Unpacking & Decrypting FlawedAmmyy

Mike Downey
Unpacking & Decrypting FlawedAmmyy

GIAC (GREM) Gold Certification

Author: Mike Downey, mdowney@protonmail.com
Advisor: Rajat Ravinder Varuni
Accepted: 2019/04/22

Abstract

Malware authors commonly utilize packers (Roccia, 2017) as a method of concealing functionality and characteristics of their malicious code, making an analyst’s job more difficult. Second stage executables may also be encrypted, requiring the analyst to gather an understanding of how this code is manipulated. The ability to unpack and decrypt malicious software is a critical step in understanding intent and the scope of malware capabilities. The goal of this paper is to provide real-world application of the unpacking and decoding techniques required to analyze a remote access Trojan (RAT) known as FlawedAmmyy. While basic static and dynamic analysis will not be covered, this paper will focus on the step-by-step procedures to unpack and decrypt a FlawedAmmyy sample within a debugger.
1. Introduction

FlawedAmmyy is a remote access Trojan (RAT) whose code is based on the remote desktop software known as Ammyy Admin. Although recently discovered in 2018, Proofpoint researchers state that the Trojan has been utilized since the beginning of 2016 (Proofpoint, 2018). The sample analyzed in this paper was observed in an email campaign delivered by the Necurs botnet (O’Neill & Harz, 2018) in mid-2018 containing a .iqy attachment (My Online Security, 2018). IQY (internet query) is a filetype utilized by Microsoft Excel and in this case, was used to retrieve a series of scripts and ultimately a malicious executable. As shown later in this paper, the executable is a packed downloader whose purpose includes obtaining and decoding the FlawedAmmyy RAT. This sample is received initially as a .png file; however, the data is an encrypted binary that the downloader will decrypt before executing. Figure 1 displays a high-level overview of this campaign.

Figure 1: Flow of the FlawedAmmyy campaign

Through this paper, the reader will be walked through step-by-step procedures for unpacking the downloader and decoding the RAT. The goal is to provide the reader with a real-world example of unpacking and decoding that could be utilized to perform

Mike Downey mdowney@protonmail.com
analysis on other pieces of malicious software. Through gaining experience and an understanding of how malicious software conceals itself, the reader can identify these techniques during the investigation of different samples in the wild.

The target audience is malware analysts and security operations center (SOC) analysts with a focus on analyzing malicious software. It is expected that readers have a basic understanding of portable executables (PE), the Windows API, x86 assembly, and debugging. These topics will not be discussed in detail.

2. VM Setup

Multiple tools can be utilized for accomplishing similar purposes and are usually left to user preference. The following list identifies the operating systems and applications used for analysis within this paper.

- Hypervisor
  - VMware Workstation
- Operating Systems
  - Microsoft Windows 10
  - REMnux
- Tools
  - x32dbg
  - HxD
  - INetSim
  - FakeDNS
- Malicious samples
  - Packed downloader (MD5: 28eae907ea38b050cbdce82bb623c00a)
  - Unpacked downloader (MD5: d920413442adde78394077c2bde093d8)
  - Encrypted FlawedAmmyy (MD5: 651df83a583b0eefd0e3d39c4f020f11)
  - Decrypted FlawedAmmyy (MD5: 7920daed2c352229e479171ee0b29457)

Mike Downey mdowney@protonmail.com
3. Unpacking the Downloader

The first stage executable that will be analyzed is simply a packed downloader for FlawedAmmyy. Although the downloader performs multiple tasks, this section will focus on the unpacking process, while section 4 will focus on the retrieval and decoding of the FlawedAmmyy sample. As this downloader is packed, static analysis would not reveal all indicators and functionality. To retrieve relevant information such as strings and imports, the analyst will need first to identify how to unpack the downloader. Working with an unpacked sample can also prove to be more efficient, as the unpacking code will no longer be present.

3.1 Decode & Write Instructions

Before reaching the instructions responsible for the unpacking process, the analyst must deal with the sample decoding and writing new instructions, as well as those instructions being moved to different memory locations. These actions can interfere with the analyst’s ability to set breakpoints, and some breakpoints may never be reached.

A call to HeapCreate (Microsoft, HeapCreate function, 2018) is found at 0x00407204 (Figure 2) to allocate space for the soon-to-be decoded instructions. The size and options for the new memory location can be observed by viewing the stack before the call is made (Figure 3): flOptions=00040000 (HEAP_CREATE_ENABLE_EXECUTE), dwInitialSize=688, and dwMaximumSize=688.

