TrickBot's new API-Hammering explained

() joesecurity.org/blog/498839998833561473



As usual, at Joe Security, we keep a close eye on evasive malware. Some days ago we detected an interesting sample, MD5: b32d28ebab62e99cd2d46aca8b2ffb81. It turned out to be a new TrickBot sample using API hammering to bypass analysis. In this blog post, we will outline the evasion and explain how it works.

The full analysis report of the TrickBot variant is available here.

Two Stage API Hammering

Right after the entry point, the sample tries to load taskmgr.exe as a DLL:

Sections loaded by Program									-
					Mapped to				Searce
File Path	Access	Type	Bose	Size	pid	Protection	Completion	Count	Address
KnownDIb326askmgr.exe	query [write [read]	unknown	unknown	unknown	unknown	unknown	object name	1	401085
	eseculo						and loand		

This is likely a trick to bypass emulators that do not check if a given DLL exists if LoadLibraryEx is called. Next, it performs a massive *printf* loop - the first stage. Since before the loop *FreeConsole* has been called all *printf* calls do basically nothing:

58	0x00401af5	_v8572 = _v8576;
59	0x00401b03	while(1) {
60	0x00401b03	a12 = a12 - 1;
61	0x00401b0b	_t124 = (_t81 + 1) % 0x215f;
62	0x00401b0d	_push("blue");
63	0x00401b17	ti03 = ti31 + ti24 - 0x2168;
64	0x00401b1e	_t84 = *_t103;
65	0x00401b20	_v8557 = _t84;
66	0x00401b2f	_v8568 = _t124;
67	0x00401b37	_t126 = ((_t84 & 0x000000ff) + _v8564) % 0x215f;
68	0x00401b39	_t88 = _t131 + _t126 - 0×2168;
69	0x00401b42	*_t103 = *_t88;
70	0x00401b4a	_v8564 = _t126;
71	0x00401b50	*_t88 = _v8557; // executed
72	0x00401b52	<pre>printf("The color: %s\n"); // executed</pre>
73	0x00401b54	_push(0x3039);
74	0x00401b5e	<pre>printf("First number: %d\n"); // executed</pre>
75	0x00401b60	_push(0x19);
76	0x00401b67	printf("Second number: %04d\n"); // executed
77	0x00401b69	_push(0x4d2);
78	0x00401b73	<pre>printf("Third number: %i\n"); // executed</pre>
79	0x00401b7b	$t_{133} = t_{132} + \theta \times 18;$
89	0x00401b7e	*_t133 = *0x403f00;
81	0x00401b86	printf("Float number: %3.2f\n"); // executed
82	0x00401b88	_push(@xff);
83	0x00401b92	<pre>printf("Hexadecimal: %x\n"); // executed</pre>
84	0x00401b94	_push(0xff);
85	0x00401b9e	printf("Octal: %o\n"); // executed
86	0x00401ba0	_push(0x96);
87	0x00401baa	<pre>printf("Unsigned value: %u\n"); // executed</pre>
.88	0x00401bac	_push(@xa);
89	0x00401bb3	printf("Just print the percentage sign %%\n"); // executed
90	0x00401bc1	asm("cdq");
91	0x00401bc4	_t115 = ((*_t103 & 0x000000ff) + (_v8557 & 0x000000ff)) % 0x215f;
92	0x00401bc6	_t101 = _v8572;
93	0x00401bcc	$t_{132} = t_{133} + \theta_{x2c};$
94	0x00401bd6	*_t101 = *_t101 ^ *(_t131 + ((*_t103 & 0x000000ff) + (_v8557 & 0x000000ff)) % 0x215f - 0x2168);
95	0x00401bdd	_v8572 = &(_t101[0]);
96	0x00401be3	f(all = 0) (
97	0x00000000	break;
98	0x00000000	}
File Will	ten	

