Check Point
SOFTWARE TECHNOLOGIES LTD.

WE SECURE THE FUTURE

Technical Report for "Offline" Ransomware

Stanislav Skuratovich
Malware Reverse Engineering Team
20/10/2015

Overview

The ransomware sample we researched encrypts all files on the infected computer system that have the specified extensions (see **Appendix A**). Most known ransomware must connect to the C&C server before performing encryption. However, this particular sample doesn't need such a connection to begin the process. This eliminates the possibility of intercepting the keys exchange between the infected machine and the C&C server.

Another noteworthy feature we found during our analysis is that the payload is written in Pascal-based language.

Functionality

Installation Process

The ransomware is packed using a packer written in *Visual Basic* language, compiled to P-code. To decrypt the body of its own code, the packer restarts the process a few times, using sections (un)mapping, overwriting, and thread context changes. After the body of the ransomware is decrypted, it performs some preparations for the file encryption process. The main body of the ransomware is written in Pascal-based language. It also uses an external library for operations with large numbers *(FGInt*[1]).
To stay persistent on the infected machine, the ransomware creates the following registry key, previously copying the original executable to the `%ProgramFiles%` directory:

```
Key: "HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Run\"
Value: "pr" = "%ProgramFiles%\"${RANSOMWARE_PATH}
```

To identify an infected computer and the date when the encryption was performed, the ransomware generates the following information string:

```
[A-Z]{36}-[\d]{2}\.[\d]{2}\.[\d]{2} [\d]{2}@[\d]{2}@[\d]{7}
```

Field	Description

[1] https://github.com/SnakeDoctor/FGInt

`[A-Z]{36}`	String of random characters that most likely specifies computer ID
`[\d]{2}\.[\d]{2}\.[\d]{2}`	Infection date in format *YYYY.MM.DD*
`[\d]{2}@[\d]{2}`	Infection time in format *HH:mm* (characters *":\\/*?<>\"|"* are replaced with *@*)
`[\d]{7}`	Randomly generated number (purpose is not known)

The ransomware sends information about the infected computer to the following URL, using this User-Agent (taken from the analyzed environment):

```
URL: http://google-update.com/install/inst.php

User-Agent: Mozilla/5.0 (Windows NT 6.3; WOW64) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/36.0.1985.125 Safari/537.36
```

Information is sent using the HTTP GET method in this format:

```
${url}?ver=${ver}&id=${identification_info}&sender=${sender}
```

Each field is described below:

Field	**Description**
`ver`	Version of the ransomware (*CL-1.0.0.0*)
`id`	Infected computer identification string (see above)
`sender`	Possibly the campaign name (‘*prizrak*’)

The ransomware saves the identification string of the infected computer in this file:

```
%ProgramFiles%\${random_filename}
```

The structure of the generated file:

```
BCDLID
ended
${identification_info}
```

The ransomware does not generate an identification string for the infected computer if it found a file in the `%ProgramFiles%` directory that contains this string:

```
BCDLID
```

Files Encryption Process

The ransomware uses different algorithms to encrypt a file. The beginning of the file (first *30000* bytes or less) is encrypted using a custom algorithm (see *Preliminary Encryption*).

The rest of the file (in blocks of *1024* bytes) is encrypted selectively using the RSA algorithm (see *Main Encryption*).

Keys Generation for Encryption

The ransomware randomly generates special buffers to encrypt files on the infected system. Algorithms used for buffer generation are described below.

RSA Keys Pair Generation

The RSA[2] keys pair is used in the *Main Encryption* routine. These steps are performed to generate the keys pair:

1. Randomly generate two buffers of length 48 bytes that contain values in the following range:

   ```
   [0-100)
   ```

   Both buffers represent numbers with a base equal to *256* bytes. One buffer represents the *p* number, and the other buffer represents the *q* number in the RSA algorithm. The RSA related number is then calculated:

   ```
   N = p * q
   ```

2. Perform operations according to the RSA key generation rules to calculate the *d* RSA related number. The RSA related number *e* is hardcoded and the value is:

   ```
   e = 65537
   ```

3. Check if the generated numbers are consistent with the RSA algorithm. If the numbers are not consistent, the ransomware performs all operations from the beginning.

   If the generated numbers are consistent with RSA algorithm, then the ransomware saves the pair *{N, e}* (see *RSA_e_N*) that will be used in *Main Encryption* and the pair *{d, N}* (see *RSA_d_N*).

