DrLtrace: Applying Binary Instrumentation for Light-Weight Dynamic Malware Analysis

Short Summary

Dynamic binary instrumentation (DBI) is a technique of analyzing the behavior of a binary application at runtime through the injection of instrumentation code. Nowadays, it is successfully applied for vulnerabilities detection, reverse-engineering and many other fields (optimization, memory leaks detection, performance analysis). However, DBI is undeservedly limited by unpacking automatization [1] [2] and several proofs of concepts for instructions, basic blocks and function calls tracing which are not possible to apply for malware analysis in practice. However, this technology can be a reasonable trade-off in terms of visibility versus runtime overhead and give us several serious benefits comparing with existent approaches for dynamic malware analysis.

In our talk, we will present DrLtrace; the first open-source binary instrumentation tool for dynamic API calls tracing. We plan to discuss a motivation of this research, describe important disadvantages of modern techniques used for malware analysis as well as present technical details of our solution.

DrLtrace successfully passes beta-testing stage and now is used in our lab on daily basis for analysis of new sophisticated malicious samples. In our demo, we will show how this tool can be applied to revel in several minutes a lot of internal technical details about several sophisticated malicious samples without even starting IDA or debugger.

Authors

Maksim Shudrak, IBM Research Israel (Haifa Research Lab).

Maksim Shudrak is a security researcher at IBM Research Israel where he is developing advanced solutions for highly-evasive malware analysis and detection. His research interests include reverse engineering, vulnerabilities hunting and malware analysis. Maksim is a main contributor to the DynamoRIO framework

He holds a PhD from Tomsk State University of Radioelectronics and Control Systems. Maksim had a chance to speak at various academic (IEEE EuroCon, IEEE EuroSim, HPC-UA) and industry security conferences (Positive Hack Days, Kaspersky CyberSecurity for The Next Generation) around the world.

Introduction & Motivation

Dynamic analysis is a powerful technique for malware investigation. A tempting idea to understand a goal and actual malicious intent of malware authors without labor-intensive reverse-engineering attracts with its simplicity many researchers. Moreover, sophisticated packers like Themida and Armadillo and/or code plus data encryption significantly facilitate (in some cases making it completely impossible) static reverse engineering of such samples thereby delaying detection. In such case, actual execution of malicious code in a sandbox can reduce amount of time required to understand malicious intent and produce detection signature. However, malware writers don’t stop there and keep successfully evolving new anti-research tricks that aim to detect and protect against unwanted analysis.

The reverse cite of dynamic analysis simplicity is a loss of visibility towards malicious behavior which is called semantic gap problem [3]. For example, ProcMon tool written by Mark Russinovich allows to handle system calls performed by the target process. However, we miss library calls thereby reducing amount of information we know about a sample which is very sufficient for malware reverse engineering. For example, we can successfully detect system calls associated with API call GetUserNameExW. However, subsequent operations in memory with user name will stay hidden for us thereby significantly reducing our understanding of an actual goal of this call.

However, we can use more sophisticated technical tricks to be able to provide more visibility towards our malicious sample. For example, PANDA [4] written on top of QEMU full-system emulator provides a rich API for tracing and even allows to handle each executed instruction in the emulated operating system. Unfortunately, despite the all sophisticated engineering solutions which significantly boosting instructions emulation speed; full-system emulation still introduces significant runtime overhead (x150-x200) which increases even more when we apply additional instrumentation procedures (x350-x450). Malware is often take advantage of this fundamental problem and apply some tricks to delay execution by applying stalling code.

For example, ZBot has a long loop at the beginning of the execution comparing `cycle_count` with `0x2DC6C0` and calling twice function at `[ebp+var_64]` which contains a code that unpack 1 byte of malicious payload per call (Figure 1).

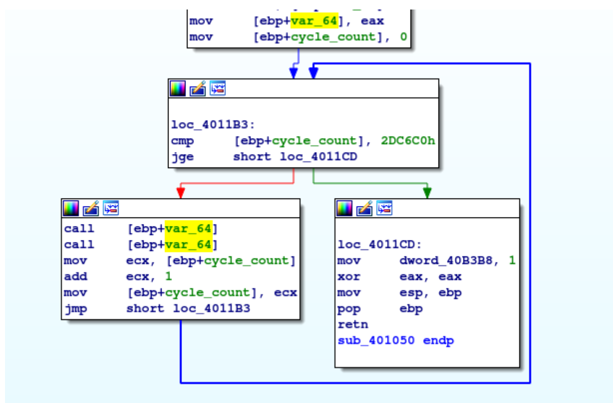


Figure 1. Stalling code in ZBot

Locky's authors made a bet on intensive usage of recursion to prevent against automatic stalling-loops bypassing (Figure 2) which significantly decreases performance of emulator. This stalling function is used in many places of the malware especially during initialization before unpacking routine.

```
int __thiscall stalling_func(int arg1)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  tmp_arg1 = arg1;
  if ( arg1 )
  {
    if ( arg1 == 1 )
    {
      result = 1;
    }
    else
    {
      res = stalling_func(0);
      res2 = res + stalling_func(tmp_arg1 - 1) + 1;
      res3 = stalling_func(1) + 1;
      res4 = res3 - (stalling_func(0) + 1);
      res5 = res4 + stalling_func(1) + 1;
      result = res5 + stalling_func(tmp_arg1 - 2) - res2;
    }
  }
  else
  {
    result = 0;
  }
  return result;
}
```

Figure 2. Locky 's stalling recursive function

On a real processor, the functions described above will be executed in several seconds or minutes while emulator may spend hours and even days trying to reach an actual payload of malware.

