922 4bf7 28e7 1e0d 9d02 b332  .L_.I"K.(......2
        0x00e0:  9676 11a4 c00a 349b 6d22 4020 cedf 56f0  .v....4.m"@...V.
        0x00f0:  0b50 fd25 3cf6 55dc 620b a0c4 e331 3eff  .P.%<.U.b....1>.
        0x0100:  6912 f90e ed32 697d 61fe 6586 0714 dcab  i....2i}a.e.....
        0x0110:  ba76 9c8a 54a1 8f2f 3c61 c7fa 11e1 323e  .v..T../<a....2>
        0x0120:  2443 c170 121a 07cd 07af 99bb dddf 4adf  $C.p..........J.
        0x0130:  9035 ce2b 505c e109 f675 e4ba 7710 11ed  .5.+P\...u..w...
        0x0140:  da1b 317f 85ef a097 6b02 ae0f 4dc6 5207  ..1.....k...M.R.
        0x0150:  2311 c525 a117 d9f6 5002 b79b b3c0 068c  #..%....P.......
        0x0160:  e73c f7f5 44ff 8ac3 b803 aa2b 8066 5ea8  .<..D......+.f^.
        0x0170:  3809 4bee 4c51 5efc d5e4 1e87 8230 a976  8.K.LQ^......0.v
        0x0180:  2d77 9acc 3686 8f14 b61e 69a7 2d2b 4603  -w..6.....i.-+F.
        0x0190:  2b10 0b81 8f23                           +....#
~
~
```

Figure 12: 18 Consecutive Identical Bytes

5. RC4 Packet Capture

For versions of Zeus after June 2013, the encryption algorithm was changed to RC4. The key used to initialize the RC4 state is the source ID of the recipient host (Andriesse & Bos, 2014). Since the source ID of the sending host is included in the message header, the receiving host will have the sending host's RC4 key, and will be able to encrypt the reply packet.

Since the packet is encrypted using RC4, the key used to perform the encryption is required to decrypt the packet (Paul, 2012). This is an improvement over the rolling XOR algorithm because no key was required to decrypt packets encrypted using the rolling XOR algorithm. Because the source ID of the receiving host is required to decrypt a packet, it is no longer possible to decrypt the lop and type fields to determine if the UDP payload length matches the predicted length of a Zeus message.

Daryl Ashley, ashley@utexas.edu

Figure 13 shows a decrypted proxy announce packet. The RC4 key was obtained by reverse engineering a binary used to infect the virtual host that produced the network traffic in the RC4 packet capture. Note that the ipv6 address and port each contain a sequence of 0x00 values as was the case for the proxy reply shown in the XOR section.

```
rnd:           89
ttl:           0
lop:           72
type:          50
session id:    0xa6 0xf4 0xba 0x78 0x31 0xd 0xa1 0x29 0x74 0x2a 0x61 0xc3 0x40 0x9b 0x34 0xb2 0xa2 0xf9 0xa6 0x12
source id:     0x98 0x51 0xee 0x64 0xc0 0xf3 0x5f 0x48 0x5a 0x1 0xd7 0x11 0x9c 0xc6 0x61 0x6b 0xe6 0x9f 0xdc 0xcc
ip type:       0x0 0x0 0x0 0x0
peer id:       0x4e 0xce 0x9f 0x9a 0xa4 0xc1 0xdf 0x7e 0x86 0xa 0xa4 0x79 0xb5 0xe4 0xa2 0x0 0x1d 0xf2 0x36 0x39
ipv4 address:  95.104.97.205
ipv4 port:     6341
ipv6 address:  0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0
ipv6 port:     0x0 0x0
```

Figure 13: Decrypted Proxy Reply Packet

Figure 14 shows the encrypted packet as displayed via tcpdump. Note that the 18 bytes at offset 0x66 are no longer identical. This is a result of the strengthened encryption that this version of Zeus is using. So, the two methods outlined in section 4 of this paper are no longer able to detect Zeus traffic.

Daryl Ashley, ashley@utexas.edu