Internal Buffers Generation

Two internal buffers are generated and used in the *Preliminary Encryption* routine. The ransomware performs these steps:

---

[2] https://en.wikipedia.org/wiki/RSA_(cryptosystem)

1. Randomly generate a buffer of length *2048* bytes that consists of upper and lower case letters and digits (see *PRE_ULD_2048*).
2. Randomly generate a buffer of length 20 bytes that consists of digits (see *PRE_DIGITS_20*).

Encryption Routine

The ransomware encrypts files that contain certain extensions (see **Appendix A**). To verify if the file is already encrypted, the ransomware checks if the end of the file ends with this string (see **Encrypted File Format**):

```
{ENCRYPTENDED}
```

If the file ends with the specified string, the ransomware skips it and proceeds to the next file.

Preliminary Encryption

The ransomware encrypts the beginning of the file (*30000* bytes or less). At first, the ransomware generates *512* random numbers in the following range:

```
[0-2048)
```

Previously generated *512* numbers are used as indexes in *PRE_ULD_2048*. The ransomware takes the values specified by these indexes and creates a new buffer with a length of *512* bytes that contains upper and lower case letters and digits (see *PRE_ULD_512*). The pseudocode is represented below:

```
PRE_ULD_512 = ''
for i in range(512):
    PRE_ULD_512 += PRE_ULD_2048[random.randint(0, 2047)]
```

These generated indexes are saved to the metadata of the encrypted file (see **Encrypted File Format**).

The ransomware performs the following steps in a loop to generate a final buffer used for encryption (see *PRE_MD5_30000*). The ransomware stops the loop when the length of the *PRE_MD5_30000* buffer exceeds *30000* bytes.

1. Calculate the MD5 hash sum of the *PRE_ULD_512* buffer, then convert that MD5 hash sum to text form and append it to the *PRE_MD5_30000* buffer.
2. Change the values in the *PRE_ULD_512* buffer according to the specified rules.
3. Calculate the MD5 hash sum of the *PRE_ULD_512* buffer, then convert that MD5 hash sum to text form and append it to the *PRE_MD5_30000* buffer.
4. Change the values in the *PRE_ULD_512* buffer according to the specified rules.

5. Calculate the MD5 hash sum of the *PRE_ULD_512* buffer, then convert that MD5 hash sum to text form and append it to the *PRE_MD5_30000* buffer.
6. Change the values in the *PRE_ULD_512* buffer according to the specified rules.
7. Go back to the first operation.

Note that the rules used for modifying buffers are different. The pseudocode is presented below:

```
PRE_MD5_30000 = ''
while len(PRE_MD5_30000) < 30000:
    # append MD5 hash sum
    PRE_MD5_30000 += md5(PRE_ULD_512).hexdigest().upper()
    # modify PRE_ULD_512 buffer operations
    for i in range(len(PRE_ULD_512)):
        b = ord(PRE_ULD_512[i]) + 0x80
        if b > 0xFF:
            b = ord(PRE_ULD_512[i]) - 0x80
        PRE_ULD_512 = PRE_ULD_512[:i] + chr(b) + PRE_ULD_512[i + 1:]
    # append MD5 hash sum
    PRE_MD5_30000 += md5(PRE_ULD_512).hexdigest().upper()
    # modify PRE_ULD_512 buffer operations
    for i in range(2, len(PRE_ULD_512)):
        b = (ord(PRE_ULD_512[i - 2]) + ord(PRE_ULD_512[i - 1])) & 0xFF
        PRE_ULD_512 = PRE_ULD_512[:i] + chr(b) + PRE_ULD_512[i + 1:]
    # append MD5 hash sum
    PRE_MD5_30000 += md5(PRE_ULD_512).hexdigest().upper()
    # modify PRE_ULD_512 buffer operations
    for i in range(len(PRE_ULD_512)):
        b = (ord(PRE_ULD_512[i]) * 2) & 0xFF
        PRE_ULD_512 = PRE_ULD_512[:i] + chr(b) + PRE_ULD_512[i + 1:]
```

After all the preparatory steps are finished, the ransomware starts the file encryption process:

1. Take one value at a time from the *PRE_DIGITS_20* and one value at a time from the *PRE_MD5_30000* buffers.
2. Take one value at a time from the original file content.
3. Pass these *3* (three) bytes to the encryption function that performs a simple mathematical operation and returns an encrypted byte.