It is also sufficient to add that according to [5], there is a huge set of instructions that behaves differently or even causes a crash in emulators. For example, instruction *xgetbv* which is used in Locky malware causes segmentation fault in the unpatched version of QEMU (Figure 3).

```
xor     ecx, ecx
xgetbv
mov     [ebp+var_10], eax
mov     [ebp+var_C], edx
mov     eax, [ebp+var_10]
mov     ecx, [ebp+var_C]
and     eax, 6
xor     ecx, ecx
cmp     eax, 6
jnz     short loc_40A56C
```

Figure 3. Locky 's xgetbv instruction which causes a crash in QEMU

API-hooking might be a reasonable trade-off between system calls monitoring and full-system emulators like PANDA, but the approach is well studied and can be easily detected and/or bypassed as shown in these works [6] [7]. The anti-hooking techniques are quite often used by malware and commercial packers like Themida.

Linux has a great tool which is based on library calls hooking called *ltrace*.

```
osboxes@osboxes:~$ ltrace ls
malloc(552)                                                    = 0x1afc010
malloc(120)                                                    = 0x1afc240
malloc(1024)                                                   = 0x1afc2c0
free(0x1afc2c0)                                                = <void>
free(0x1afc010)                                                = <void>
__libc_start_main(0x402a00, 1, 0x7fffa84730b8, 0x413be0 <unfinished ...>
strrchr("ls", '/')                                             = nil
setlocale(LC_ALL, "" <unfinished ...>
malloc(5)                                                      = 0x1afc010
free(0x1afc010)                                                = <void>
```

Figure 4. A part of ltrace output for ls command

Using one bash command, we can easily get the full trace of library-calls of a certain executable. *Why don't we have such solution for Windows which is also transparent against anti-research tricks used by modern malware?*

It turns that there is a technique that can help us to have such tool for Windows and trace API calls transparently towards executed program. This technique is called dynamic binary instrumentation aka DBI.

Technique description

Dynamic binary instrumentation is a technique of analyzing the behavior of a binary application at runtime through the injection of instrumentation code. Nowadays, the most efficient and transparent solution for binary instrumentation is DynamoRIO DBI framework supported by Google. Curious reader might be interested why DynamoRIO and not Intel Pin which is also very popular DBI framework. We decided to use DynamoRIO motivated by the following reasons:

1) The source code of DynamoRIO is available on github and distributed under BSD license while Intel Pin is proprietary.
2) One of the basic requirements for DynamoRIO at the time of development was transparency towards the instrumented executable.
3) DynamoRIO uses different technology of instrumentation based on code transformation (described in Figure 5) while Intel PIN uses special trampolines which is not transparent towards analyzed executable.