```
08:54:53.122241 IP 192.168.1.1.9358 > 10.1.1.1.8029: UDP, length 420
        0x0000:  4500 01c0 060e 0000 8011 887f c0a8 0101  E.............S..
        0x0010:  0a01 0101 248e 1f5d 01ac c1f9 36c7 0915  Y...$..]....6...
        0x0020:  4709 6b6f d3f5 2545 a254 8654 37d4 37ed  G.ko..%E.T.T7.7.
        0x0030:  78a1 84d4 a301 6ef3 0a39 8831 ab2b cc86  x.....n..9.1.+..
        0x0040:  e573 bedc 3b02 a8b5 2a37 2e8a ea68 4f56  .s..;...*7...hOV
        0x0050:  d1c7 689f 685c f52c 0133 cd7f 38ae a40c  ..h.h\.,.3..8...
        0x0060:  3ba9 97a3 15dd 9ef2 8385 4947 90a8 09a6  ;.........IG....
        0x0070:  0d4a 101b 33c9 d60c 983e 10ef 2564 cb90  .J..3.....>..%d..
        0x0080:  ceca d56e 57bb e232 b216 adb7 8342 b5a8  ...nW..2.....B..
        0x0090:  8cd8 3281 d192 d459 4998 ca5c d078 4830  ..2....YI..\.xH0
        0x00a0:  a72b 6823 24b3 70a4 c0f0 ef6f 63b5 813b  .+h#$.p....oc..;
        0x00b0:  c562 cbba 7db3 c491 42e8 0bb2 1b1c 6e26  .b..}...B.....n&
        0x00c0:  0cb9 f148 ca09 6ff0 883f 3418 41aa cf26  ...H..o..?4.A..&
        0x00d0:  2f8e 3eb4 dde4 0a3b f2cc d4fb f28d c584  /.>....;........
        0x00e0:  da06 94f3 f926 ecb4 3076 0145 3b1f 3f18  .....&..0v.E;.?.
        0x00f0:  3de0 ee9f 3ec6 b08b 622c c64e 7e2a 2a33  =...>...b,.N~**3
        0x0100:  2b9a f4f7 1560 7a14 91fb c783 891b 9728  +....`z.........(
        0x0110:  f9df 0be1 bf0f e110 4f78 9ab9 41c3 15ec  .........Ox..A...
        0x0120:  025f e767 8525 f990 8aaa 6ce6 5b4a 2b69  ._.g.%....l.[J+i
        0x0130:  d7be 6986 0a74 0beb a3bb e721 9574 2708  ..i..t.....!.t'.
        0x0140:  6ede e91b 0150 75d9 72da df46 70ac e7f6  n....Pu.r..Fp...
        0x0150:  ed34 ccb3 86c6 879a 594b 2827 4ba1 3f33  .4......YK('K.?3
        0x0160:  00de eb8c e25b f596 038b 1690 81f1 3c19  .....[.........<.
        0x0170:  7187 b9c7 5327 507d 3f41 58ce 3248 552b  q...S'P}?AX.2HU+
        0x0180:  7940 c3fe 870f 764f c520 2139 a038 b8c5  y@....vO..!9.8..
        0x0190:  7ff1 9647 9b49 f888 d91a 63ad 4979 f1e1  ...G.I....c.Iy..
        0x01a0:  6c79 2e14 8518 98ad f7c7 66aa 20c5 9758  ly.........f....X
        0x01b0:  d996 7186 d7ca 3952 b42a 5c0a 36e0 6a22  ..q...9R.*\.6.j"
~
~
```

Figure 14: Proxy Announcement Encrypted Using RC4

In order to find an anomaly in the network traffic, several packets transmitted between the same hosts must be inspected. Recall that the source ID of the sending host will be included at a specific location of the Zeus message header. Also recall that the RC4 key used to encrypt the message is the source ID of the recipient of the message. If the same source ID is reused to initialize the RC4 state prior to encryption of each packet, the sender's encrypted source ID will be identical. Figure 15 shows this anomaly in the packet capture.

Although it may not be possible to recover the unencrypted source IDs of the two infected hosts from this packet capture, this anomaly may be useful in identifying potential Zeus messages. For example, in the packet capture shown in Figure 15, the

Daryl Ashley, ashley@utexas.edu

lengths of the four packets are different, and the packet contents are encrypted. But, the 20 bytes within the packet sent by IP address 192.168.1.1 highlighted in red are identical. Similarly, the 20 bytes sent by IP address 10.1.1.1 highlighted in blue are identical. This does not definitively prove that the two hosts are infected with Zeus. However, this may be useful for identifying hosts that are good candidates for further investigation.

