https://medium.com/@ryancor/extracting-embedded-payloads-from-malware-aaca8e9aa1a9

Mar 4, 2020

Extracting Embedded Payloads From Malware

One of my all time favorite subfields of reverse engineering is the dissection of viruses. In this article I will be exploring malware from the infamous APT29 adversarial group. I will extricate an embedded executable from the main loader that has been classified as Coll Cozy Bear. This loader will need its DOS Stub / PE header reconstructed. Let's get started.

Below are the Indicators of Compromise (IOC) from the dynamic link library (DLL) we are about to open in a disassembler. An IOC is a piece of forensic data such as observed artifacts found on a network or in an operating system. These consist of hash values, IP addresses, exploit tools, and TTPs (Tactics, Techniques, and Procedures).

- MD5(AudioSes.dll)= 16bbc967a8b6a365871a05c74a4f345b
- SHA1(AudioSes.dll)= 9858d5cb2a6614be3c48e33911bf9f7978b441bf
- SHA256(AudioSes.dll)= b77ff307ea74a3ab41c92036aea4a049b3c2e69b12a857d26910e535544dfb05

Before we dive into our lab experiment, let's briefly discuss what a DOS stub is and why 64 bit .exe files need to have one in the header.

Background

When the PE format was introduced in 1994, DOS was still very much alive and in active use. The DOS stub was needed to make sure that Windows' executables are compatible with a DOS loader but not run the risk of one being ran on unsuitable or older kernel versions. An example of this header in action would be if you attempted to run a Windows NT executable on an MS-DOS kernel, you would get this message: "This program cannot be run in DOS mode.".

The PE file format is put together as a linear stream of data. It starts with the MS-DOS MZ Header, a real-mode program stub, and a PE file signature. Next is the file headers and optional header which tells the Windows operating system what architecture the binary is (32-bit, 64-bit) and the executable format type (driver, library, executable).

Below we can take a peek at the headers inside a hex editor (Hex Fiend):

The PE standard fields are where things can get a little puzzling when trying to repair broken headers. Up until now, the DOS header and DOS stub are consistent with one another; meaning they never change no matter the executable type. These fields contain extensive information that is fundamental for loading and running an executable. This follows suit with the optional headers field that include the ImageBase, Major and Minor OperatingSystemVersion, SizeOfImage, and Windows Subsystem field that determines if any help is required to run the image.

Now that we have a bit more experience with the PE formats, lets navigate over to VirusTotal[.]com and search the hashes I shared at the beginning of this write-up. From there we can download the file onto our test machine and start our analysis.

➜ file AudioSes.dll

AudioSes.dll: PE32+ executable (DLL) (GUI) x86-64 (stripped to external PDB), for MS Windows

Running the file command, we can confirm that this file is a DLL. Rather than throwing this into a debugger like we did in our last exercise, we'll most likely have to gather intelligence on this malware through static analysis.

Research Exercise

For this lab, I'm going to use IDA Pro, but any flavor of disassemblers or decompilers will work; Ghidra, Hopper, Radare2, or Binary Ninja, you name it.

The first function hiding in plain sight when opening this file up will be the DLL Main Entry point:

Interestingly enough, usually at this point we would see some call to another component from the entry point, however, we don't see much. It is common for malware developers to place an initialization function from the entry point using DLL_PROCESS_ATTACH; this would load the defined process. Which is why if we were to try registering this DLL with regsvr32.exe, nothing would happen. This can only mean that there is going to be an interesting exported function to examine. If we scroll on over to the list of imports and exports, you'll notice that there are two elements in use.

PointFunctionCall() was not invoked when looking into the DllEntryPoint. But, because it was placed in this specific section, it means that another binary can call this module out by loading the library at runtime using the LoadLibrary() Win32 API call. Our options seem fairly limited on what to look at here, so let's head on over to this function call in the disassembler and see if we can find anything of merit.

