In this report, I will demonstrate the methods to unpack a SmokeLoader executable manually. While this can be automated with UnpacMe tool, I still found it helpful to unpack manually to understand what is happening. This analysis will not be going over the actual SmokeLoader payload ,instead how I went about extracting it for future reference and technique practice.

Smokeloader Summary

SmokeLoader is a bot that has been used to distribute malware such as Vidar. SmokeLoader has also been around since 2011, one of the samples I have seen when browsing malware repository sites such as MalwareBazar.

Stage 1 Analysis:

Sha256SUM	0098901d9b40d0d1e34f820347bd7af6d3 9da582115aab1918a5a71dd03bf7b2

Tactics and Techniques

Junk Code and Anti Anti-Sandbox

The first stage of Smokeloader has a smokescreen, plagued by a large amount of garbage API calls, all of which do nothing to progress the code. While running the program through a debugger, I noticed it took a significant amount of time to get to my first breakpoint. In addition to slowing down the static analysis work on this sample, the time it would take to carry out the intended purpose of the code could be used as an anti-sandbox technique which is comparable to having a large sleep time.

```
do
{
  if ( v4 + uBytes == 94 )
  {
    ReplaceFileW(0, 0, 0, 0, 0, 0);
    GlobalAddAtomA(0);
    HeapSize(0, 0, 0);
  }
  ++v4;
}
while ( v4 < (int)&loc_40C892 + 1 );
sub_4011CF(v2);
do
{
  if ( v0 == 30238 )
    mw_junk();
  ++v0;
}
while ( v0 < 2386641 );
dword_80C128 = (int)ALLOCATED_BYTES;
```

```
v0 = 0;       int v13; // [esp+154h] [ebp-4h]
if ( uBytes == 522 )
{
  FileTimeToSystemTime(&FileTime, 0);
  TerminateProcess(0, 0);
  ResetEvent(0);
  GetConsoleAliasesW(0, 0, 0);
  GetVersionExW(&VersionInformation);
  ResetWriteWatch(0, 0);
  SetComputerNameExW(ComputerNameNetBIOS, &Buffer);
  v10 = 7;
  v9 = 0;
  v8 = 0;
  sub_401808(&v7, 6);
  v13 = 0;
  realloc(0, 0);
  sin(0.0);
  floor(0.0);
  atan(0.0);
  v13 = -1;
  sub_4017C4(1, 0);
}
```

Decryption

Amidst this junk code, is the decryption routine which is shellcode that is later executed.

Stage 2 – Shellcode Analysis

The analysis of this shellcode can be done in two separate ways. The first and least preferred method is loading up the shellcode as an additional binary within IDA pro. Using this method as a standard practice undercuts the advantage of the Windows types that IDA automatically provides for the original stage 1 payload. While it is an easier method, I found that looking for the entry point and scrolling through with this method was difficult and tedious. The more preferred method is converting the shellcode into an executable and then loading it into IDA pro.

Tactics and Techniques:

API Hashing and API Resolution

The Shellcode uses API-Hashing and LDR structure to load and search for Windows APIs to resolve. The two APIs used are LoadLibraryA and GetProcAddress, which are then being used to resolve more APIs dynamically. The API hashing algorithm can be found here on my GitHub.

```
int __stdcall mw_api_resolution(int LIBRARY_HASH, int FUNCTION_HASH)
{
  struct _LIST_ENTRY *i; // ecx
  struct _LIST_ENTRY *v3; // edx
  int v4; // ecx
  int v5; // ebx
  _DWORD *v6; // edx
  unsigned __int16 *v7; // ebx
  int v8; // edx
  int v9; // ecx
  int v11; // [esp-8h] [ebp-14h]

  for ( i = NtCurrentPeb()->Ldr->InLoadOrderModuleList.Flink; mw_hash_algo(i[6].Flink, LIBRARY_HASH, 2); i = v3 )
    ;
  v11 = *(_DWORD *)(v4 + 24);
  v5 = v11 + *(_DWORD *)(*(_DWORD *)(v11 + 60) + v11 + 120);
  v6 = (_DWORD *)(v11 + *(_DWORD *)(v5 + 32));
  v7 = (unsigned __int16 *)(v11 + *(_DWORD *)(v5 + 36));
  while ( mw_hash_algo((_BYTE *)(v11 + *v6), FUNCTION_HASH, 1) )
  {
    v6 = (_DWORD *)(v8 + 4);
    ++v7;
  }
  return *(_DWORD *)(4 * *v7 + v9) + v11;
}
```