Mike Downey mdowney@protonmail.com
HeapCreate will return (via the EAX register) a handle to the newly allocated memory location (0x02450000 in this instance). By utilizing the “Follow in Dump” feature of x32dbg to monitor this memory location, modifications to the bytes in this location can be observed.

Soon after the call to HeapCreate is the loop responsible for decoding and writing the unpacking code (Figure 2). The EBX register in this instance will act as a counter and is compared to 688 (the size of the heap). At 0x00407231 the value at EAX is moved into our newly allocated memory (now referenced by esi). This instruction is the point where bytes will begin to change in the monitored memory location.

After the loop is a call to the new memory location (ESI) which contains the newly written instructions. The packer just allocated new memory on the heap, decoded additional instructions, wrote the instructions to the new location, and is now jumping to the newly written instructions.

Within the newly written instructions is a call to VirtualAlloc (Microsoft, VirtualAlloc function, 2018) with the same size observed previously, 688 (Figure 4). In this instance, the newly allocated space starts at 0x00500000.
Figure 4: Allocation of new space and transfer of data

Figure 4 depicts the instruction “rep movsb” (Aldeid, movsb, 2015) that moves bytes from memory referenced by ESI (the decoded instructions that we are now executing) to memory referenced by EDI (memory location created by VirtualAlloc), followed by a return.

The stack can be utilized to view where the return instruction will direct code execution (Figure 5).

![Figure 5: Stack showing the return location (0050007E)](image)

On top of the stack is 0x0050007E. This memory location is where execution will be directed when the return is taken. The offset of the next instruction we see after the return in Figure 4 is 7E (0x0245007E). Note that the address of the return in figure 5 is also at offset 7E of the newly allocated space. The instructions being executed were merely copied to a new location, and execution redirected to the next logical instruction within that new location. Because of these actions, analysis becomes more difficult by removing the ability to set breakpoints on specific addresses.

### 3.2 Unpacking

After successfully dealing with the downloader writing and moving instructions, the analyst can finally begin analyzing the unpacking process. Figure 6 shows another call to VirtualAlloc at 0x005005CC. Once again, the newly allocated memory (returned by EAX) should be monitored to identify what will be written to this space. Unlike the previous call to VirtualAlloc, the newly allocated space will eventually hold the unpacked executable.

Mike Downey mdowney@protonmail.com
At 0x0050009C is the instruction “rep movsb” (Figure 7). This instruction will move 126,464 bytes (ECX = 0001EE00) from the location referenced by ESI to the location referenced by EDI (Figure 8). By viewing the “Memory Map” tab of x32dbg, it can be determined that the address 0x0040B0C0 (source of the move) is within the .rsrc section (Microsoft, PE Format, 2019) (Figure 9). This realization is essential as we now know that the packed executable is being stored in the .rsrc section. This new understanding will come in handy in section 3.2.1 by allowing the analyst to fast-track the unpacking process.

Viewing the dump of this memory space shows that data has been written (Figure 10). As this is still the packed executable, it does not appear legible at this time.
Once the data has been moved, the sample will proceed to the function responsible for unpacking the encoded executable (Figure 11). As part of the unpacking process, the value of EAX is XOR’d with the bytes “964132AC”.

At this time, the FlawedAmmyy downloader is unpacked into the memory region that was being monitored. This memory region is now ready to be dumped to disk. The MD5 hash of the unpacked downloader is: d920413442adde78394077c2bde093d8.

To understand how the code would be executed, additional analysis is necessary. Figure 13 shows a call to VirtualProtect (Microsoft, VirtualProtect function, 2018) that is
changing the access protection on the beginning virtual address of the sample (0x00400000) to allow the address space to be written. Shortly after is another “rep movsb” instruction overwriting address space within the current sample being debugged (EDI) with a portion of the now unpacked executable (ESI). Figure 14 shows the register values before executing the “rep movsb” instruction. Effectively, the packer will overwrite the memory of the current process with the unpacked code, a section at a time.