```
08:38:16.815674 IP 192.168.1.1.9358 > 10.1.1.1.8029: UDP, length 239
        0x0000:  4500 010b 0543 0000 8011 89ff c0a8 0101  E....C.......S..
        0x0010:  0a01 0101 248e 1f5d 00f7 88aa 43ff e625  Y...$..]....C..%
        0x0020:  323a 76b6 031a 52cc 6b31 bf21 6e15 a74f  2:v...R.k1.!n..O
        0x0030:  2345 2962 a301 6ef3 0a39 8831 ab2b cc86  #E)b..n..9.1.+..
        0x0040:  e573 bedc 3b02 a8b5 ebe4 bebb 5a4e e447  .s..;.......ZN.G
        ...

08:38:16.980156 IP 10.1.1.1.8029 > 192.168.1.1.9358: UDP, length 508
        0x0000:  4500 0218 ead8 0000 6911 ba5c 0a01 0101  E.......i..\Y...
        0x0010:  c008 0101 1f5d 248e 0204 de6c 9d6b 9dcc  .S...]$....l.k..
        0x0020:  604b 2aaf dc7a 055f 4d83 a671 71f1 cbd3  `K*..z._M..qq...
        0x0030:  043c 2312 4967 9882 8058 2738 b9bc f730  .<#.Ig...X'8...0
        0x0040:  f129 fc3d af68 998c 1526 e220 1e51 5cf7  .).=.h...&...Q\.
        ...

08:50:03.215328 IP 192.168.1.1.9358 > 10.1.1.1.8029: UDP, length 164
        0x0000:  4500 00c0 05d2 0000 8011 89bb 8053 1e88  E............S..
        0x0010:  0a01 0101 248e 1f5d 00ac a1e0 c0a8 0101  Y...$..]....P.9'
        0x0020:  211d bd8c 34cd ad0e d9a4 bbd5 ac72 f22f  !..<4........r./
        0x0030:  fcf9 bf5d a301 6ef3 0a39 8831 ab2b cc86  ...]..n..9.1.+..
        0x0040:  e573 bedc 3b02 a8b5 783b ac9a 2101 ea57  .s..;...x;..!..W
        ...