The pseudocode for is presented below:

```
def encrypt_data(file_data, PRE_MD5_30000, PRE_DIGITS_20):
    enc_data = ''
    fs = min(30000, len(file_data))
    for i in xrange(fs):
```

```
        enc_data += pack('<B', encrypt_byte(PRE_DIGITS_20[i % 20],
                                            unpack('<b', PRE_MD5_30000[i])[0],
                                            unpack('<b', file_data[i])[0]))
    return enc_data


def encrypt_byte(sc, md5_byte, file_byte):
    if sc == '0':
        return (file_byte + md5_byte) & 0xFF
    if sc == '1':
        return (file_byte - md5_byte) & 0xFF
    if sc == '2':
        return (file_byte + md5_byte + ord(sc)) & 0xFF
    if sc == '3':
        return (file_byte + md5_byte + int(sc)) & 0xFF
    if sc == '4':
        return (file_byte - md5_byte - ord(sc)) & 0xFF
    if sc == '5':
        return (file_byte - md5_byte - int(sc)) & 0xFF
    if sc == '6':
        return (file_byte + int(sc)) & 0xFF
    if sc == '7':
        return (file_byte - int(sc)) & 0xFF
    if sc == '8':
        return (file_byte + ord(sc)) & 0xFF
    if sc == '9':
        return (file_byte - ord(sc)) & 0xFF
    return None
```

The buffer with the encrypted content is written to the file instead of the original content.

Main Encryption

When the size of the file exceeds *30000* bytes, the ransomware encrypts it in blocks. The size of each encrypted block is equal to *1024* bytes. Depending on the size of the file (up to *3* blocks), the ransomware randomly chooses where to start the offsets of the encryption blocks and the number of blocks to encrypt.
The ransomware encrypts each chosen block using the RSA algorithm. The *RSA_e_N* (see **RSA Keys Pair Generation**) is used as a public key. The ransomware stores information about the size of the cipher text and starts the offset of the encrypted chunk with these tags:

```
{BLOCKSSTART}-{BLOCKSEND}
```

If the size of the cipher text is less than *1024* bytes, the ransomware performs padding with '*\x00*' bytes to *1024* bytes. If the size of the cipher text exceeds *1024* bytes, the ransomware stores the overflow bytes in the metadata (see *stolen bytes*). For more detailed information, see **Encrypted File Format.**

Encrypted File Format

As the ransomware should be able to decrypt the encrypted data, it stores a metadata at the end of the file. The format is easy to read and is presented with a detailed description below:

Color	Meaning
Red	Start/End tags
Green	Names of relevant variables for file encryption/decryption
Blue	Comments
Violet	Metadata

```
{ENCRYPTSTART} // Start tag for encrypted file metadata
  {[0-9]+}:    // Encrypted and encoded PRE_ULD_2048
  {[0-9]+}:    // Encrypted and encoded RSA Private Key RSA_d_N used for
               // main decryption
  {[0-9]+}:    // Encrypted and encoded PRE_DIGITS_20
  {[0-9]+}:    // Encrypted and encoded Computer ID and
               // Infection Timestamp Information (Check integrity
               // with plaintext Computer ID and Infection Timestamp)
  {[0-9]}      // Encrypted and encoded campaign information 'prizrak'
  {[A-Z]{36}-${TIMESTAMP}} // Computer ID and
                           // Infection Timestamp Information
                           // (${identification_info})
  {[0-9]+}     // Original size of encrypted file
  {CL-1.0.0.0}      // Most likely ransomware version
  {${ORIGINAL_FILENAME}} // Original name of encrypted file
  {[0-9A-F]{32}}    // MD5 hashsum of 3 concatenated buffers
                    // (PRE_ULD_2048, RSA_d_N, PRE_DIGITS_20). Possibly
                    // used for integrity check during decryption
  {[0-9]+ … [0-9]+} // Array of numbers used as indexes in PRE_ULD_2048
                    // buffer. Generated buffer PRE_ULD_512 is used
                    // in Preliminary encryption routine
  {BLOCKSSTART}     // Start tag for Main Encryption
    {[0-9]+}        // (Offset_x >> 10) == Modified offset in file
                    // where 1st Main encrypted block starts
    {[0-9]+}        // Cipher text length
    {${ENCRYPTED_DATA_PART_x}} // Piece of 1st cipher text that length
                               // exceeds 1024 bytes (stolen bytes)
    …
```

```
    {[0-9]+}          // (Offset_y >> 10) == Modified offset in file
                      // where n^th Main encrypted block starts
    {[0-9]+}          // Cipher text length
    {${ENCRYPTED_DATA_PART_y}} // Piece of n^th cipher text that length
                               // exceeds 1024 bytes (stolen bytes)
  {BLOCKSEND}         // End tag for Main Encryption
  {[0-9]+}            // Total length of metadata till the {BLOCKSEND} tag
{ENCRYPTENDED}        // End tag for encrypted file metadata
```

As most of the fields are self-explanatory, only these are described below: the encryption and encoding routine for *Preliminary Encryption* & *Main Encryption* buffers, campaign information, computer ID, and infection timestamp.