Next, the shellcode creates a structure to store all the dynamically resolved APIs across the program.

```
int __cdecl mw_api_resolution_function(struct_a1 *a1)
{
  int result; // eax
  int GetProcAddress; // [esp+10h] [ebp-34h]
  _WORD v3[16]; // [esp+14h] [ebp-30h] BYREF
  int v4; // [esp+34h] [ebp-10h]
  int KernelHandle; // [esp+38h] [ebp-Ch]
  int LoadLibraryA; // [esp+3Ch] [ebp-8h]
  int v7; // [esp+40h] [ebp-4h]

  a1->start_value_zero = 0;
  v7 = 0;
  v4 = 0x401805;
  a1->payload = (struct_possible_start_address *)byte_401805;
  a1->possible_end_address = v4 + 61;
  LoadLibraryA = mw_api_resolution(872072, 874374);
  GetProcAddress = mw_api_resolution(872072, 3443706);
  a1->LoadLibraryA = LoadLibraryA;
  a1->GetProcAddress = GetProcAddress;
  KernelHandle = 0;
  strcpy((char *)v3, "kernel32.dll");
  KernelHandle = ((int (__stdcall *)(_WORD *))a1->LoadLibraryA)(v3);
  strcpy((char *)v3, "GlobalAlloc");
  LOBYTE(v3[6]) = 0;
  a1->GlobalAlloc = ((int (__stdcall *)(int, _WORD *))a1->GetProcAddress)(KernelHandle, v3);
  strcpy((char *)v3, "GetLastError");
  a1->GetLastError = ((int (__stdcall *)(int, _WORD *))a1->GetProcAddress)(KernelHandle, v3);
  strcpy((char *)v3, "Sleep");
  v3[3] = 0;
  LOBYTE(v3[4]) = 0;
  a1->Sleep = ((int (__stdcall *)(int, _WORD *))a1->GetProcAddress)(KernelHandle, v3);
  strcpy((char *)v3, "VirtualAlloc");
  a1->VirtualAlloc = ((int (__stdcall *)(int, _WORD *))a1->GetProcAddress)(KernelHandle, v3);
0000022E mw_api_resolution_function:11 (40102E) (Synchronized with IDA View-A)
```

Decryption

The shellcode uses the same decryption routine to decrypt the next stage;like the second stage, this is also shellcode.

Stage 3 Analysis

The third stage was incredibly hard to analyze statically and it proved to be more efficient to set breakpoints on VirtualAlloc and see what was being stored in the buffer. Finally, what came out from this was the final payload, a MZ executable which I have identified as SmokeLoader.

Summary:

The usage of junk code and API Hashing, along with using the LDR structure to load modules, are techniques I see more often in my analysis. While I took the time to analyze this sample statically, the unpacking process for this sample is straightforward, and putting breakpoints on the VirtualAlloc function to see what is written to their buffers is sufficient. This is a note to point out in the following analysis of a variant of this type for faster unpacking.

What's been learned from this analysis are better techniques for recognizing API hashing and methods of analyzing shellcode effectively and promptly.

I also want to say that I found this analysis particularly rewarding because of the hashing algorithm I found within the code. It was a variation of a rotate right 13 algorithm in addition to adding 10 at the end of it. As of reading this, I have submitted this hashing algorithm to hashdb to be used in future use. This was my first contribution to the cybersecurity industry, something I am thrilled to have done.