08:50:03.641173 IP 10.1.1.1.8029 > 192.168.1.1.9358: UDP, length 68
        0x0000:  4500 0060 d0de 0000 6911 d60e 0a01 0101  E..`....i...Y...
        0x0010:  c008 0101 1f5d 248e 004c b772 8e4c 91ce  .S...]$..L.r.L..
        0x0020:  736c e125 6bad fa9d ff16 a285 b396 9eb3  sl.%k...........
        0x0030:  db80 b52d 4967 9882 8058 2738 b9bc f730  ..-Ig...X'8...0
        0x0040:  f129 fc3d af68 998c 660e 1b34 619c 57c6  .).=.h..f..4a.W.
        ...
```

Figure 15: Encrypted Source Identifiers

A Snort preprocessor may be used to detect this type of traffic as well. However, detection has been made more difficult because the information needed to find a potential Zeus packet is no longer available in a single UDP packet. Instead, the preprocessor would need to maintain enough information for each UDP packet received so that future

Daryl Ashley, ashley@utexas.edu

packets could be analyzed for matching encrypted source IDs. This may not be practical on a network that generates a large amount of traffic.

6. Conclusion

It can be argued that the encryption methods used by Gameover Zeus are a weakness that can be exploited by security analysts. For example, the use of the rolling XOR algorithm appears to violate several ideas that are central to the idea of modern cryptography.

Modern cryptography considers the notion of "security through obscurity" to be a bad idea. History has shown that this approach has failed many times (Klein, 2014). This paper shows that reverse engineering efforts were useful in identifying some weaknesses that can be leveraged to help detect the malware. However, this is not an optimal solution. For example, suppose 1000 new malware variants are written, and each uses a custom encryption algorithm that has some sort of weakness. The task of reverse engineering all of the executables and writing 1000 dynamic preprocessors does not seem practical.

Another idea of modern cryptography is the development of encryption algorithms that are computationally expensive to attack (Goldreich, 2001). For example, suppose an attacker has access to encrypted ecommerce data. The attacker may have many months and a large number of computers to try to extract information from the encrypted data. Modern cryptographic algorithms attempt to thwart this type of attack.

The rolling XOR algorithm used by Zeus is trivial to decrypt once the algorithm is known. This custom algorithm would not be considered an acceptable form of encryption from the standpoint of modern cryptography. So, why does this algorithm pose problems for signature based intrusion detection systems? The answer may be that the task of encryption and evasion are significantly different. An intrusion detection system does not have many months to decrypt the network packets that it analyzes. If the goal of Zeus's encryption is simply to evade detection, it may not need to use an encryption algorithm that will protect data against a brute force attack that will last several months and will be run on a number of computers. It simply needs to evade detection from a device that is potentially responsible for analyzing gigabits of data each second. Taken in this context, the weakness in Zeus's encryption may not be as glaring after all.

Daryl Ashley, ashley@utexas.edu

References

Andriesse D & Bos H. (2014). *An Analysis of the Zeus Peer-To-Peer Protocol*. Retrieved from: http://www.few.vu.nl/~da.andriesse/papers/zeus-tech-report-2013.pdf

Ashley, D. (2008). *Developing a Snort Dynamic Preprocessor*. Informally published manuscript, Retrieved from http://www.sans.org/reading-room/whitepapers/tools/developing-snort-dynamic-preprocessor-32874

Cert Polska. (2013). *Technical Report Zeus-P2P monitoring and analysis*. Retrieved from http://www.cert.pl/PDF/2013-06-p2p-rap_en.pdf

Goldreich, O. (2001). *Foundations of cryptology: Vol. 1*. Cambridge: Cambridge University Press.

Klein, P. N. (2014). *A cryptography primer: Secrets and promises*. New York: Cambridge University Press.

Litzenberger, Dwayne. (2012 May 24). *Crypto.Cipher.ARC4*. Retrieved from: https://www.dlitz.net/software/pycrypto/api/current/Crypto.Cipher.ARC4-module.html

Manpage of TCPDUMP. (2014, July 11). Retrieved November 25, 2014, from http://www.tcpdump.org/manpages/tcpdump.1.html

Oberheide, Jon. (2008 October 15). *dpkt Tutorial #2: Parsing a PCAP File*. Retrieved from: https://jon.oberheide.org/blog/2008/10/15/dpkt-tutorial-2-parsing-a-pcap-file/

Paul, G., & Maitra, S. (2012). *RC4 stream cipher and its variants*. Boca Raton: Taylor & Francis.

Symantec. (2014, June 2). *International Takedown Wounds Gameover Zeus Cybercrime Network | Symantec Connect*. Retrieved from http://www.symantec.com/connect/blogs/international-takedown-wounds-gameover-zeus-cybercrime-network

Appendix 1: ZeusHost Python Library