File Path	Offset	Leigh	Value	Ancii	Completion	Count	Source Address	Symbol
unknown	unknown	17	64 68 66 20 63 67 6c 61 72 3a 20 62 6c 75 65 0d 0a	The color: blue.	invalid handle	12	77C22FBA	RtUserThreadStart
unknovn	unknovn	21	46 69 72 73 74 20 6e 75 6d 62 65 72 3e 20 31 32 33 34 36 0d 0e	First number: 12345.	invalid handie	11	77C22FBA	REUserThreadStart
unknovn	unknovm	21	53 65 63 67 6e 64 20 6e 75 6d 62 65 72 3a 20 30 30 32 35 0d 0a	Second number: 0025.	invalid handie	11	77C22FBA	RHUserThreadStart
urknovn	unknovm	29	54 68 69 72 64 20 6e 75 6d 62 65 72 3a 20 31 32 33 34 6d 8a	Third number: 1234.	invalid handle	11	77C22FBA	RIUserThreadStart
unknown	unknown	50	46 6c 6f 61 74 20 6c 75 6d 62 65 72 3a 20 33 2c 31 34 0d 8a	Float number: 3.14.	invalid handle	11	77C22FBA	RIUserThreadStart
unknown	unknown	17	40 65 70 61 64 65 63 69 6d 61 6c 3a 20 65 66 6d 8a	Hoxadecimal #.	invalid handle	11	77C22FBA	REUserThreadStart
unknove	unknovm	12	4f 63 74 61 6c 3a 20 33 37 37 6d 8a	Octal: 377	invalid handle	11	77C22FBA	R#UserThreadStart
unknown	unknovn	21	65 6e 73 69 67 6e 65 64 20 75 61 6c 75 65 3a 20 31 35 30 0d 0a	Unsigned value: 150.	invalid handle	11	77C22FBA	REUserThreadStart
unknown	unknovn	34	4a 75 73 74 20 70 72 69 6e 74 20 74 68 65 20 71 65 72 63 65 6e 74 61 67 66 20 73 69 67 6e 21 25 0d 0a	Just print the percentage sign %.	invalid handle	11	7TC22FBA	R#UserThreadStart

This code has been directly copied from the documentation of printf:

Formatting other Types

Until now we only used integers and floats, but there are more types you can use. Take a look at the following example



So what is the purpose of those numerous *printf* loops? Well, sandboxes are designed to log all behavior including the 1.8M calls. As a result, the massive amount of calls delay the execution process and overload the sandbox with junk data. As a result, the final payload is never called.

This behavior is called **API Hammering**. API Hammering is not a new technique, we have already seen it several years ago e.g. in the <u>Nymaim</u> Loader. Joe Sandbox detects the API hammering successfully and rates it as malicious:

Malware Analysis System Evasion:	
High number of junk calls founds (likely related to see	andbox DOS / API hammering)
Source: Global behavior	Junk call stats: NtWriteFile 1841508

Right after the *printf* flood, the sample performs another loop to delay execution by creating and writing to a temporary file - the second stage. In between it performs random sleeps:

File Activities					54	ow Windo	vs behavlor
File Opened							-
File Path	Access	Options	Content overwritten	Completion	Count	Source Address	Symbol
C:Usersiveer/AppOstal.cov/TempTog3055.tep	road attributes synchronize generic read generic write	synchranous io non alert (non directary file	true	success or wall	1	8F067C	CreateFileW
C:Useniuser/AppData/LocalTempYog33C5 tmp	read attributes synchroniza generic read generic write	synchronous io non alert non directary Be	false	success or walt	83	BF071A	CreateFileW
C:Usersiuser/AppData/LocalTemp/log3DC5 trep	read attributes synchronize generic read generic write	synchranous io non alert non directary file	false	success ar walt	1	80904E	CreateFileW