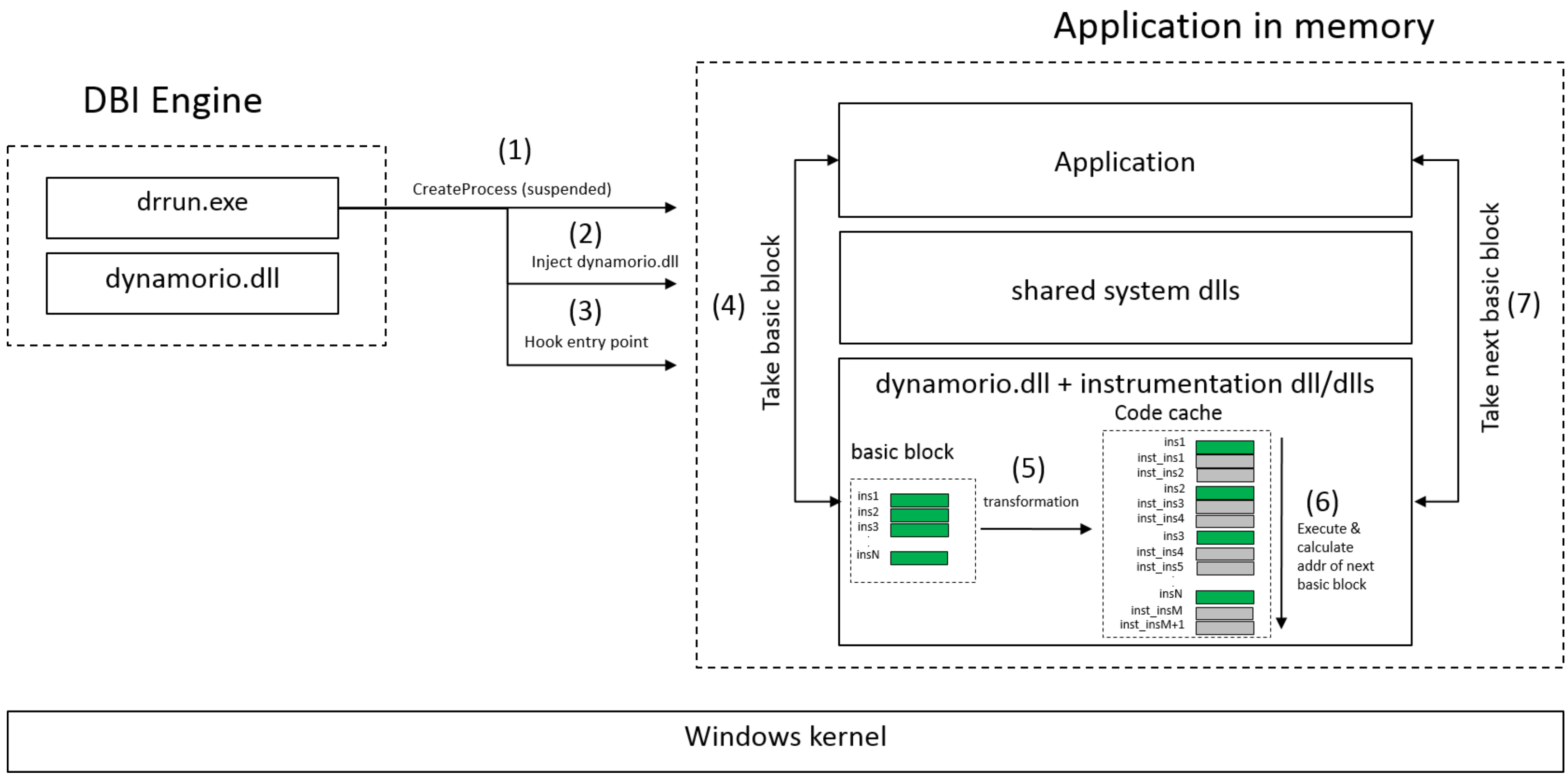


Figure 5. DynamoRIO architecture and step-by-step description

Per Figure 5, the first stages look like a classical DLL-injection. An application starts in suspended state, the core DBI-framework's DLL is injected in a target and then control flow is redirected into this DLL. At the next step, the library code performs environment initialization, loads all required additional modules and then loads user-written DLL to begin instrumentation.

After that, at stages 4-6, it begins sequential translation of each basic block (a sequence of instructions without conditional or unconditional jumps) into intermediate representation. Depending on user's requirements (defined in a user DLL), framework performs injection of instrumentation code (marked gray in our example) and then second translation back into machine code for execution in so-called "code cache". Then it performs address calculation of a next executed basic block - > translation -> instrumentation -> translation back into machine code -> execution -> calculation of a next basic block and so on until the program finish.

Even though the general idea seems relatively trivial, correct and effective (in terms of reliability, transparency and performance) implementation of this approach is a complex engineering task which is considered in these works [8] [9] [10].

DrLtrace

Based on this framework, the authors have implemented a new tool for library calls tracing called DrLtrace. The general technical approach which is the basis of DrLtrace is represented in Figure 6.

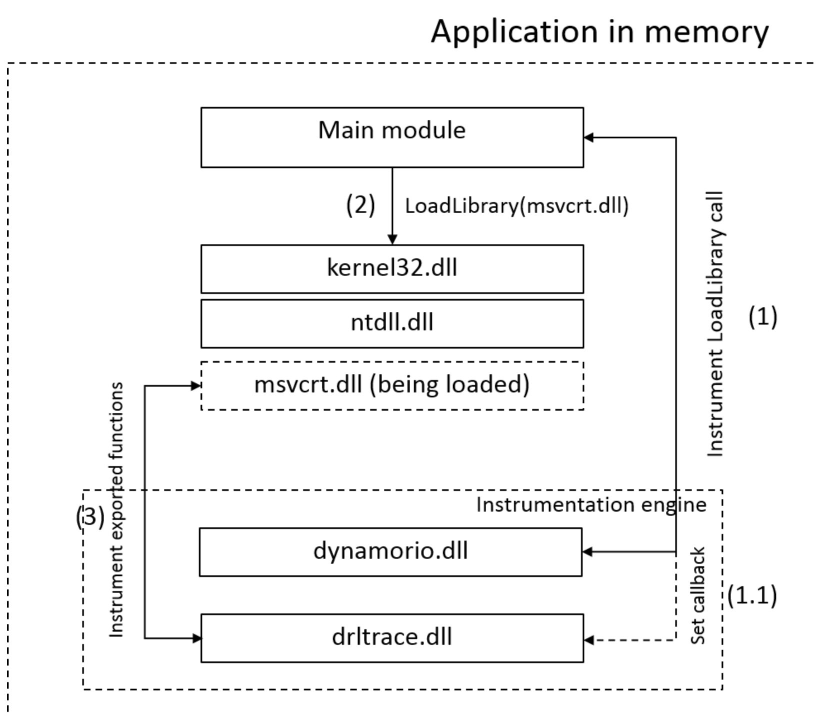


Figure 6. An example of instrumentation scheme of msvcrt.dll by DrLtrace

At the first step (after DBI-framework injects instrumentation engine and initialize environment as described in Figure 5), DrLtrace asks DynamoRIO to perform instrumentation of LoadLibrary call to be able to handle new libraries being loaded by the target process (Figure 7). When the process tries to load a new library, DynamoRIO redirects control flow in DrLtrace.dll. In turn, DrLtrace enumerates exported functions in the newly loaded DLL and registers a special callback for each of them. Thus, if some exported function would be called by malware, DrLtrace's callback will be executed before this function and the tool will be able to log all required information such as a function name and arguments. Another callback might be registered after the function to save results of execution.