The malware loader is calling VirtualAlloc() at the start, and an allocation used in this context makes me believe this DLL is unpacking some sort of program code in memory, we just don't know what at the moment. Virtual allocation takes four parameters, the first being lpAddress, the starting address of the region to allocate. The second parameter is the size of the region, in bytes. The last two are flAllocationType and flProtect, these are responsible for the type of memory allocation the malware author wants to establish; such as, memory commits versus memory reserves. flProtect's definition is already in the name, the malware wants to protect this region of memory it is allocating (I.E. PAGE_NOACCESS, PAGE_GUARD). Understanding this bit, we can come up with some pseudo-C code that looks like this:

someBuffer = VirtualAlloc(lpAddress, 0x46A00ui64, 0x3000u, 4u);

The second number is obvious to us, it's allocating 289,280 (0x46A00ui64) bytes, but what does hex 0x3000u or 4u mean in terms of allocation type and protections? After referencing Microsoft's documentation, you'll quickly find the meaning of these values. The value 0x3000u is typically MEM_COMMIT | MEM_RESERVE which reserves and commits an allocation in one step. The value 4u in the last parameter is a protection mechanism that is defined as PAGE_READWRITE; enabling read-only or read/write access to the committed region of pages. If Data Execution Prevention is enabled, attempting to execute code in the committed region results in an access violation.

Moving down a few lines in the disassembly we come to a fairly obvious memcpy():

The malware is calling on rep movsb, and the movsb instruction can be preceded by the rep prefix for block moves of ECX bytes, words, or doublewords. What's happening here is a block that is the size of 0xEF (239) bytes from data address 0x00402008 (labeled as unk_402008) is being copied into register RDI. A decompiled look at this operation would look something like this:

```
qmemcpy(someBuffer<RDI>, unk_402008, 0xEF);
```

Double-clicking on unk_402008 will take us to the .data section and present us what seems to be unknown data/bytes.

Looks like a bunch of garbage right? Right. Might be safe to assume these unrevealed bytes will be used in a decryption routine as data from address unk_402008 got copied over into a buffer; which we will see used later on below in a XOR loop. Going back to PointFunctionCall(), the next operation is our most important hint yet:

FindResourceA is used to determine the location of a resource with the specified type and name in the specified module. Always proceeding, is LoadResource, which retrieves a handle that can be used to obtain a pointer to the first byte of the specified resource in memory. To put it differently, this function loads a resource from a PE file into memory. Malware will use resource sections to store strings, configuration information, or other malicious files. Let's decompile what we see above so it is easier to read (Assembly can be perplexing to look at after all).

```
someBuffer = FindResourceA(hModule, 1, 2);

LoadResource(hModule, someBuffer);
```

You might be asking yourself what the 1 and 2 are in the parameters for FindResourceA(). To put it simply, the last two parameters are set for Name and Type of the resource section, so if there are multiple resources, the malware can know where to look specifically. It is safe to say if one were to take a look into the resource section, we will uncover something. To do so, we can use a great program called wrestool that can extract both icons, cursors, and files from 32-bit (PE) and 16-bit (NE) executables and libraries.

```
➜ wrestool AudioSes.dll
```

--type=2 --name=1 --language=0 [type=bitmap offset=0x5060 size=289339]

--type=16 --name=1 --language=0 [type=version offset=0x4badc size=636]

Things are starting to add up for us at this point, the 2 for type, and the 1 for name which appeared in our disassembly are also appearing to be valid resources. What is even more captivating for us is that the size of that resource is a familiar number we saw earlier (289,280) with the malware's call to VirtualAlloc(). Our goal now is to pull out the resource section and check out the embedded payload.

➜ wrestool --name=1 --type=2 -R -x AudioSes.dll > extracted.pe➜ file extracted.pe

extracted.pe: PC bitmap, Windows 3.x format, 32 x 32 x 24

Hmmm, a BMP file? Something doesn't seem right here. Time to throw this into Hex Fiend.

It is pretty apparent what is happening here, the section headers are all present but the DOS Header, DOS Stub, PE Signature and Standard Fields are all missing and have been replaced with a BMP file signature (424D38) along with hexadecimal 0xFF's. Unfortunately from how much is missing, we can't just grab any PE header from any Win32 executable and replace the bytes in this file, it will cause us to have a corrupted binary. Instead, we'll navigate back to the IDA Pro disassembly; finding the missing stub will solve this problem.

