securitykitten.github.io/_posts/2014-11-26-getmypass-point-of-sale-malware.md

github.com/malware-kitten/securitykitten.github.io/blob/master/_posts/2014-11-26-getmypass-point-of-sale-malware.md

malware-kitten

malware-kitten/
securitykitten.github.io

Jekyll theme inspired by Swiss design

0 Contributors | 0 Issues | 0 Stars | 0 Forks

layout	title	date
category-post	Getmypass Point of Sale Malware	2014-11-26 13:00:33 -0500

Introduction

While doing some digging recently on VirusTotal I had a rule trigger on what appears to be a new POS malware family.

The MD5 (1d8fd13c890060464019c0f07b928b1a) is the malware that I will be dissecting in this post.

The first interesting thing that struck my eye is the incredibly low detection rate which at the time of this writing was 0/55.

SHA256:	6bffe5385dd1321fe5b99dec3f8858be9ff99c8629c1c8d6f414eebaa663a710
File name:	file09.exe
Detection ratio:	0 / 55
Analysis date:	2014-11-26 01:19:32 UTC (16 hours, 44 minutes ago)

Secondly (and what may be affecting detection) is that the binary is signed from “Bargaining active” which is currently a valid certificate.

Publisher	Bargaining active
Signature verification	Signed file, verified signature
Signing date	1:28 PM 11/9/2014
Signers	[+] Bargaining active

Status	Valid
Valid from	1:00 AM 8/5/2014
Valid to	12:59 AM 8/6/2015
Valid usage	Code Signing
Algorithm	SHA1
Thumbrint	4B49E7698615732941AD4789FBACB989B639E301
Serial number	2C 75 BA 23 12 ED BD 2E 6A 5A 5A FF 77 48 F1 0C

So digging into the code a bit, this malware appears to do something in common with other POS RAM scrapers.

- Process Dumping
- Searching for CC data
- Validation using Luhn's algorithm
- Writing that to a file
- Encrypting / Encoding file

There doesn't appear to be any C2 functionality in this particular piece of malware so this is more of a utility than a backdoor. This malware also does not contain code to do any of the following:

- Lateral movement
- Credential harvesting
- Pushing the harvested data to a non-local file
- Keylogging

Analysis

The malware will first search for an ini file named 1.ini in the same directory as the malware. Without the ini file the malware will exit. Thanks to Josh Grunzweig for pointing out the ini format.

```
[settings]
proc=notepad.exe
time=1000
cryp=1
```

The cryp argument is responsible for toggling on/off functionality to encrypt the collected CC data with RC4.

The malware will also create a mutex when running "1yn8RQLkm8"

```
push    ebp
mov     ebp, esp
push    ecx
push    offset a1yn8rqlkm8 ; "1yn8RQLkm8"
push    1
push    0
call    ds:CreateMutexW
test    eax, eax
jz      short loc_402EE7
```

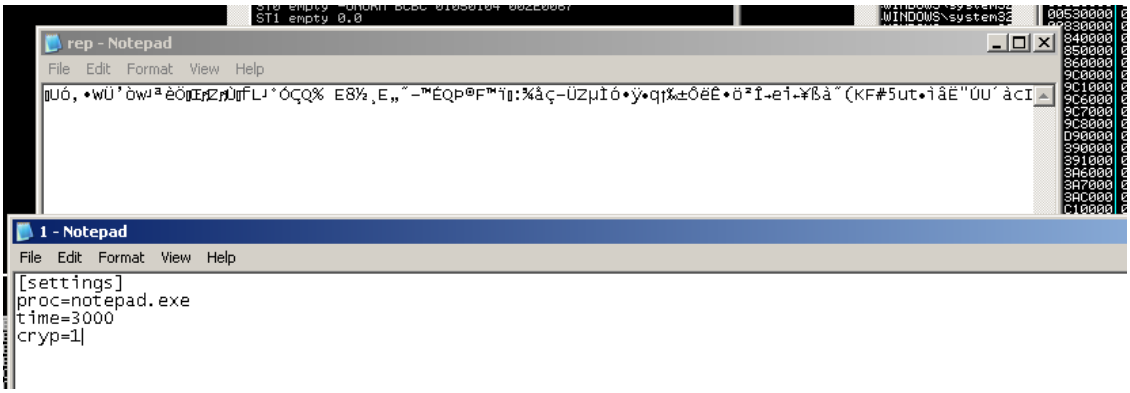
Diving in head first, the first function that stuck out to me is at loc 0x402360. This function is responsible for iterating over processes, calling OpenProcess, and then ultimately ReadProcessMemory.

Very (very!) rough logic for this would look resemble:

{% highlight ruby %} {% raw %} procs = CreateToolhelp32Snapshot Process32FirstW(procs) do OpenProcess while true if VirtualQueryEx ReadProcessMemory else break CloseHandle while Process32NextW CloseHandle VirtualFree {% endraw %} {% endhighlight %}

Many of the POS ram scrapers will use this same sort of functionality to crawl and enumerate processes. One difference is that this malware does use a whitelist (in the ini file) and only dumps processes the user would specify.