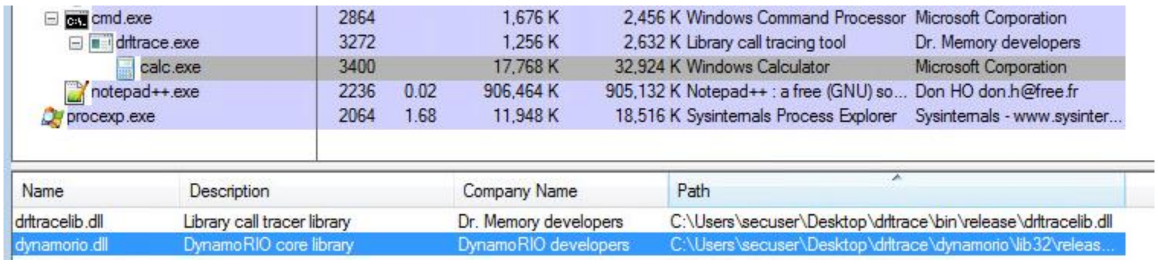


Figure 7. DynamoRIO core library and DrLtrace.dll in the memory of calculator

In practice, the usage of DrLtrace is very simple. A user needs to specify a log directory and a name of a target process in the following way:

```
drltrace.exe -logdir . – calc.exe
```

That's all, the framework will inject required DLLs in the target process, starts instrumentation and in parallel will log information about all library calls which are executed in the target process. Example of the log file which was produced for calculator is shown in Figure 8.

```
~~3516~~ ntdll.dll!RtlInitUnicodeString
    arg 0: 0x0021d1e4 (type=<unknown>*, size=0x0)
    arg 1: \Registry\Machine\System\Setup (type=wchar_t*, size=0x0)
    and return to module id:8, offset:0xf305
~~3516~~ ntdll.dll!ZwOpenKey
    arg 1: 0x20019 (type=unsigned int, size=0x4)
    arg 2: 0x0021d1ec (type=OBJECT_ATTRIBUTES*, size=0x4)
    and return to module id:8, offset:0xf352
~~3516~~ ntdll.dll!KiFastSystemCall
    arg 0: 0x7516f352
    arg 1: 0x0021d214
    and return to module id:22, offset:0x45d14
~~3516~~ ntdll.dll!KiFastSystemCallRet
    arg 0: 0x7516f352
    arg 1: 0x0021d214
    and return to module id:22, offset:0x45d14
~~3516~~ ntdll.dll!RtlInitUnicodeString
    arg 0: 0x0021d204 (type=<unknown>*, size=0x0)
    arg 1: OOBEInProgress (type=wchar_t*, size=0x0)
    and return to module id:8, offset:0xf36f
~~3516~~ ntdll.dll!ZwQueryValueKey
    arg 0: 0x190 (type=HANDLE, size=0x4)
    arg 1: 0x0021d204 (type=UNICODE_STRING*, size=0x4)
    arg 2: 0x1 (type=int, size=0x4)
    arg 4: 0x80 (type=unsigned int, size=0x4)
    and return to module id:8, offset:0xf395
~~3516~~ ntdll.dll!KiFastSystemCall
    arg 0: 0x7516f395
    arg 1: 0x00000190
```

Figure 8. An example of the log file produced by running calc.exe

The format of the output is simple and can be easily parsed by an external script:

```
~~[thread id]~~ [dll name]![api call name]
arg [arg #]: [value] (type=[Windows type name], size=[size of arg])
and return to module id:[module unique id], offset:[offset in memory]
```

The module unique identifiers are printed at the end of the log file (Figure 9)

```
Module Table: version 3, count 30
Columns: id, containing_id, start, end, entry, checksum, timestamp, path
  0,   0, 0x005e0000, 0x00755000, 0x005edbf7, 0x00000000, 0x59ce1b0f,  C:\Users\secuser\Desktop\drltrace\bi
  1,   1, 0x00f80000, 0x01040000, 0x00f92d6c, 0x000cbd30, 0x4ce7979d,  C:\Windows\system32\calc.exe
  2,   2, 0x57260000, 0x573c1000, 0x57303940, 0x00136d65, 0x59ce1b0b,  C:\Users\secuser\Desktop\drltrace\dy
  3,   3, 0x707c0000, 0x707f2000, 0x707c37f1, 0x00035432, 0x4ce7ba42,  C:\Windows\system32\WINMM.dll
  4,   4, 0x737b0000, 0x73940000, 0x7384d063, 0x001936bc, 0x4f9235ab,  C:\Windows\WinSxS\x86_microsoft.winc
  5,   5, 0x739b0000, 0x739f0000, 0x739ba2dd, 0x0004a58b, 0x4a5bdb38,  C:\Windows\system32\UxTheme.dll
  6,   6, 0x73d80000, 0x73f1e000, 0x73dae6b5, 0x0019ca5f, 0x4ce7b71c,  C:\Windows\WinSxS\x86_microsoft.winc
  7,   7, 0x74440000, 0x74449000, 0x74441220, 0x000138c1, 0x4a5bdb2b,  C:\Windows\system32\VERSION.dll
  8,   8, 0x75160000, 0x751ab000, 0x75167e10, 0x00052995, 0x50b83b16,  C:\Windows\system32\KERNELBASE.dll
  9,   9, 0x752c0000, 0x752ca000, 0x752c136c, 0x000093af, 0x4a5bda19,  C:\Windows\system32\LPK.dll
```