Immediately following the LoadResource() call is another transfer of bytes in memory that look like this:

```
qmemcpy(pBuffer, someBuffer + 298, 0x46911ui64);
```

Why is the memcpy() now pointing the newly received data from memory + 298? Also, now the size has gone from 0x46A00 to 0x46911. If we take a look at the hexdump of the embedded payload we have retrieved using wrestool, notice that is the exact size of the corrupted BMP headers.

```
➜ head -c 298 extracted.pe | hexdump
0000000 42 4d 38 0c 00 00 00 00 00 00 36 00 00 00 28 00
0000010 00 00 20 00 00 00 20 00 00 00 01 00 18 00 00 00
0000020 00 00 02 0c 00 00 12 0b 00 00 12 0b 00 00 00 00
0000030 00 00 00 00 00 00 ff ff ff ff ff ff ff ff ff ff
0000040 ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
*
0000120 ff ff ff ff ff ff ff ff ff ff
```

000012a

Using some simple math, the encrypted bytes we saw earlier is close to the same size of the BMP stub;

>>> 0x46A00 - 0x46911

239 (Size of enc_bytes)

>> 298 (Size of BMP headers)

The encrypted data from address 0x00402008 is most likely going to be our fixed PE headers, and moving down in the code base, a decryption algorithm is about to load up in the queue. This is a two step decoding process, first let's step over to address 0x004010CE:

The transaction occurring here is the transfer of lpAddress to RSI, that holds the pointer to the BMP buffer bytes. RCX gets handed over the size of the encrypted address bytes. Before the loop starts at loc_4010E0, the key gets stored in RBX (0xC5). Iteration over the bytes at RSI begins with a XOR operation between first byte at address 0x00402008 and the key at BL (lower 8 bits of RBX). The second important action is in the next line at 0x004010E2; the keys bits are rotated by 1 byte for each incrementation. RSI increases by one then loops back to the XOR instruction. We can pseudo-C this fairly quickly as such:

```
bufferStub = lpAddress;

size = 239;

key = 0xC5;do

{

  *bufferStub ^= key;

  key = __ROR1__(key, 1);

  bufferStub = (bufferStub + 1);

  --size;

} while(size != 0);
```

Be cognizant of when we write our decrypter in python, that we will have to implement our own version of ROR. From here the malware uses the decrypted buffer stub and replaces the corrupted header. The buffer is then used as lpAddress to call out the last function at 0x0040114E. At this point, the subroutine at 0x0040114E allows the malware to load the library from the embedded payload. It then calls out to GetProcAddress() to obtain a specific function pointer that we have not been able to resolve yet because the embedded payload hasn't been decrypted.

```
func = GrabFunctionAddressFromEmbeddedDll(lpAddress);// 0x0040114E()
...
if ( func )
{
  (func)(funcParam, 1i64, 0i64);
  while ( 1 )
    Sleep(0xF00Du);
}
```

The name of this function will be resolved dynamically which is why we can't see it being defined in the decompiler (just labeled as func). And if we resolve the embedded payload, there is a possibility that the subroutine imported at runtime will show in what might be thought of now as another DLL. This a good indicator with typical anti-reversing behavioral patterns seen in malware, as most of the time a reverse engineer will have to debug a program in order to see the resolved function tag.

Putting It All Together

At this point, there is enough information to write our decryption tool in python. This script will be all encompassing; from extracting the payload, to searching the encrypted bytes in memory, decrypting the stub, then rewriting the corrupted headers with the newly decoded one.

The ROR instruction is similar to SHR except the shifted bits are rotated to the other end. We can define the ROR functions as:

```
def rightRotate(n, d):
  return (n >> d)|(n << (8 - d)) & 0xFFFFFFFF
```

We are going to take n and shift right d number of times and bitwise-or the remaining result by n which then shifts to the left 8 bits subtracted by the amount of shifts (d) we need. In our case n=0xC5, d=0x01. A bitwise-and will be needed to make sure all we get back is a byte size instead of something larger like a qword or dword.