![Figure 13: VirtualProtect call with PAGE_READWRITE and movsb to rewrite memory of the current process with the unpacked code](image)

![Figure 14: Registers showing the source (ESI) and destination (EDI) of movsb](image)

### 3.2.1 Fast-Tracking the Unpacking Process

With the information gathered during the unpacking process, the analyst can now unpack similar samples more efficiently. Since it’s now clear that the packed executable is stored in the .rsrc section, this will be the starting point. With the executable loaded into x32dbg and paused at the entry point, the first step is to set an access memory
breakpoint (Singleshoot) on the .rsrch memory region (Figure 15).

<table>
<thead>
<tr>
<th>Address</th>
<th>Size</th>
<th>Info</th>
</tr>
</thead>
<tbody>
<tr>
<td>00400000</td>
<td>00010000</td>
<td>dr.exe</td>
</tr>
<tr>
<td>00401000</td>
<td>00070000</td>
<td>&quot;text&quot;</td>
</tr>
<tr>
<td>00402000</td>
<td>00020000</td>
<td>&quot;.data&quot;</td>
</tr>
<tr>
<td>00403000</td>
<td>00030000</td>
<td>&quot;rsrch&quot;</td>
</tr>
<tr>
<td>00404000</td>
<td>00040000</td>
<td>Reserved</td>
</tr>
<tr>
<td>00405000</td>
<td>00006000</td>
<td>Reserved</td>
</tr>
<tr>
<td>00406000</td>
<td>0000A000</td>
<td>Reserved</td>
</tr>
<tr>
<td>00407000</td>
<td>0001A000</td>
<td>Reserved</td>
</tr>
<tr>
<td>00408000</td>
<td>000E6000</td>
<td>Reserved</td>
</tr>
<tr>
<td>00580000</td>
<td>000C5000</td>
<td>Device\HarddiskVolume4\windows\System32\locale.nls</td>
</tr>
<tr>
<td>00680000</td>
<td>000D0000</td>
<td>Reserved</td>
</tr>
<tr>
<td>00770000</td>
<td>00003000</td>
<td>Thread 15B4 Stack</td>
</tr>
</tbody>
</table>

Figure 15: Data moved from an address within the .rsrch section

When the executable is run, the debugger will pause at the “rep movsb” instruction (Figure 16) as this is the first instruction that accesses the specified memory location.

Figure 16: Data moved from an address within the .rsrch section

Viewing the registers (Figure 17), ESI should reference an address within the .rsrch section, and EDI will reference the memory location of where the data will be moved to (0x00440000 in this instance). Make a note of this location and monitor it.

Figure 17: Registers showing the source (ESI) and destination (EDI) of movsb

Running to the next instruction (pop esi) will complete the move. With the data moved, the next step is to set another access memory breakpoint on the memory location

Mike Downey mdowney@protonmail.com
previously referenced by EDI (Figure 18).

Figure 18: Memory map showing the location of the next breakpoint

When run again, the debugger will stop in the middle of the unpacking loop when the memory region being monitored is accessed. The debugger will pause at the “lodsb” (Aldeid, lodsb, 2015) instruction within the loop (Figure 19). Running to the instruction after the loop (“popad”) will ensure the loop completes. Viewing the monitored memory location confirms that the process is complete (Figure 20) and the unpacked executable can be dumped from memory.

Figure 19: Unpacking loop

Mike Downey mdowney@protonmail.com
4. Retrieving and Decrypting FlawedAmmyy

With the sample now successfully unpacked, the analyst can continue analysis to determine the capabilities and functionality of the sample. Performing static analysis with tools such as “strings” will now identify items of interest such as a URL and additional functions. The analysis performed in this section will utilize the unpacked sample previously retrieved in section 3. Debugging the unpacked downloader will identify where the FlawedAmmyy RAT is retrieved and how the sample is decrypted.

4.1 Retrieve the Encrypted FlawedAmmyy

As the downloader is responsible for retrieving the RAT, the analyst requires a method of providing the sample during debugging. One solution is to configure inetsim to provide the sample upon request. The encrypted FlawedAmmyy sample that will be retrieved (MD5: 651df83a583b0eefd0e3d39c4f020f11) should be stored in /var/lib/inetsim/http/fakefiles, and the inetsim configuration file (/etc/inetsim/inetsim.conf) should be edited to include a new http_static_fakefile (Figure 21).

Mike Downey mdowney@protonmail.com
Figure 21: Inetsim configuration concerning the malicious load.png

Figure 22 displays a function that appears as likely interesting due to the file path and URL passed as arguments. We can theorize that this function will be responsible for retrieving and saving a file.

