Your Facebook connection is now secured! Thank you for your support!

blog.avast.com/2013/06/18/your-facebook-connection-is-now-secured/

Jaromír Hořejší 19 Jun 2013

Your Facebook connection is now secured! Thank you for your support!

The title of this blog post may make you think that we will discuss the security of your Facebook account. Not this time. However, I will analyze an attack which starts with a suspicious email sent to the victim's email account.

The incoming email has the following subject, '**Hey <name> your Facebook account has been closed!**' or '**Hi <name> your Facebook account is blocked!**'. The email has a ZIP file attachment with name <name>.zip, which contains a downloader file named <name>.exe. <name> stands for a random user name. After a user downloads and executes the executable file, he is presented with the message saying that "Your Facebook connection is now secured! Thank you for your support!" It tries to convince you that there was a problem with your Facebook account, which was later successfully solved by executing the application from the email attachment.

Let's look inside the executable file!

fbsec01

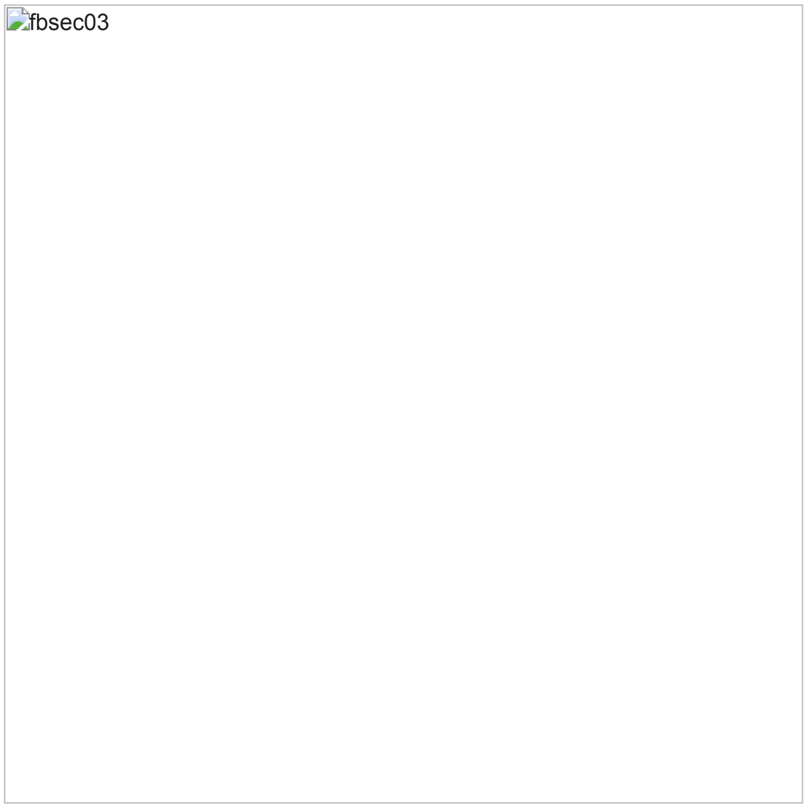
Unlike many other malware samples, which use various malware cryptors (see an article about the interesting one), this malware sample does not use any cryptor. Instead, when observing the instruction flow, we notice many useless registry computations and memory operations, which make it harder to analyze the sample. All text strings and names are encrypted in the malicious file and decrypted on the fly, when needed. In the figure below, you can see many registry and memory operations from address 0x408a8c to 0x408aca, whose purpose is to make it difficult to understand the original function of the code.

fbsec21

Whenever I get a suspicious file, I start OllyDbg and begin analyzing the file. In the case of this sample, I loaded it in OllyDbg, made it run and the following error message box appeared.

fbsec02

Then I tried to figure out what could be wrong with the sample I just started to analyze. In the beginning, there is a loop with 0x109 = 265 iterations. In each iteration, a new thread is created.

