

Deep Malware and Phishing Analysis

By Joe Security LLC

Archived: 2026-04-06 01:37:20 UTC



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:

File Path	Access	Type	Base	Size	Mapped to pid	Protection	Completion	Count	Source Address
KoonDb32taskmgr.exe	query write read execute	unknown	unknown	unknown	unknown	unknown	object name not found	1	401C85

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          _t103 = _t131 + _t124 - 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 - 0x2168;
69 0x00401b42          *_t103 = *_t88;
70 0x00401b4a          _v8564 = _t126;
71 0x00401b50          *_t88 = _v8557; // executed
72 0x00401b52          printf("The color: %s\n"); // executed
73 0x00401b54          _push(0x3039);
74 0x00401b5e          printf("First number: %d\n"); // executed
75 0x00401b60          _push(0x19);
76 0x00401b67          printf("Second number: %04d\n"); // executed
77 0x00401b69          _push(0x4d2);
78 0x00401b73          printf("Third number: %i\n"); // executed
79 0x00401b7b          _t133 = _t132 + 0x18;
80 0x00401b7e          *_t133 = *0x403f00;
81 0x00401b86          printf("Float number: %3.2f\n"); // executed
82 0x00401b88          _push(0xff);
83 0x00401b92          printf("Hexadecimal: %x\n"); // executed
84 0x00401b94          _push(0xff);
85 0x00401b9e          printf("Octal: %o\n"); // executed
86 0x00401ba0          _push(0x96);
87 0x00401baa          printf("Unsigned value: %u\n"); // executed
88 0x00401bac          _push(0xa);
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          _t132 = _t133 + 0x2c;
94 0x00401bd6          *_t101 = *_t101 ^ (*(_t131 + (( *_t103 & 0x000000ff) + (_v8557 & 0x000000ff)) % 0x215f - 0x2168);
95 0x00401bdd          _v8572 = &(_t101[0]);
96 0x00401be3          if(_a12 == 0) {
97 0x00000000              break;
98 0x00000000          }

```

File Path	Offset	Length	Value	Ascii	Completion	Source Code	Source Address	Symbol
unknown	unknown	17	64 68 95 28 63 0f 6c 6f 72 3a 20 62 6e 75 65 9d 0a	The color: blue	Invalid handle	12	77C22FBA	RtlUserThreadStart
unknown	unknown	21	46 69 72 73 74 20 6e 75 6d 62 65 72 3e 20 31 32 33 34 35 9d 0a	First number: 12345.	Invalid handle	11	77C22FBA	RtlUserThreadStart
unknown	unknown	21	53 65 63 6f 6e 64 20 6e 75 6d 62 65 72 72 3a 20 38 39 3e 35 9d 0a	Second number: 0325.	Invalid handle	11	77C22FBA	RtlUserThreadStart
unknown	unknown	20	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	RtlUserThreadStart
unknown	unknown	20	46 6c 6f 61 74 20 6e 75 6d 62 65 72 3a 20 33 2e 31 34 6d 8a	Float number: 3.14.	Invalid handle	11	77C22FBA	RtlUserThreadStart
unknown	unknown	17	48 65 70 61 64 65 63 69 6d 61 6c 3a 28 65 06 0d 8a	Hexadecimal: f	Invalid handle	11	77C22FBA	RtlUserThreadStart
unknown	unknown	12	4f 63 74 61 6c 3a 20 33 37 3f 6d 8a	Octal: 377.	Invalid handle	11	77C22FBA	RtlUserThreadStart
unknown	unknown	21	65 6e 73 68 67 6e 65 64 20 75 61 6c 75 65 3a 28 31 35 38 9d 0a	Unsigned value: 150.	Invalid handle	11	77C22FBA	RtlUserThreadStart
unknown	unknown	34	4a 75 73 74 20 70 72 69 6e 74 20 74 68 65 20 78 65 72 63 65 6e 74 61 6f 65 20 73 68 67 6e 28 25 0d 0a	Just print the percentage sign %.	Invalid handle	11	77C22FBA	RtlUserThreadStart

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:

```
#include<stdio.h>

main()
{
    printf("The color: %s\n", "blue");
    printf("First number: %d\n", 12345);
    printf("Second number: %04d\n", 25);
    printf("Third number: %i\n", 1234);
    printf("Float number: %3.2f\n", 3.14159);
    printf("Hexadecimal: %x\n", 255);
    printf("Octal: %o\n", 255);
    printf("Unsigned value: %u\n", 150);
    printf("Just print the percentage sign %%\n", 10);
}
```

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 [Nymaim](#) 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 sandbox 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 Paths	Access	Options	Content overwritten	Completion	Count	Source Address	Symbol
C:\Users\user\AppData\Local\Temp\log13C5.tmp	read attributes synchronize generic read generic write	synchronous to non alert non directory file	true	success or wait	1	BF067C	CreateFileW
C:\Users\user\AppData\Local\Temp\log13C5.tmp	read attributes synchronize generic read generic write	synchronous to non alert non directory file	false	success or wait	83	BF071A	CreateFileW
C:\Users\user\AppData\Local\Temp\log13C5.tmp	read attributes synchronize generic read generic write	synchronous to non alert non directory file	false	success or wait	1	BD9D4E	CreateFileW

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

```
{
  "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"
  ],
  "modules": [
    "awarab".
  ]
}
```

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 [Joe Sandbox Cloud Basic](#) or [contact us](#) for an in-depth technical demo!

Source: <https://www.joesecurity.org/blog/498839998833561473>