Within this function is a call to InternetOpenA (Microsoft, InternetOpenA function, 2018) and InternetOpenUrlA (Microsoft, InternetOpenUrlA function, 2018), with the URL provided as an argument (Figure 23).

Shortly after, the file “C:\ProgramData\Settings\wsus_41a480” is created (Figure 24) via a call to CreateFileA (Microsoft, CreateFileA function, 2018) to prepare for the storage of what is soon to be retrieved from the observed URL.

Mike Downey mdowney@protonmail.com
Figure 24: Create wsus_41s480.tmp

Figure 25 displays a call to InternetReadFile (Microsoft, InternetReadFile, function 2018) to read data from the handle opened by InternetOpenUrl. WriteFile (Microsoft, WriteFile function, 2018) is called afterward to complete this process by writing the data to wsus_41a480.tmp.

Figure 25: Write malicious file

Viewing the newly written data in a hex editor (HxD in this instance), will reveal that this file does not contain a recognizable header (Figure 26). Additional analysis is necessary to identify how this sample is decrypted.

Figure 26: Encrypted File
4.2 Decrypt FlawedAmmyy

Figure 27 depicts the two functions responsible for decrypting FlawedAmmyy. The first function is responsible for setting up a buffer to be utilized in the decryption process. Before the “setup buffer” call is taken, the stack (Figure 28) shows three arguments being passed: a key utilized to modify the buffer, the key size, and the location of the buffer.

Within the first function is a loop (Figure 29) at 0x004041B0 - 0x004041B9 that modifies each byte of the buffer to contain the bytes 00 - FF (Figure 30). This instruction set is followed by a second loop at 0x004041C4 - 0x004041F8 that utilizes the key “qfewjq4ewqf32df43dFD2h5” to modify the buffer further. Figure 31 displays the buffer after completion of this loop. Understanding each step in this loop is not necessary for gaining a high-level view of the loader’s activity. However, this function can be replicated in Python, as shown in section 4.2.1.
Figure 29: Initialization and modification of the buffer

<table>
<thead>
<tr>
<th>Address</th>
<th>Hex</th>
<th>ASCII</th>
</tr>
</thead>
<tbody>
<tr>
<td>0019F190</td>
<td>00</td>
<td>01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F</td>
</tr>
<tr>
<td>0019F1A0</td>
<td>10</td>
<td>11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F</td>
</tr>
<tr>
<td>0019F1B0</td>
<td>20</td>
<td>21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E 2F</td>
</tr>
<tr>
<td>0019F1C0</td>
<td>30</td>
<td>31 32 33 34 35 36 37 38 39 3A 3B 3C 3D 3E 3F</td>
</tr>
<tr>
<td>0019F1D0</td>
<td>40</td>
<td>41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F</td>
</tr>
<tr>
<td>0019F1E0</td>
<td>50</td>
<td>51 52 53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F</td>
</tr>
<tr>
<td>0019F1F0</td>
<td>60</td>
<td>61 62 63 64 65 66 67 68 69 6A 6B 6C 6D 6E 6F</td>
</tr>
<tr>
<td>0019F200</td>
<td>70</td>
<td>71 72 73 74 75 76 77 78 79 7A 7B 7C 7D 7E 7F</td>
</tr>
<tr>
<td>0019F210</td>
<td>80</td>
<td>81 82 83 84 85 86 87 88 89 8A 8B 8C 8D 8E 8F</td>
</tr>
<tr>
<td>0019F220</td>
<td>90</td>
<td>91 92 93 94 95 96 97 98 99 9A 9B 9C 9D 9E 9F</td>
</tr>
<tr>
<td>0019F230</td>
<td>A0</td>
<td>A1 A2 A3 A4 A5 A6 A7 A8 A9 AA AB AC AD AE AF</td>
</tr>
<tr>
<td>0019F240</td>
<td>B0</td>
<td>B1 B2 B3 B4 B5 B6 B7 B8 B9 BA BB BC BD BE BF</td>
</tr>
<tr>
<td>0019F250</td>
<td>C0</td>
<td>C1 C2 C3 C4 C5 C6 C7 C8 C9 CA CB CC CD CE CF</td>
</tr>
<tr>
<td>0019F260</td>
<td>D0</td>
<td>D1 D2 D3 D4 D5 D6 D7 D8 D9 DA DB DC DD DE DF</td>
</tr>
<tr>
<td>0019F270</td>
<td>E0</td>
<td>E1 E2 E3 E4 E5 E6 E7 E8 E9 EA EB EC ED EF EE</td>
</tr>
<tr>
<td>0019F280</td>
<td>F0</td>
<td>F1 F2 F3 F4 F5 F6 F7 F8 F9 FA FB FC FD FE FF</td>
</tr>
</tbody>
</table>