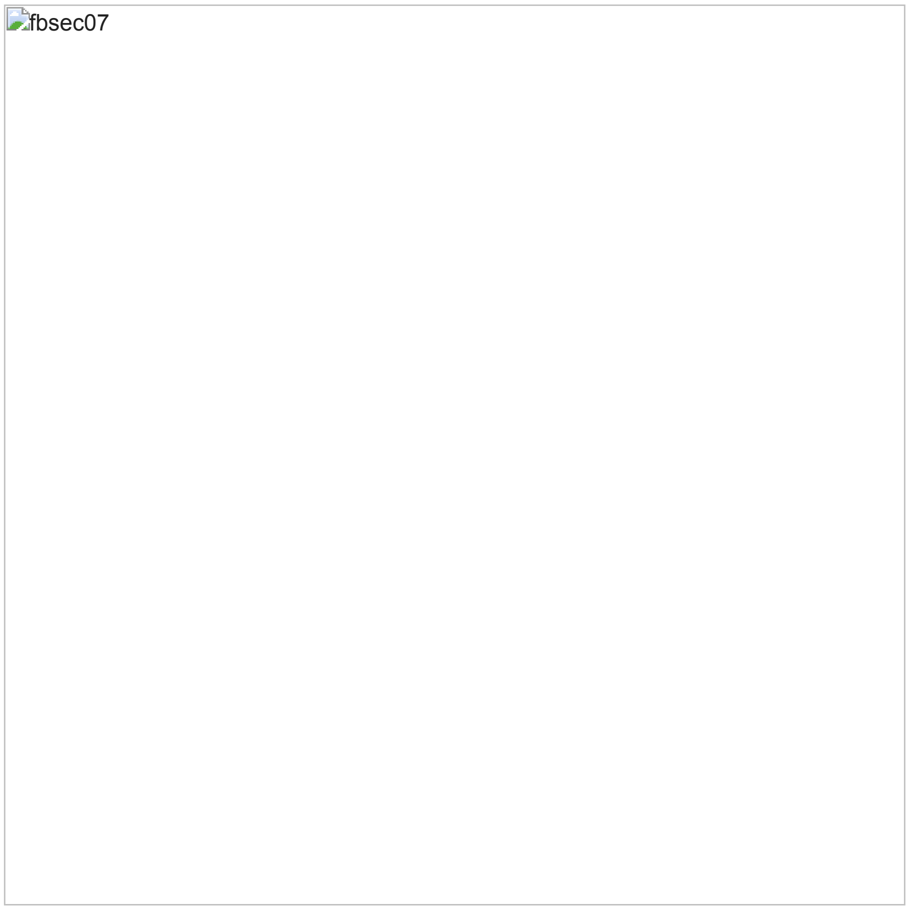


Each thread executes its own thread function, in which it creates a manual-reset event object (CreateEventA), which requires the use of functions ResetEvent or SetEvent to change the event state. Later on, the function WaitForSingleObject with timeout 0x2710 = 10000 ms = 10 seconds makes the thread wait for setting the state of the event manually. If the state of the event is set or if its timeout expires, a DWORD value at addressOfProcedure is XORed with a certain value unique for each thread. These per-thread unique values are taken from arrayOfDwords table, which starts at address 0x422488 (file offset 0x20a88).