```
import struct
from Crypto.Cipher import ARC4 as rc4

HEADER_LENGTH = 44
PEERLISTREQUEST_LENGTH = 28
PEERLISTREPLY_LENGTH = 450
PROXYREPLY_LENGTH = 304
PEER_STRUCT_LENGTH = 45
TYPE_PEERLISTREQUEST = 2
TYPE_PEERLISTREPLY = 3
TYPE_PROXYREPLY = 6
TYPE_PROXYANNOUNCE = 50

def rc4decrypt(key, payload):
   decrypted = []
   r = rc4.new(key)
   dec = r.decrypt(payload)
   for c in dec:
      decrypted.append(ord(c))
   return decrypted

def xordecrypt(payload):
   decrypted = []
   decrypted.append(ord(payload[0]))
   for i in range(1, len(payload)):
      decrypted.append(ord(payload[i]) ^ ord(payload[i-1]))
   return decrypted

def print_header(rnd, ttl, lop, type, header):
   sessionid = header[4:24]
   sourceid = header[24:44]
   print "rnd:        ", rnd
   print "ttl:        ", ttl
   print "lop:        ", lop
   print "type:       ", type
   print "session id: " + " ".join(hex(n) for n in sessionid)
   print "source id:  " + " ".join(hex(n) for n in sourceid)

def verify_packet_length (lop, type, length):
   if type == TYPE_PEERLISTREQUEST:
      expected_length = HEADER_LENGTH + PEERLISTREQUEST_LENGTH +
lop
      if expected_length == length:
         print "*** Peer List Request Packet - lop is correct ***"
```

Daryl Ashley, ashley@utexas.edu

```
            return 1

    if type == TYPE_PEERLISTREPLY:
        expected_length = HEADER_LENGTH + PEERLISTREPLY_LENGTH + lop
        if expected_length == length:
            print "*** Peer List Reply Packet - lop is correct ***"
            return 1

    if type == TYPE_PROXYREPLY:
        expected_length = HEADER_LENGTH + PROXYREPLY_LENGTH + lop
        if expected_length == length:
            print "*** Proxy Reply Packet - lop is correct ***"
            return 1

    if type == TYPE_PROXYANNOUNCE:
        expected_length = HEADER_LENGTH + PROXYREPLY_LENGTH + lop
        if expected_length == length:
            print "*** Proxy Announce Packet - lop is correct ***"
            return 1
return 0

def decode_peerlistrequest(payload):

    print "Decoded Peer List Request:"
    identifier = payload[HEADER_LENGTH:HEADER_LENGTH+ 20]
    random = payload[HEADER_LENGTH+20:HEADER_LENGTH+28]
    print "identifier: " + " ".join(hex(n) for n in identifier)
    print "random:     " + " ".join(hex(n) for n in random)

def decode_peerstruct(peerstruct):
    iptype = peerstruct[0]
    peerid = peerstruct[1:21]
    ipv4addr = peerstruct[21:25]
    ipv4port = peerstruct[25:27]
    ipv6addr = peerstruct[27:43]
    ipv6port = peerstruct[43:45]

    print "ip type:      ",  iptype
    print "peer id:      " + " ".join(hex(n) for n in peerid)
    print "ipv4 address: " + ".".join(str(n) for n in ipv4addr)
    print "ipv4 port:    ", struct.unpack("<h", struct.pack("BB", ipv4port[0],
ipv4port[1]))[0]
    print "ipv6 address: " + " ".join(hex(n) for n in ipv6addr)
    print "ipv6 port:    " + " ".join(hex(n) for n in ipv6port)

def decode_peerlistreply(payload):
```

```
    print "Decoded Peer List: "
    for i in range(0, 10):
        begin = i * PEER_STRUCT_LENGTH + HEADER_LENGTH
        end = begin + PEER_STRUCT_LENGTH
        decode_peerstruct(payload[begin:end])

def decode_proxyreply(payload):
    iptype = payload[HEADER_LENGTH:HEADER_LENGTH+4]
    proxyid = payload[HEADER_LENGTH+4:HEADER_LENGTH+24]
    ipv4addr = payload[HEADER_LENGTH+24:HEADER_LENGTH+28]
    ipv4port = payload[HEADER_LENGTH+28:HEADER_LENGTH+30]
    ipv6addr = payload[HEADER_LENGTH+30:HEADER_LENGTH+46]
    ipv6port = payload[HEADER_LENGTH+46:HEADER_LENGTH+48]