Note: an alternative way of writing the rotate right function uses a lambda which takes three arguments instead of two:

```
ror = lambda val, r_bits, max_bits: \
 ((val & (2**max_bits-1)) >> r_bits%max_bits) | \
```

```
(val << (max_bits-(r_bits%max_bits)) & (2**max_bits-1))
```

max_bits is used to define the maximum bits allowed to rotate, in our case with the first non-lambda function, we know that value is going to be 8 since the assembly instruction ROR operates this way.

The next function introduced is going to help the process of extracting the encrypted PE header bytes from the data resource's virtual address. The reason why we are automating this process instead of hardcoding the array of bytes is to help resolve these bytes dynamically in case other variants of this malware ever arise and the encrypted PE header consists of different data bits.

```
def getEncryptedData(filename, starting_offset, ending_offset):
  pe = pefile.PE(filename)
  encrypted_pe_header = []  for section in pe.sections:
    if b".data" in section.Name:
      offset = section.VirtualAddress
      start = offset + starting_offset
      end = offset + ending_offset
      for byte in section.get_data()[start - offset:end - offset]:
        encrypted_pe_header.append(ord(byte))return encrypted_pe_header
```

This function takes three arguments; the target file, offset of the first encoded byte, and the offset of the last byte found in that data array. We iterate through all PE Sections until we find .data. Once that is found, an assignment is made to find the main offset of that section by calling section.VirtualAddress.

For example, if the two bytes => 0x1dc0 is an instruction offset in a file, the four bytes here => 0x00401dc0 is an address in a program's virtual memory where this particular instruction should reside.

Furthermore, we add our offset to the starting_offset and ending_offset which will result in looking something like this: start=0x00401dc0, end=0x00401fa4. The last loop recapitulates through all bytes in the data section by calling section.get_data(), but it is indexed by our specific offsets. Every byte is then appended to our array, and the subroutine returns.

Before the main function is implemented, we have to write the routine for grabbing the resource file using wrestool.

```
def grabResourceFromFile(target_file, output_file):
  bashCommand = "wrestool --name=1 --type=2 -R -x %s" %
                                        (target_file)
```

```
 process = subprocess.Popen(bashCommand.split(),
                            stdout=subprocess.PIPE)
 output, error = process.communicate()
 if error == None:
  file = open(output_file, 'wb')
  file.write(output)
  file.close()
 else:
  print("Error with running wrestool {}".format(error))
  exit(-1)
```

A simple subprocess call should do the trick on invoking command line interface tools. An error check is executed after process.communicate() since it will scan for the existence of the tool in your environment, along with any type of permission error issues. In short, all this module is doing is using the crafted command we made earlier and writing the bytes out to a small binary file. Side note for future extractions, we don't need to shell out our own function to grab the resource section; pftriage has a builtin component that dumps the resource section with a single invocation.

Putting this all together in our main routine will incorporate all the above functions then truncate some of the straggling 0xFF bytes.

```
encrypted_pe_header = getEncryptedData(target_file, 0x08, 0xF7)
for i in range(bufferSize):
 temp = encrypted_pe_header[i] ^ key
 key = rightRotate(key, 1) & 0xff
 pe_header.append(binascii.hexlify(chr(temp)))grabResourceFromFile(target_file, output_file)with open(output_file, 'r+b') as f:
 f.seek(0)
 for byte in range(len(pe_header)):
  f.write(binascii.unhexlify(pe_header[byte]))
 f.seek(BMP_HeaderSize)
 data = f.read(os.path.getsize(output_file))
 f.seek(bufferSize)
 f.write(data)
```

```
f.seek(-(BMP_HeaderSize-bufferSize), os.SEEK_END)

f.truncate()
```

The algorithm looks pretty similar to what we saw the malware using. The main difference here after patching is the data we write to the dumped resource file needed to be trimmed of unnecessary 0xff's that were remained in the stub of the header and at the end of file. That is why in the code snippet above, there was a few additional f.seek(arg1, arg2) in order to point to specific sections of the file and cut the remaining 59 bytes (298b–239b) from both areas in memory.