Figure 9. An example of the module table for calc.exe

DrLtrace can easily filter out interlibrary calls and print only API calls performed from the main module (or from a heap) of a target application by specifying *-only_from_app* option which is very useful in case of applications that generate huge logs. DrLtrace also has several useful external scripts to filter API calls for certain library, print only potentially interesting API calls and strings.

Why is DrLtrace cool ?

- Fast enough to perform analysis of malicious samples without being detected by time-based anti-research techniques (Figure 10).

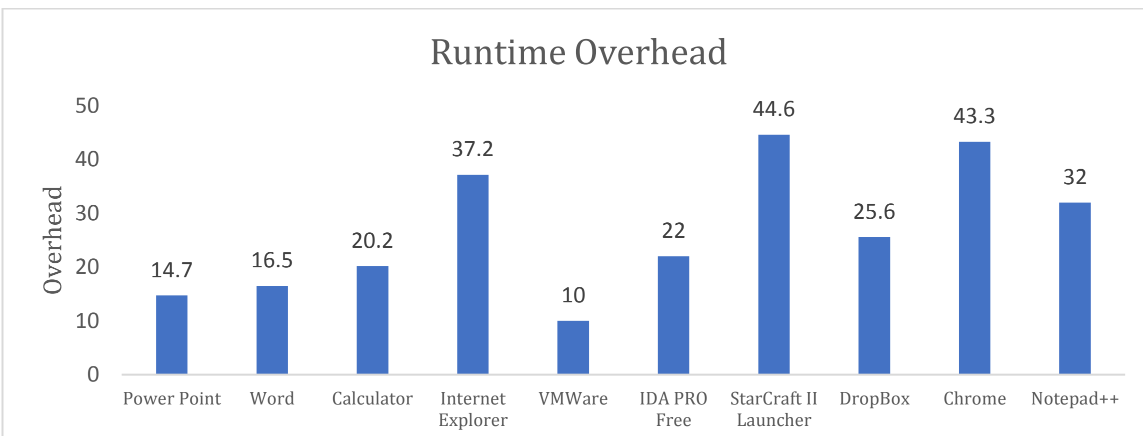


Figure 10. Runtime overhead of DrLtrace for several heavy-weight Windows applications

- Supports both x86 and x64 (ARM in future).
- Supports both Windows and Linux (macOS in future).
- Supports self-modifying code.
- Supports all types of library linkage (static and dynamic).
- Not-detectable by standard malware anti-research approaches (anti-hooking, anti-debugging and anti-emulation).
- User can easily add a new function prototype to tell DrLtrace how to print more details about previously unknown API calls (even about non-system DLLs). External configuration file is used.
- Easy-to-use and modify for your own purposes (no additional requirements, no heavy-weight GUI interface).
- Open-source (BSD-license).

DrLtrace successfully passes beta-testing stage and now is used in our lab on daily basis for analysis of a new incoming malware samples.

Case studies

Let us show three examples when DrLtrace helps to reveal important information and significantly reduce amount of work required for malware analysis.

It is important to note that below, we provided a lot of text and screenshots of what we plan to present in a live demo at the BlackHat Arsenal. We will run each sample under DrLtrace and show how the tool can be easily used to do fast dynamic analysis of malicious samples.

Example 1. EmbusteBot

In one of our recent collaborations with IBM Security Trusteer team, we found a new financial malware that targets dozens of major Brazilian banks, but beyond its generic capabilities further employs specific malicious schemes for different banks, and allows an attacker to gain full control of a victim's endpoint. We were first who found this sample and dubbed the malware *EmbusteBot*, after the Portuguese word "Embuste", a hoax/scam.

The Brazilian cybercriminal scene [11] is known for its affinity for Delphi-based malcode, and the sample we analyzed is no exception to the popular use of Delphi in Brazilian malware. The malware's authors in this case employ a scheme where a benign executable is used to load a malicious DLL on the target endpoint to activate the payload. While in general, the sample doesn't intensively use anti-research tricks, there is some encryption of sensitive strings in several important parts of the DLL, as well as time-based anti-research checks the malware performs.

The overall purpose of EmbusteBot is to:

- Find out which browser window runs on the victim's machine;
- Match the window title with a list of banks it targets;
- Take over the victim's endpoint, use fake overlays in some cases;
- Launch fraudulent transactions from the victim's account.

EmbusteBot's most likely delivery path lies in malware-laden email spam. The malware's execution on target endpoints begins with dynamic loading of a malicious DLL to find out what browser the victim uses, and what's on the active tab.