    print "ip type:      " + " ".join(hex(n) for n in iptype)
    print "peer id:      " + " ".join(hex(n) for n in proxyid)
    print "ipv4 address: " + ".".join(str(n) for n in ipv4addr)
    print "ipv4 port:    ", struct.unpack("<h", struct.pack("BB", ipv4port[0],
ipv4port[1]))[0]
    print "ipv6 address: " + " ".join(hex(n) for n in ipv6addr)
    print "ipv6 port:    " + " ".join(hex(n) for n in ipv6port)
```

Appendix 2: XOR Packet Python Script

```
import ZeusHost as zeus
import dpkt

filename = "xor.pcap"
def main():
   for ts, pkt in dpkt.pcap.Reader(open(filename, 'r')):
      eth = dpkt.ethernet.Ethernet(pkt)
      if eth.type!=dpkt.ethernet.ETH_TYPE_IP:
         continue
      ip = eth.data

      if ip.p == dpkt.ip.IP_PROTO_UDP:
         udp = ip.data
         print "UDP source port", udp.sport
         if udp.sport == 16503:
            payload = udp.data
            # Decrypt the packet payload
            decrypted = zeus.xordecrypt(payload)
            # Map the first 4 bytes
            rnd = decrypted[0]
            ttl = decrypted[1]
            lop = decrypted[2]
            type = decrypted[3]
            length = len(decrypted)
            # Use lop and type fields to verify that this is possibly a Zeus packet
            if zeus.verify_packet_length(lop, type, length):
               print "Length of UDP packet: ", length
               # Print the Zeus packet header
               zeus.print_header(rnd, ttl, lop, type, decrypted)
               # Decode Peer List Requess
               if type == zeus.TYPE_PEERLISTREQUEST:
                  zeus.decode_peerlistrequest(decrypted)
               # Decode replies to peer list requests
               if type == zeus.TYPE_PEERLISTREPLY:
                  zeus.decode_peerlistreply(decrypted)
               # Decode replies to proxy requests
               if type == zeus.TYPE_PROXYREPLY:
                  zeus.decode_proxyreply(decrypted)

if __name__=="__main__":
   main()
```

Appendix 3: RC4 Packet Python Script

```
import ZeusHost as zeus
import dpkt

CLIENTKEY="darylashley"
filename = "rc4.pcap"

def main():
    for ts, pkt in dpkt.pcap.Reader(open(filename, 'r')):
        eth = dpkt.ethernet.Ethernet(pkt)
        if eth.type!=dpkt.ethernet.ETH_TYPE_IP:
            continue
        ip = eth.data
        if ip.p == dpkt.ip.IP_PROTO_UDP:
            udp = ip.data
            if udp.dport > 0:
                payload = udp.data
                # Decrypt the packet payload
                decrypted = zeus.rc4decrypt(CLIENTKEY, payload)
                # Map the first 4 bytes
                rnd = decrypted[0]
                ttl = decrypted[1]
                lop = decrypted[2]
                type = decrypted[3]
                length = len(decrypted)
               # Use lop and type fields to verify that this is possibly a Zeus packet
                if zeus.verify_packet_length(lop, type, length):
                    print "Length of UDP packet: ", length
                    # Print the Zeus packet header
                    zeus.print_header(rnd, ttl, lop, type, decrypted)
                    # Decode Peer List Requess
                    if type == zeus.TYPE_PEERLISTREQUEST:
                        zeus.decode_peerlistrequest(decrypted)
                    # Decode replies to peer list requests
                    if type == zeus.TYPE_PEERLISTREPLY:
                        zeus.decode_peerlistreply(decrypted)
                    # Decode replies to proxy requests
                    if type == zeus.TYPE_PROXYANNOUNCE:
                        zeus.decode_proxyreply(decrypted)

if __name__=="__main__":
    main()
```

Daryl Ashley, ashley@utexas.edu