Figure 30: Buffer initialized to 00 - FF

<table>
<thead>
<tr>
<th>Address</th>
<th>Hex</th>
<th>ASCII</th>
</tr>
</thead>
<tbody>
<tr>
<td>0019F190</td>
<td>AE</td>
<td>A8 F2 B9</td>
</tr>
<tr>
<td>0019F1A0</td>
<td>B9</td>
<td>AD 9D 89 43 11 2A 5A EA 46 0E 99 50</td>
</tr>
<tr>
<td>0019F1B0</td>
<td>04</td>
<td>05 29 39 2F DA 5F EC 5C A5 56</td>
</tr>
<tr>
<td>0019F1C0</td>
<td>10</td>
<td>11 26 15 17 14 19 1A 1B 1C 1D 1E 1F</td>
</tr>
<tr>
<td>0019F1D0</td>
<td>20</td>
<td>21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E 2F</td>
</tr>
<tr>
<td>0019F1E0</td>
<td>30</td>
<td>31 32 33 34 35 36 37 38 39 3A 3B 3C 3D 3E 3F</td>
</tr>
<tr>
<td>0019F1F0</td>
<td>40</td>
<td>41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F</td>
</tr>
<tr>
<td>0019F200</td>
<td>50</td>
<td>51 52 53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F</td>
</tr>
<tr>
<td>0019F210</td>
<td>60</td>
<td>61 62 63 64 65 66 67 68 69 6A 6B 6C 6D 6E 6F</td>
</tr>
<tr>
<td>0019F220</td>
<td>70</td>
<td>71 72 73 74 75 76 77 78 79 7A 7B 7C 7D 7E 7F</td>
</tr>
<tr>
<td>0019F230</td>
<td>80</td>
<td>81 82 83 84 85 86 87 88 89 8A 8B 8C 8D 8E 8F</td>
</tr>
<tr>
<td>0019F240</td>
<td>90</td>
<td>91 92 93 94 95 96 97 98 99 9A 9B 9C 9D 9E 9F</td>
</tr>
<tr>
<td>0019F250</td>
<td>A0</td>
<td>A1 A2 A3 A4 A5 A6 A7 A8 A9 AA AB AC AD AE AF</td>
</tr>
<tr>
<td>0019F260</td>
<td>B0</td>
<td>B1 B2 B3 B4 B5 B6 B7 B8 B9 BA BB BC BD BE BF</td>
</tr>
<tr>
<td>0019F270</td>
<td>C0</td>
<td>C1 C2 C3 C4 C5 C6 C7 C8 C9 CA CB CC CD CE CF</td>
</tr>
<tr>
<td>0019F280</td>
<td>D0</td>
<td>D1 D2 D3 D4 D5 D6 D7 D8 D9 DA DB DC DD DE DF</td>
</tr>
</tbody>
</table>

Mike Downey mdowney@protonmail.com
After the buffer setup, the second function displayed in Figure 27 will be taken.

Viewing the stack (Figure 32) before the call shows the three arguments being passed: the location of the encrypted FlawedAmmyy sample, the size of the sample, and the location of the buffer modified in the previous function.

Within this function is a loop that will utilize the previous buffer to decrypt the sample (Figure 33).

Figure 33: FlawedAmmyy decryption loop
As the loop is stepped through, changes will be made to the address space being monitored. A Portable Executable (PE) file header will be shown, as well as the string “This program cannot…” (Figure 34). At this point, it can be determined that this is the last function before the sample is fully decrypted.

Figure 34: Decrypted FlawedAmmyy executable

Once the loop completes, the decrypted sample can be dumped from x32dbg, providing the FlawedAmmyy sample. To see how this new code is executed, the downloader will be further analyzed.

Figure 35 displays the file “C:\ProgramData\Settings\wsus.exe” being created, and a call to WriteFile. The decrypted data shown in Figure 34, now referenced by “ebp+8” will be written to wsus.exe.