Thread Activities					
Thread	Delayed				-
TID	Delay	Completion	Count	Source	symbol
3640	-234	teucomo or walt	1	BF0556	Sleep
3640	-224	success or wait	1	DF0556	Sleep
3640	-276	naucess or wait	2	BF0556	Sleep
3640	-502	success or walk	2	BF0556	Sloop
3640	-365	saccasa or wait	1	DP0556	Sleep
3640	-272	success or wait	1	DF0556	Sleep
3640	-250	sectors or wait	1	BF0556	Sleep
3640	-366	success or well	1	BP0556	Sheep
3640	-641	success or wait	1	DF0556	Sleep
3640	-362	second or wait	1	BF0556	Sloop
3640	-428	success or wall	1	BF0556	Sheep
3640	-03	success or wat	1	DF0556	Sleep
3640	-523	filew to associate	5	BF0556	Sloop
3640	-17	success or wall	1	BF0556	Ship
3640	-291	success or wait	1	BP0556	Sheep
3640	-414	success or wait	1	BF0556	Sloop
3640	-200	success or wall	1	BF0556	Shorp
3640	-64	success or wait	2	BP0556	Sheep
3640	-149	success or wait	1	DF0556	Sloop
3640	-337	success or wall	1	BF0556	Sleep
3640	-109	secous or wait	2	BP 0556	Sheep
3640	-449	saccess or wait	2	DF0556	Sleep
3640	-190	seccess or wait	1	BF0556	Steep
3640	-520	second or wall	1	BF0556	Sheep
3640	-503	BACCERS OF WHR	1	BP0556	Sheep
36.43	.311	success or with	2	DEBAGA	Classe

Again, the purpose is to overload the sandbox and delay the execution. This time however the all calls are valid.

WERMGR

Finally, when this loop is passed, the sample starts and injects TrickBot (by using directly Nt* APIs) into legit wermgr.exe - the process responsible for Windows error handling and reporting:

Pro	cess Activities							5	show Win	dows behavior
Pro	cess Created									-
PID	File Path	Cmdline			Flags	Completion	Count	Source Address	Symbo	1
5872	C:/Windows/System32/wermgr.exe	C:/Windows/sy	stem32u	wermgr.exe	create suspended	success or wait	1	BF1822	CreateF	rocessInternalW
Memory	Written									E
								Sos	ace	
P10	File Path	Base	Length	Value		Completion		Count Add	frees	Symbol
9072	C.1Windows/System32Ivering: exe	1EBD06F0000	4024	$\begin{array}{c} B_{0} \subset e \in [0, 1] A_{0} \subset D \subset O \subset A_{0} \subset A_{0} \subset O \subset A_{0} \subset A_{0} \subset O \subset A_{0} \subset A_{0$	$\begin{array}{cccccccccccccccccccccccccccccccccccc$	00.00 E0 excess or wait 0.00 00 E0 excess or wait 0.00 00 E0 E0 00 E0 0.00 00 E0 E0 E0 00		1 DF1	122	CreateProcessimenal
5672	C.Windows/System32/wermgreae	6204065208	4	68 90 EF DD EB 61 90 00		success or wait		1 DF1	822	CreateProcessInternal
5672	C:WindowstBystawsD2wermgreee	1EBOCF0000	144616	2 56 66 41 57 60 76 84 70 25 77 6 TC 11 7C 487 C 487 C 58 7C 50 7C 57 FC 59 7C 49 7C 69 7C 50 7C 55 FC , FC 59 7C 49 7C 69 7C 55 FC , FC 59 7C 49 7C 60 7C 55 FC , 50 40 00 60 80 00 60 89 00 00 1 50 00 00 60 80 00 60 89 00 00 1 50 00 00 60 80 00 60 89 00 00 1 50 00 00 60 80 00 60 89 00 00 1 50 00 00 60 80 00 60 89 00 00 1 50 00 00 60 80 00 60 89 00 00 1 50 00 00 60 80 00 60 89 00 00 1 50 00 00 80 80 00 60 89 00 00 1 50 00 00 80 80 00 60 89 00 00 1 50 00 60 80 00 60 80 00 00 1 50 00 60 80 00 60 80 00 00 1 50 00 60 80 00 60 80 00 00 1 50 00 60 80 00 60 80 00 00 1 50 00 60 80 00 60 80 00 00 1 50 00 60 80 00 60 80 00 00 1 50 00 60 80 00 60 80 00 00 1 50 00 60 80 00 60 80 00 00 80 00 00 1 50 00 80 80 00 00 80 00 00 80 00 00 1 50 00 80 80 00 00 80 00 00 80 00 00 1 50 00 80 80 00 80 80 00 00 80 00 00 1 50 00 80 80 00 80 80 00 00 80 00 00 80 00 0	$\begin{array}{c} C = 0 \ 70 \ 40 \ 70 \ 40 \ 70 \ 40 \ 70 \ 40 \ 70 \ 40 \ 70 \ 50 \ 70 \ 7$	B3 7C 5E seccess or wait 57C 42 57C 42 57C 48 5 80 90 00 5 80 90 00 5 80 90 00 6 80 90 00 6 80 90 00 6 80 90 00		1 255	DEOFBASE4	MW/m/VinaalMomoy
4.6.7.7	C. Mandoural Sciences With arriver and	755704643862	100	AR BRIDG TO DO DO DO AR BS	A 45 00 CB 53 C3 00	success or web		4 777		A MARKAGE A COMPANY OF