The ransomware performs the following steps to encrypt and encode the data:

1. Append this string to the beginning of each plaintext:

```
Asshole
```

2. Encrypt each modified plaintext *3* (three) times using *3* (three) hardcoded RSA keys one at a time: ***KEY C***, ***KEY B***, ***KEY A*** (see **Appendix B**).
3. Encode each cipher text using the following algorithm:

```
def encode_metadata(datalist):
    metadata = ''
    for data in datalist:
        for d in data:
            d_n = ord(d)
            if d_n < 10:
                metadata += '00%u' % d_n
            elif d_n < 100:
                metadata += '0%u' % d_n
            else:
                metadata += '%u' % d_n
        metadata += ':'
    return metadata[:-1]
```

File Decryption Process

In this section we describe the possible decryption process. As we don't have access to the decrypter used by the ransomware author(s), the following is based on the technical-related information we obtained.

Decryption Process

As all the needed data for file decryption is stored as metadata at the end of encrypted file, decryption is possible if the *Preliminary Encryption* and *Main Encryption* buffers are decoded and decrypted. The routine responsible for encoding `encode_metadata` (see **Encrypted File Format**) is easy invertible, so the only thing left for us to do is to decrypt RSA encrypted buffers. The pseudocode for the decoding routine is presented below:

```
def decode_metadata(datalist):
    buffers = list()
    enc_buffers = datalist.split(':')
    for eb in enc_buffers:
        if len(eb) % 3:
            raise Exception('Encoded data invalid size')
        dec_buffer = ''
        for i in xrange(0, len(eb), 3):
            b = eb[i:i+3].lstrip('0')
            b = int(b) if b != '' else 0
            dec_buffer += chr(b)
        buffers.append(dec_buffer)
    return buffers
```

As was mentioned previously, the ransomware encrypts these buffers using *3* hardcoded public keys one at a time. To decrypt these buffers, we need the *3* private keys which will make the RSA key pairs with the *3* hardcoded public keys. It is not possible to decrypt the encoded files in a reasonable period of time, as factorizing *768* bits RSA key and finding *{d, N}* requires approximately *2* years using multiple computers.[3]In our description of the decryption process, we assume that we have the *3* RSA private keys.

Preliminary Decryption

As all metadata is decrypted, the *PRE_ULD_2048* and *PRE_DIGITS_20* buffers are decrypted as well. Randomly generated numbers (*512*) that were used as indexes in the *PRE_ULD_2048* buffer are stored in metadata. To generate the *PRE_MD5_30000* buffer, the same steps are used as in the encryption routine (see **Encryption Routine**).

[3] http://iamnirosh.blogspot.com.by/2015/02/factoring-rsa-keys.html

After all the preparatory steps are finished, the decrypter starts the process of file decryption:

1. Take one value at a time from the *PRE_DIGITS_20* and one value at a time from the *PRE_MD5_30000* buffers.
2. Take one value at a time from encrypted file content.
3. Pass these *3* (three) bytes to the decryption function that performs a simple mathematical operation and returns a decrypted byte.

The pseudocode for the decryption routine is presented below:

```
def decrypt_data(enc_file_data, PRE_MD5_30000, PRE_DIGITS_20):
    dec_data = ''
    fs = min(30000, len(enc_file_data))
    for i in xrange(fs):
        dec_data += pack('<B', decrypt_byte(PRE_DIGITS_20[i % 20],
                                            unpack('<b', PRE_MD5_30000[i])[0],
                                            unpack('<b', enc_file_data[i])[0]))
    return dec_data


def decrypt_byte(sc, md5_byte, enc_byte):
    if sc == '0':
        return 0x100 + enc_byte - md5_byte if enc_byte < md5_byte else
enc_byte - md5_byte
    if sc == '1':
        return enc_byte + md5_byte
    if sc == '2':
        b = ord(sc) + md5_byte
        return 0x100 + enc_byte - b if enc_byte < b else enc_byte - b
    if sc == '3':
        b = int(sc) + md5_byte
        return 0x100 + enc_byte - b if enc_byte < b else enc_byte - b
    if sc == '4':
        return enc_byte + md5_byte + ord(sc)
    if sc == '5':
        return enc_byte + md5_byte + int(sc)
    if sc == '6':
        return 0x100 + enc_byte - int(sc) if enc_byte < int(sc) else enc_byte
- int(sc)
    if sc == '7':
        return enc_byte + int(sc)
    if sc == '8':
        return 0x100 + enc_byte - ord(sc) if enc_byte < ord(sc) else enc_byte
- ord(sc)
    if sc == '9':
        return enc_byte + ord(sc)
    return None
```