Figure 35: Write file

Afterward, the .tmp file containing the encrypted executable is then removed with a call to DeleteFileA (Microsoft, DeleteFileA function, 2018). The downloader will verify the decryption process by checking the first two bytes of the buffer (the address contained in EAX) to ensure a proper PE file header of MZ (Figure 36).
The downloader will check if the user is an admin and create a new service for persistence (Mitre, n.d.) by utilizing ShellExecute (Microsoft, ShellExecuteA function, 2018) (Figure 37) with the command “cmd /C sc create foundation binPath=\"C:\ProgramData\Settings\wsus.exe -service\" type= own start= auto error= ignore displayname=\"Foundation\"”. This instruction is followed by another call to ShellExecute to start the service with the command “cmd /C net.exe start foundation y.”

Once the service has been started, the downloader will utilize GetModuleFileName (Microsoft, GetModuleFileNameA function, 2018) and GetShortPathName (Microsoft, GetShortPathNameW function, 2018) to retrieve the path for the downloader, then call ShellExecute (Figure 38) to remove the file with the command “cmd.exe /C del path_to_file.” The downloader process then ends with a call to ExitProcess.
4.2.1 Recreating the Decryption Functions in Python

As a challenge, I’ve recreated the decryption function in python. This script has been tested against multiple samples and proven successful. Utilizing the script requires that the user has obtained the unpacked downloader and the encrypted FlawedAmmyy sample. As the key is constant in the downloader, I was able to identify a method to extract the key. The python script will attempt to identify the key in the unpacked downloader by searching for strings within a range of bytes in proximity to the URL string (Figure 39).

```python
def extract_keys(loader):
    contents = loader.read()

    # key typically appears near the callout string
    # obtain offset of the beginning of the callout string
    offset = contents.find("http")

    # extract data within 1000 bytes before and after the offset of the callout string
    minimum = offset - 1000
    maximum = offset + 1000
    key_range = contents[minimum:maximum]

    # regex to extract potential keys
    expression = re.compile('\x00\x00(?P<key>\w{3,})\x00+')

    # obtain all matches within the defined range
    keys = expression.findall(key_range)
    print("Potential keys discovered:")
    print(keys)
    return keys
```

*Figure 39: ammyy-decryptor.py - extract potential keys*

Each string discovered in this range is utilized to modify the buffer and attempt decryption, then the file header checked to ensure success (Figure 40).
Figure 40: ammyy-decryptor.py script - utilize each extracted key and check the file header for success

The output of the script can be observed in Figure 41, displaying potential keys and successful decryption.

Figure 41: Output of ammyy-decryptor.py

Recreating the decryption functions was performed by walking through each instruction in the disassembler, recreating the logic in Python, then checking the values of the variables alongside the corresponding registers. The main library utilized was binascii, specifically the hexlify function to convert between the binary and hexadecimal representations. Most of the byte manipulation was performed on the base-10 equivalent, but the same could have been accomplished by performing binary arithmetic. Although this was somewhat of a tedious process, it proved to be a great excuse to brush off some scripting skills and apply them to a new area of focus. The full Python script can be found at the following Github repo: https://github.com/mal-labs/ammyy-decryptor

Mike Downey mdowney@protonmail.com
5. Conclusion

Continuously analyzing new samples allows analysts to sharpen their skill set and identify unique characteristics that they’ve yet to experience. This paper detailed the practical utilization of the skills obtained via the GREM course and applied those skills toward analyzing FlawedAmmy. Before examining the unpacking process, techniques were identified that could prove to complicate analysis, including the unpacker writing new code during runtime. Unpacking the FlawedAmmyy downloader allowed for determining how the next stage executable is retrieved. As observed in section 4, analysis of the RAT required the retrieved sample to be decrypted, utilizing the decryption functions within the downloader. Understanding the techniques found within the FlawedAmmyy downloader could allow analysts to analyze similar samples they discovered throughout their career.

References


Mike Downey mdowney@protonmail.com


Mike Downey mdowney@protonmail.com


# Upcoming SANS Training

Click here to view a list of all SANS Courses

<table>
<thead>
<tr>
<th>SANS OnDemand</th>
<th>OnlineUS</th>
<th>Anytime</th>
<th>Self Paced</th>
</tr>
</thead>
<tbody>
<tr>
<td>SANS SelfStudy</td>
<td>Books &amp; MP3s OnlyUS</td>
<td>Anytime</td>
<td>Self Paced</td>
</tr>
</tbody>
</table>