Let's start EmbusteBot under DrLtrace:

```
drltrace.exe -logdir . -print_ret_addr – vdeis.exe
```

The sample will stop very early and in the log file we can clearly see that it requires vdeis2.dll to be in the `C:\Users\Public\Media\`

```
~~1464~~ KERNEL32.dll!LoadLibraryA
    arg 0: C:\Users\Public\Media\vdeis2.dll (type=char*, size=0x0)
```

Figure 11. The path to DLL where the sample wants to find it

If we place the DLL in the right directory, the malware initiates a search queue where it scans for specific window class names that represent targeted web browsing applications, such as Internet Explorer (see Figure 12).

```
~~2556~~ USER32.dll!GetForegroundWindow
    arg 0: 0x0022fbbc (type=void, size=0x0)
    and return to module id:32, offset:0x291b60
~~2556~~ ntdll.dll!KiFastSystemCall
    arg 0: 0x01741b60
    arg 1: 0x0022fbbc
    and return to module id:10, offset:0x13369
~~2556~~ ntdll.dll!KiFastSystemCallRet
    arg 0: 0x01741b60
    arg 1: 0x0022fbbc
    and return to module id:10, offset:0x13369
~~2556~~ USER32.dll!GetClassNameW
    arg 0: 0x0009014a (type=<unknown>, size=0x0)
    arg 2: 0x400 (type=int, size=0x4)
    and return to module id:32, offset:0x28641b
~~2556~~ ntdll.dll!KiFastSystemCall
    arg 0: 0x777b2a4d
    arg 1: 0x0009014a
    and return to module id:10, offset:0x11b6c
~~2556~~ ntdll.dll!KiFastSystemCallRet
    arg 0: 0x777b2a4d
    arg 1: 0x0009014a
    and return to module id:10, offset:0x11b6c
~~2556~~ USER32.dll!CharUpperBuffW
    arg 0: PROCEXPL (type=wchar_t*, size=0x0)
    arg 1: 0x8 (type=DWORD, size=0x4)
    and return to module id:32, offset:0x219d4
```

```
~~2556~~ USER32.dll!GetForegroundWindow
    arg 0: 0x0022fbbc (type=void, size=0x0)
    and return to module id:32, offset:0x291b60
~~2556~~ ntdll.dll!KiFastSystemCall
    arg 0: 0x01741b60
    arg 1: 0x0022fbbc
    and return to module id:10, offset:0x13369
~~2556~~ ntdll.dll!KiFastSystemCallRet
    arg 0: 0x01741b60
    arg 1: 0x0022fbbc
    and return to module id:10, offset:0x13369
~~2556~~ USER32.dll!GetClassNameW
    arg 0: 0x000c0318 (type=<unknown>, size=0x0)
    arg 2: 0x400 (type=int, size=0x4)
    and return to module id:32, offset:0x28641b
~~2556~~ ntdll.dll!KiFastSystemCall
    arg 0: 0x777b2a4d
    arg 1: 0x000c0318
    and return to module id:10, offset:0x11b6c
~~2556~~ ntdll.dll!KiFastSystemCallRet
    arg 0: 0x777b2a4d
    arg 1: 0x000c0318
    and return to module id:10, offset:0x11b6c
~~2556~~ USER32.dll!CharUpperBuffW
    arg 0: IEFrame (type=wchar_t*, size=0x0)
    arg 1: 0x7 (type=DWORD, size=0x4)
    and return to module id:32, offset:0x219d4
```

Figure 12. The malware scans for window classes appearing in the foreground, looking for specific class names related to web-browsers. Left example: Process Explorer in the foreground, right example: Internet Explorer in the foreground.

EmbusteBot checks for window classes of the top 3 most popular web-browsers, *Internet Explorer (IEFrame in Figure X), Google Chrome, Mozilla Firefox,* checking if any appear on the victim's screen's foreground.

The overall flow of events here is as follows:

1. Get handle of a foreground window.
2. Get class name of a foreground window.
3. Compare class name with decrypted strings:
    a. IEFRAME (Internet Explorer),
    b. CHROME_WIDGETWIN_1 (Google Chrome),
    c. MOZILLAWINDOWCLASS (Mozilla Firefox),
    d. SUNAWTFRAME (Java),
    e. APPLICATIONFRAMEWINDOW (Window 10 Applications),
    f. BUTTONCLASS, MAKROBROWSER (generic bundled Internet browsers).
4. If the class name contains one of the substrings, jump to step 5. If not, return to step 1 after a short pause.
5. Get text title of a foreground window.
6. Compare the title with an elaborate list of decrypted strings of 50 Brazilian bank names and banking web application names in the uppercase (CharUpperBuffW API call is used, see Figure 13).

```
~~2556~~ USER32.dll!GetWindowTextW
    arg 0: 0x000c0318 (type=<unknown>, size=0x0)
    arg 2: 0x200 (type=int, size=0x4)
    and return to module id:32, offset:0x291ccb
```

```
~~2556~~ USER32.dll!CharUpperBuffW
    arg 0: Bank name - P - Microsoft Internet Explorer - Windows Internet Explorer
    arg 1: 0x46 (type=DWORD, size=0x4)
    and return to module id:32, offset:0x219d4
```

Figure 13. EmbusteBot checks the title of a foreground window to match the names of targeted Brazilian banks