fbsec05

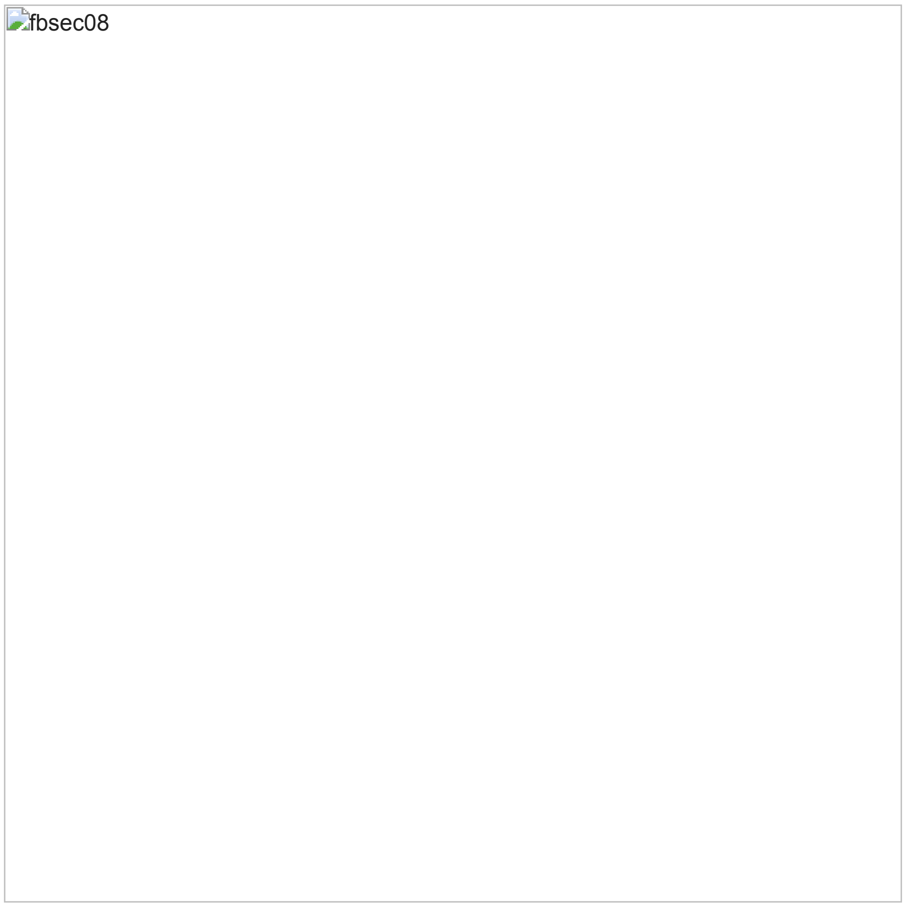
An application then sets events for the four given threads, which causes function WaitForSingleObject to end immediately. DWORD at addressOfProcedure is then XORed with the four corresponding values from arrayOfDwords. After these four XOR operations, addressOfProcedure contains the address of function which will be called by the main program.

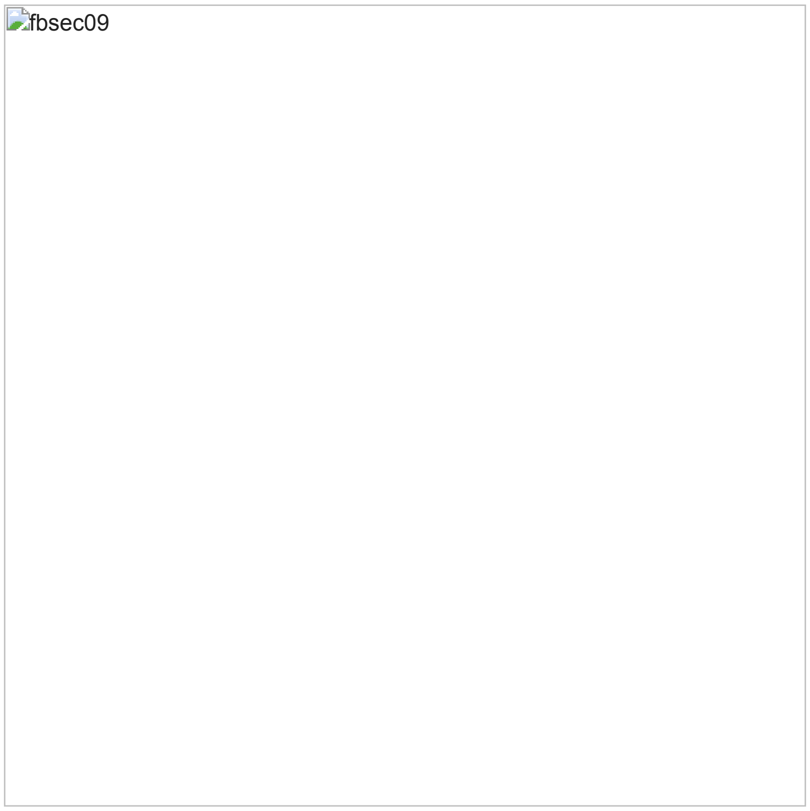
fbsec06



In our situation, the thread to be woken up are 0xbf, 0xd4x 0xd3x 0xf5. In arrayOfDwords, the thread unique values are stored at addresses
0x20a88 + 0xbf*4 = 0x20d84
0x20a88 + 0xd4*4 = 0x20dd8
0x20a88 + 0xd3*4 = 0x20dd4
0x20a88 + 0xf5*4 = 0x20e5c
from where we can get per-thread unique values, which after being XORed give us the following result:

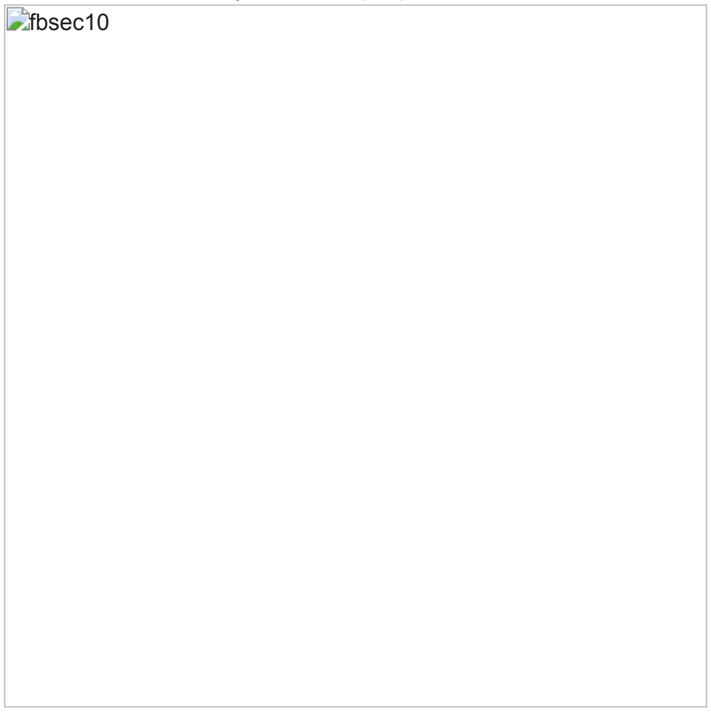
0x9329c591 XOR 0xc3b12028 XOR 0x732eb78b XOR 0x23f618f2 = 0x00404ac0
Therefore the next address of execution will be 0x404ac0.


fbsec08

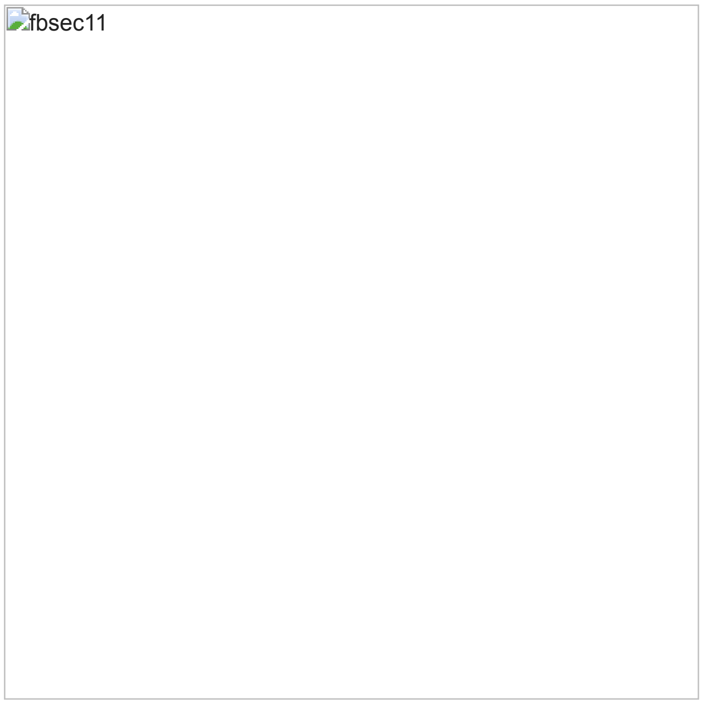

fbsec09

While 261 out of 265 created threads are still sleeping (and waiting for the event being set or timeout interval to elapse) just four threads are woken up, and these threads compute the function address which will be called. OllyDbg cannot handle this situation correctly, computes the wrong destination address and therefore displays the above mentioned error message. After 10 seconds, timeouts of all threads will elapse and addressOfProcedure will be modified to an invalid address value, however, it will happen after the program already jumped to address 0x404ac0 and DWORD value at addressOfProcedure is no longer important.