Main Decryption

As all metadata is decrypted, the *RSA_d_N* RSA private key is decrypted as well. The *RSA_d_N* is the private key which forms a pair with the *RSA_e_N* public key. These keys are generated during the preparation for file encryption (see **RSA Keys Pair Generation**). As *RSA_e_N* was used for encryption, *RSA_d_N* is used for decryption. After the cipher text is decrypted, the decrypter overwrites data in the encrypted file on the offset specified in the metadata, using the following mathematical operation:

```
OriginalFileOffset = MetadataFileOffset << 10
```

The decrypter performs the same operations for all the encrypted blocks.

Additional Information

Researched sample:
SHA1: c30b1b23fa75acb6a0fbeca6fb2d64b3a5a2ab36
SHA256: 1caf864c9b28b4f72a8dea5db128aeae9cd79d1063baa86c0d383d67d0fdacb5

Appendix A – Encrypted Files Extensions

.113	.docx	.ldf	.orf	.raw	.wab
.1cd	.dt	.m2v	.p12	.rtf	.wps
.3gp	.dwg	.m3d	.pdf	.rwl	.wps
.7z	.fbf	.max	.pef	.rx2	.x3f
.accdb	.fbk	.mdb	.ppsx	.sbs	.xls
.arj	.fbw	.nbd	.ppt	.sn1	.xlsb
.asm	.fbx	.nrw	.pptm	.sna	.xlsk
.bak	.fdb	.nx1	.pptx	.spf	.xlsm
.cdr	.gbk	.odb	.pst	.sr2	.xlsx
.cer	.gho	.odc	.ptx	.srf	.zip
.cpt	.gzip	.odp	.pwm	.srw	
.csv	.jpeg	.ods	.pz3	.tbl	
.db3	.jpg	.ods	.qic	.tib	
.dbf	.key	.odt	.r3d	.tis	
.doc	.keystore	.old	.rar	.txt	

Appendix B – RSA Public Keys Used For Metadata Encryption

```
------------------------------ KEY A -------------------------------------
dec:
662779388586780303538494866342702933792030483897051903527680690903308651875867711522917241525768049072800838212410352953701726830479565455740613841556879406800825965981848871900652136540769465780676572748771815364771934237545984489:65537
hex:
6D49B740C4084BC47AA59110E29989E24C2C4F86D6A447005608E0E2E2B5E7535C9561A5F8EEA4A6C3D600C7FE5B2CFB249AC341B00257026032E74234ED51BF382F08154B915833D1CAFA398F90C16A45877EC5DCB345B4189E7D2BE773F1E9:10001

------------------------------ KEY B -------------------------------------
dec:
139716505558976822770659009294299492691531410194395846373252232920930188767405394250365424511425046836362350201255744144733898222119858849323070714092160910764583931376213652423737364499228173516805776980337087128602059087635515041:65537
hex:
1709D00C8A3A755802BB1DBC72AB8982B44A43BCD471471D230FB0935A982B1825227F1F06B6C1764A0F2D7B0F2FF252832211F63BE89A6D510A781BB0BCC4A68E4529D9BA0D23A178E058059E5562B5B9F8926E6639CE0E9FA47CDC4FAF86A1:10001

------------------------------ KEY C -------------------------------------
dec:
559611419631854938133641349573555815604974811205505339453066703156238651122762295928380065175410779004543613448789183577569257851705952694301090956677836877721330046053679412662561119767661372553341227925535274114602929369989758591:65537
hex:
5C46B74C860ED16DFAFFE8E0A5ACB499677DFB30E78FA634B0AD9CC831858AB2DDA4C5B522AFFC5AEE93D0626C10C3AC799ADB089FF656ED9C67E713493D805F523CFAE83CE3483AAAB329FA358845F71E896E58E3B811402B289E209BABBE7F:10001
```

Special thanks to Aliaksandr Trafimchuk for help in researching this ransomware.