7. If the window title contains one of the above substrings and the activation file *171703.reg (depends on the current system date)* presented in C:\Users\Public\Media (Figure 14, activation File), the malware commences its malicious activity. If note, it returns to step 1 after a short pause.

```
~~2556~~ KERNEL32.dll!GetFileAttributesW
    arg 0: C:\Users\Public\Media\171703.reg (type=wchar_t*,
    and return to module id:32, offset:0x22c02
```

Figure 14. Activation file

Upon confirming that the victim is browsing their bank's website, and an active window was successfully matched with a target bank, EmbusteBot collects general information about the infected endpoint's OS (using API calls presented in Figure 15) and hardware environment in the following format:

```
MACHINE_NAME;Windows X Service Pack X(version X.X, BUILD XXXX XX-bit Edition)Disabled;XX-XX-XX-XX;Disabled;0.0.4
XX-XX-XX-XX is the MAC-address of a victim's machine.
```

```
~~2556~~ RPCRT4.dll!UuidCreateSequential
    arg 0: 0x0022faec
    arg 1: 0x0022fb34
    and return to module id:32, offset:0x28810d
~~2556~~ KERNELBASE.dll!InterlockedDecrement
    arg 0: 0x761172c8 => 0x270f (type=long*, size=0x4)
    and return to module id:4, offset:0x1b93c
~~2556~~ KERNEL32.dll!GetCPInfo
    arg 0: 0xfde9 (type=uint, size=0x4)
    and return to module id:32, offset:0x1f2c6c
~~2556~~ KERNELBASE.dll!GetCPInfo
    arg 0: 0xfde9 (type=uint, size=0x4)
    and return to module id:32, offset:0x1f2c6c
~~2556~~ KERNEL32.dll!WideCharToMultiByte
    arg 0: 0xfde9 (type=uint, size=0x4)
    arg 1: 0x0 (type=DWORD, size=0x4)
    arg 2: WIN7_X86_SP1;Windows 7 Service Pack 1 (Version 6.1, Build 7601, 32-bit Edition);Disabled;00-50-56-8B-FA-E7;Disabled;0.0.4
    arg 3: 0x79 (type=int, size=0x4)
    arg 5: 0x0 (type=int, size=0x4)
    and return to module id:32, offset:0xe67c
```

Figure 15. UuidCreateSequential[1] and the hardware environment of our sandbox which malware prepares for sending

[1] The fun fact about UuidCreateSequential is that the API call was introduced by Microsoft to allow creation of **UUID**s using the MAC address of a machine's Ethernet card. Thus, malware uses this API call to get a MAC address. It can be also used to detect VM (for example, VMware uses the Organizationally Unique Identifier (OUI) 00:50:56). The second fun fact, if we want to fool our malware and change MAC via the standard Windows interface it wouldn't work. For some reason (probably it uses some low-level interface to get MAC), UuidCreateSequential returns a real MAC-address of Ethernet card. Thus, we should change it using the interface provided by our VM.

Then, the sample installs a hook procedure that monitors low-level keyboard events (Figure 16) along with the screen capturing (Figure 17).

```
666323 ~~2556~~ USER32.dll!SetWindowsHookExW
666324     arg 0: 0xd (type=int, size=0x4)
666325     arg 1: 0x017332a8 (type=<unknown>, size=0x0)
666326     arg 2: 0x00400000 (type=<unknown>, size=0x0)
666327     arg 3: 0x0 (type=DWORD, size=0x4)
666328     and return to module id:32, offset:0x283262
```

*Figure 16. The sample installs hook to monitor low-level keyboard events (arg 0: 0xd = **WH_KEYBOARD_LL**)*

```
3539007 ~~2556~~ USER32.dll!GetDC
3539008     arg 0: <null> (type=<unknown>, size=0x0)
3539009     and return to module id:32, offset:0xfb266
3539018 ~~2556~~ GDI32.dll!CreateCompatibleDC
3539019     arg 0: 0x66010d48 (type=<unknown>, size=0x0)
3539020     and return to module id:32, offset:0xfb277
3539029 ~~2556~~ GDI32.dll!CreateCompatibleBitmap
3539030     arg 0: 0x66010d48 (type=<unknown>, size=0x0)
3539031     arg 1: 0x690 (type=int, size=0x4)
3539032     arg 2: 0x41a (type=int, size=0x4)
3539033     and return to module id:32, offset:0xfb2e7
3539077 ~~2556~~ GDI32.dll!SelectObject
3539078     arg 0: 0x27010f88 (type=<unknown>, size=0x0)
3539079     arg 1: 0x32050963 (type=<unknown>, size=0x0)
3539080     and return to module id:32, offset:0xfb6b2
3539089 ~~2556~~ GDI32.dll!BitBlt
3539090     arg 0: 0x0e010f38 (type=<unknown>, size=0x0)
3539091     arg 1: 0x0 (type=int, size=0x4)
3539092     arg 2: 0x0 (type=int, size=0x4)
3539093     arg 3: 0x690 (type=int, size=0x4)
3539094     arg 4: 0x41a (type=int, size=0x4)
3539095     arg 5: 0x27010f88 (type=<unknown>, size=0x0)
3539096     and return to module id:32, offset:0xfb73a
```