After executing the procedure from address 0x404ac0, the main program body begins. The program flow can be split into three main branches. At first, the program tries to find out, if it was executed with a command line parameter containing string WATCHDOGPROC. If yes, the left (red) branch is chosen. If not, the right (green) branch is executed.



If WATCHDOGPROC string in command line parameters was not found, then there is another branch asking if the name of the current executable is usfqvololjv.exe. If not, we take the left (red) branch.



In this (left) branch, the file copies itself into %APPDATA%\ltrhborczvnt\usfqvololjv.exe, executed itself, establishes persistence via registry key, displays the message "Your Facebook connection is now secured! Thank you for your support! Facebook" and terminates.

fbsec12

fbsec13

The whole process is repeated again, but now the condition where the current program name is compared with usfqvololjv.exe is satisfied. We can see that usfqvololjv.exe copies itself under another name tjsotyw.exe and executes it with commandline parameter "WATCHDOGPROC usfqvololjv.exe".

fbsec14

Now it becomes clear that usfqvololjv.exe is a master process and tjsotyw.exe is a slave process.

fbsec15

The master process (usfqvololjv.exe) then continues into an internet communication loop, which generates traffic to seemingly legitimate websites. The URL address is always in format <WORD1><WORD2>.net/forum/search.php?email=<EMAIL_ADDRESS>&method=post

fbsec16

The only task of the slave process is to check if the master process is running. If not, it restarts the master process. Similarly, if the master process finds out that the slave process is not running, it restarts the slave process, so both processes keep running all the time, keeping an eye one on another.

Domain names are generated by an algorithm, which uses the value of the current time. It starts with obtaining the current Unix epoch time, which is a system for describing time, defined as the number of seconds that have elapsed since 00:00:00 Coordinated Universal Time (UTC), 1 January 1970. This number is then divided by 0x200 = 512. Time is then divided into 512 seconds = 8 minutes 32 seconds long time chunks.

Let's look at a particular example. For the time interval between "Fri, 07 Jun 2013 12:45:52 GMT" and "Fri, 07 Jun 2013 12:54:23 GMT", we get Unix epoch times between 0x51B1D600 and 0x51B1D7FF. After dividing any of the numbers between previously mentioned borders by 0x200 = 512, we get the following result.

fbsec17

The result (0x28d8eb) is then converted to its binary form and its last 15 binary digits are reordered (LSB bit of 0x28d8eb goes to the 3rd position, the second LSB bit goes to 9th position, etc...). From the newly reordered 15 binary digits, the first 7 binary digits form a number, which gives us an index of the first word in the table of words. The last 8 digits form another number, which gives us an index of the second word in the table of words. These two words are then concatenated and a generic top-level domain .net is appended. The following picture illustrates how this domain-generation algorithm works.

fbsec18

The table of words is constant and encrypted in the original file. There are exactly 384 words in this table. As I mentioned above, from the current time stamp, a 15 digit number is generated. From this number, the first 7 digits give us 128 possibilities (2^7), last 8 digits give us 256 possibilities (2^8), which makes a total of 128 + 256 = 384 words. If we choose one word from the first group and one word from the second group, it gives us a total count of 128 * 256 = 32768 possible domains, which may be contacted.

fbsec19

However, the domain-generation algorithm does not try to connect to only one website withing a given 8.5 minute time chunk. It tries 0x55 = 85 domains for numbers following 0x28d8eb, i.e. 0x28d8eb, 0x28d8ec, 0x28d8ed, 0x28d8ee ... 0x28D93f. When all 85 possibilities are tried, then the time stamp is taken again and the whole process repeats.

When a payload is downloaded after a successful connection to the generated domain, it is then written to %TEMP% directory, named g52<random>arg.exe and executed.

fbsec20

Conclusion:

Obfuscation does not need to be done with a cryptor. Filling the code with many useless registry and memory instructions can do the same job.

Malware authors often use domain-generation algorithms. If malware connects to just a few websites to get updates or payloads, it is easy to block these domains and make malware ineffective. However, in the case of generating many domain names via domain-generation algorithms, it is often impossible to block all the randomly generated domains, either because of their huge number or because of the fact, that some of these domains may be legitimate websites.

SHAs:
EC8B88A96D1B4917334BDAD7F2E580EAD4D9B71D111A1591BB5B965DA3E27CF6