Below is a screenshot of the configuration file 1.ini and the encrypted track1 and track2 CC information:



When running the malware in a debugger, I posted sample track data into notepad and stepped though execution. The malware will locate the notepad process (using the above loop) and then pass those results to a function to search for strings that look like track data. These are then parsed and the results are passed to a function that will use the Luhn's algorithm to process and check for valid numbers. A lookup table is used rather than calculating a digital root. This is the same version of the algorithm used in FrameworkPOS and Dexter.

{% highlight c %} {% raw %} v5 = 0; v6 = 2; v7 = 4; v8 = 6; v9 = 8; v10 = 1; v11 = 3; v12 = 5; v13 = 7; v14 = 9; v16 = 1; v15 = 0; v17 = a2; while (1) { v2 = v17--; if (!v2) break; if (v16) v4 = *(_WORD *)(a1 + 2 * v17) - 48; else v4 = *(&v5 + *(_WORD *)(a1 + 2 * v17) - 48); v15 += v4; v16 = v16 == 0; } return v15 % 10 == 0; } {% endraw %} {% endhighlight %}

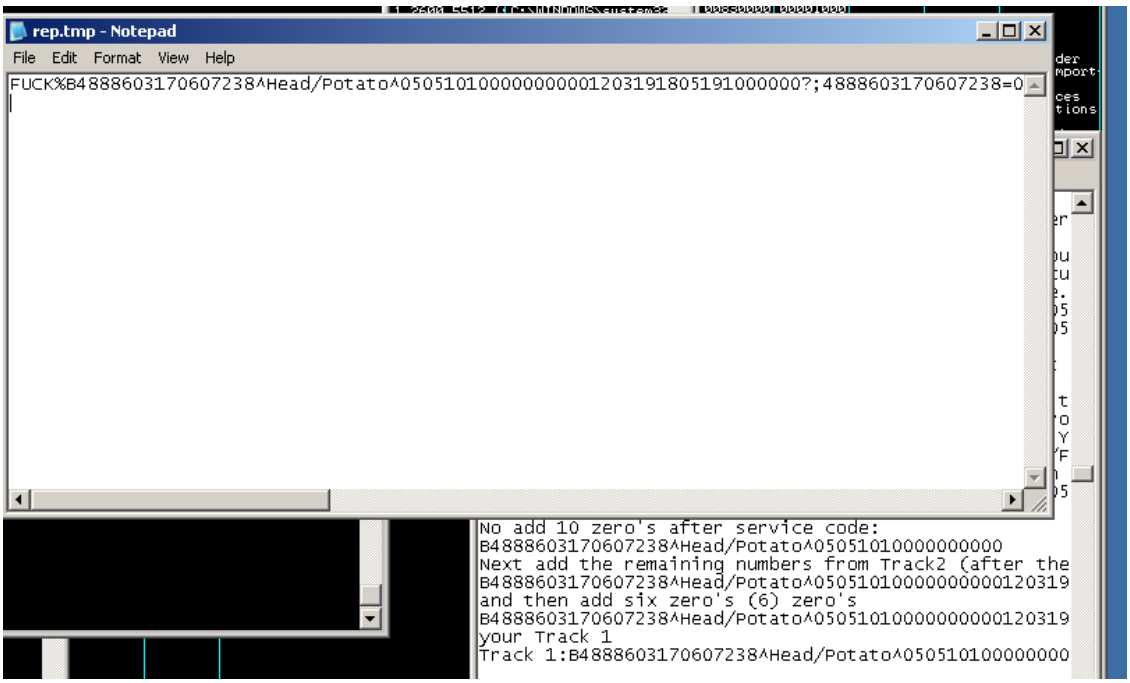
Which in source code would look more like this

{% highlight c %} {% raw %} 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; } {% endraw %} {% endhighlight %}

Once the numbers have been validated, they are passed to an RC4 function and written out to rep.tmp and rep.bin the RC4 password used is "getmypass"

```
push    9
push    offset aGetmypass ; "getmypass"
call    passToRc4
add     esp, 14h
pop     ebp
```

Disabling the "cryp" option in the config file will write plaintext data to the rep.tmp and rep.bin files

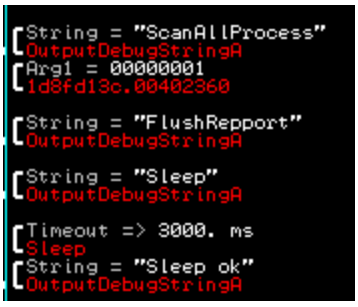


Final Thoughts

To run this malware successfully the attacker would need several pieces of information:

- Credentials
- Name of the POS executable / service
- A method for moving the data out of the network

This malware seems to be in its infancy. There are debug strings still existent in the malware indicate to me that the author is still testing the tool or is still actively developing it.



It's important to track tools like this from their very young stages so that researchers can watch them develop and eventually grow into the next big tool. While this isn't the most advanced POS RAM scraper there is, it's still capable of bypassing all 55 AV's used to scan it.