It's noticeable that a 32bit sample is able to inject successfully into 64bit *wermgr.exe* on a Windows 64bit.

In wermgr.exe TrickBot fully unpacks itself:



This enables Joe Sandbox to successfully detect TrickBot and extract full configurations:

E-Banking Fraud:

Yara detected Trickbot

Source: Yara match

File source: Process Memory Space: wermgr.exe PID: 5764, type: MEMORY

Threatname: Trickbot

ſ
"gtag": "ono45",
"C2 List": [
"110.232.76.39:449",
"134.119.191.11:443",
"107.175.72.141:443",
"36.91.45.10:449",
"185.90.61.9:443",
"5.1.81.68:443",
"185.99.2.65:443",
"185.99.2.66:443",
"45.6.16.68:449",
"110.50.84.5:449",
"181.112.157.42:449",
"181.129.104.139:449",
"200.107.35.154:449",
"182.253.113.67:449",
"85.204.116.216:443",
"95.171.16.42:443",
"103.111.83.246:449",
"194.5.250.121:443",
"181.129.134.18:449",
"134.119.191.21:443",
"190.136.178.52:449",
"110.93.15.98:449",
"91.235.129.20:443",
"80.210.32.67:449",
"36.89.182.225:449",
"185.14.31.104:443",
"192.3.247.123:443",
"36.66.218.117:449",
"122.50.6.122:449",
"103.12.161.194:449",
"121.100.19.18:449",
"85.204.116.100:443",
"131.161.253.190:449",
"36.92.19.205:449",
"78.108.216.47:443",
"36.89.243.241:449",
"51.81.112.144:443"
J,
"modules": [
"bwarab".

Conclusion

In contrast to many other evasions, API Hammering is one of the more interesting techniques since it directly exploits the design of a sandbox. No matter what technology your favorite sandbox uses, it has to handle API Hammering correctly.

You are interested to get a list of other evasive malware analyses? Check out these other blogs:

- New Sandbox Evasions spot in VBS samples
- Analyzing Azorult's Anti-Analysis Tricks with Joe Sandbox Hypervisor
- Fighting Country Aware Microsoft Office Macro Droppers with VBA Instrumentation
- Malicious Documents: The Evolution of country-aware VBA Macros

or this extensive list of evasive samples.

Interested in Joe Sandbox? Register for free at <u>Joe Sandbox Cloud Basic</u> or <u>contact us</u> for an in-depth technical demo!