Conclusion

After running our finalized decrypter program, we can see that the dumped decrypted header looks like the fixed stub:

```
➜ python decrypt_pe_stub_from_resource.py

[!] Searching PE sections for .data

[!] Dumping resource section from AudioSes.dll[!] Replacing first 298 bytes of resource dump.

['4d', '5a', '90', '00', '03', '00', '00', '00', '04', '00', '00', '00', 'ff', 'ff', '00', '00', 'b8', '00', '00', '00', '00', '00', '00', '00', '40', '00', '00', '00', '00', '00', '00', '00', '00', '00', '00', '00', '00', '00', '00', '00', '00', '00', '00', '00', '00', '00', '00', '00', '00', '00', '00', '00', '00', '00', '00', '00', '00', '00', '00', '00', '80', '00', '00', '00', '0e', '1f', 'ba', '0e', '00', 'b4', '09', 'cd', '21', 'b8', '01', '4c', 'cd', '21', '54', '68', '69', '73', '20', '70', '72', '6f', '67', '72', '61', '6d', '20', '63', '61', '6e', '6e', '6f', '74', '20', '62', '65', '20', '72', '75', '6e', '20', '69', '6e', '20', '44', '4f', '53', '20', '6d', '6f', '64', '65', ...][!] Replaced first 239 bytes of file

[!] Finished truncating trailing bytes➜ file extracted.dll && openssl md5 extracted.dll

extracted.dll: PE32+ executable (DLL) (console) x86-64 (stripped to external PDB), for MS Windows

MD5(extracted.dll)= b43128868cb3eadaae228b56b87ff856
```

It looks like we have successfully repaired the malware's embedded resource file. If we search the MD5 hash above in VirusTotal, you'll notice that this is considered a malicious executable by 31 next generation anti-viruses.

Loading the new DLL inside of IDA Pro, we can see everything was disassembled successfully.

At first glance, this malware looks to be a Cobalt Strike Beacon loader file that creates a named pipe for covert communication between application to application. If we take a look at the main detections page from VirusTotal, you'll notice a lot of keyword references to Riskware/Cobalt_Strike and Cobalt_Strike (PUA).

After referencing CobaltStrike[.]com's main page, I was almost certain that this behavior is something pretty well known with stageless payloads. It is said that stageless beacon artifacts always include an executable, a service executable, DLLs, and shellcode that runs the beacon payload.

Cobalt Strike's payload is a reflective DLL with a valid program patched in over the PE header. This program will perform actions such as resolve the memory address where the bootstrap program resides and call the reflective loader as an argument. The DLLMain() function that gets called (as seen above) is what passes control to the beacon and what will eventually make calls to a DNS server.

We can further explore how all this comes together in part 2 of these types of payloads.

Thank you for following along! I hope you enjoyed it as much as I did. If you have any questions on this article or where to find the challenge, please DM me at my Instagram: @hackersclub or Twitter: @ringoware

Happy Hunting :)

P.S. If you missed the link to my decrypter program above, you can find the source code here. It is both python2 and python3 compatible.

ID	Name	Description
T1027.002	Obfuscated Files or Information: Software Packing	The encrypted data from address 0x00402008 is most likely going to be our fixed PE headers, and moving down in the code base, a decryption algorithm is about to load up in the queue. The name of this function will be resolved dynamically which is why we can't see it being defined in the decompiler (just labeled as func). And if we resolve the embedded payload, there is a possibility that the subroutine imported at runtime will show in what might be thought of now as another DLL.
T1140	Deobfuscate/Decode Files or Information	The encrypted data from address 0x00402008 is most likely going to be our fixed PE headers, and moving down in the code base, a decryption algorithm is about to load up in the queue.
T1071.001	Application Layer Protocol: Web Protocols	At first glance, this malware looks to be a Cobalt Strike Beacon loader file that creates a named pipe for covert communication between application to application. The DLLMain() function that gets called (as seen above) is what passes control to the beacon and what will eventually make calls to a DNS server.
T1573	Encrypted Channel	At first glance, this malware looks to be a Cobalt Strike Beacon loader file that creates a named pipe for covert communication between application to application.