Figure 17. The sample takes a screenshot of the whole screen and performs a bit-block transfer of the color data corresponding to a rectangle of pixels from the specified source device context into a destination device context

Example 2. Gootkit

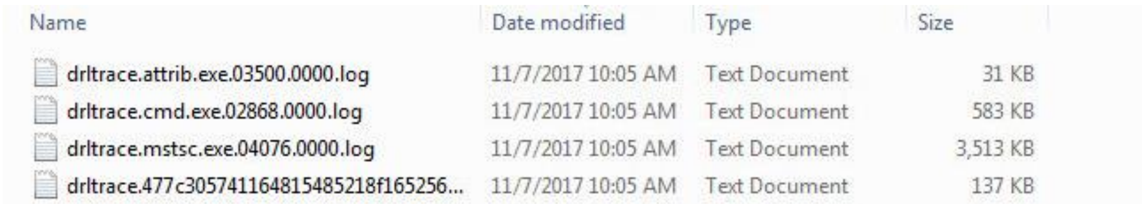
Discovered in the wild in the summer of 2014, GootKit is considered to be one of the more advanced banking Trojans active nowadays. It is used in online banking fraud attacks on consumer and business bank accounts mostly in European countries. GootKit is an ongoing malware project that implements advanced stealth and persistency alongside real time web-based activities like dynamic web injections that it can display directly in the infected machine's browser. GootKit affects the three most popular browsers: Internet Explorer, Mozilla Firefox, and Google Chrome.

We have undertaken analysis of the most current version of the GootKit dropper. Relying on DrLtrace we discovered new advanced anti-research capabilities which is used to bypass detection in virtual machines or sandboxes and improves Gootkit capabilities to ultimately infect more endpoints. Let's present how DrLtrace helped us in detection of these capabilities.

Starting the process:

```
drltrace.exe -logdir . -print_ret_addr --
477c305741164815485218f165256126bcd3fa6ce305f187aceb08cab89d8f01.exe
```

After several seconds, we see that the main process breaks up into three child processes. DrLtrace will follow child processes and provide logs for all of them (Figure 18).

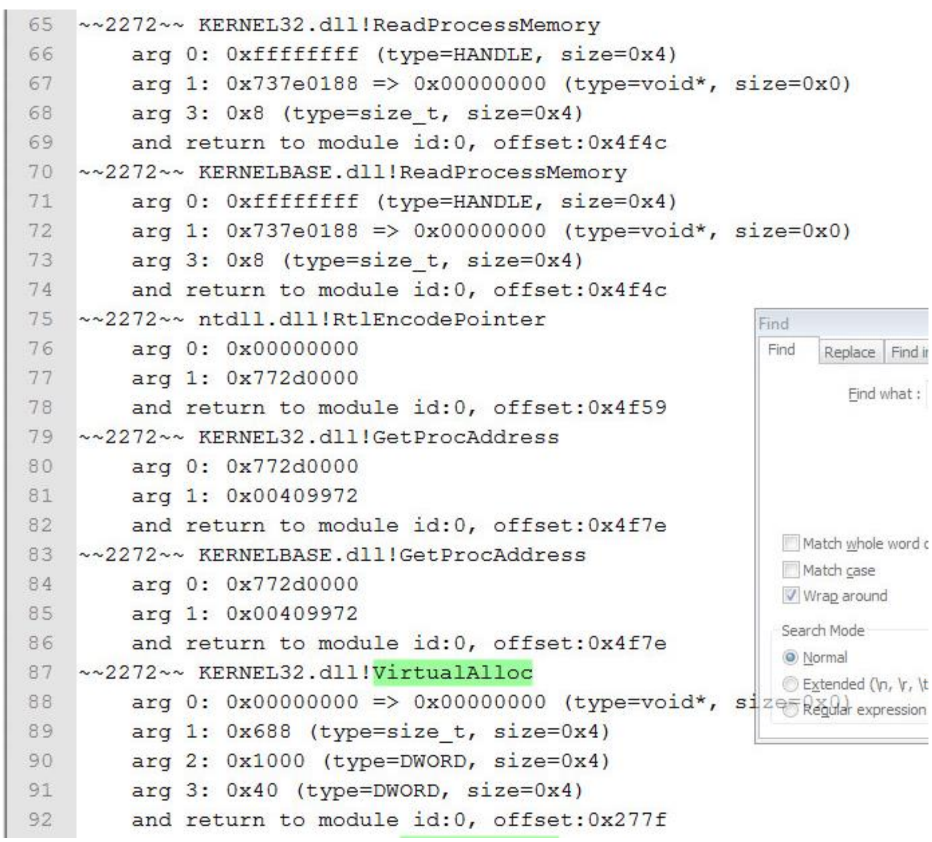


Name	Date modified	Type	Size
drltrace.attrib.exe.03500.0000.log	11/7/2017 10:05 AM	Text Document	31 KB
drltrace.cmd.exe.02868.0000.log	11/7/2017 10:05 AM	Text Document	583 KB
drltrace.mstsc.exe.04076.0000.log	11/7/2017 10:05 AM	Text Document	3,513 KB
drltrace.477c305741164815485218f165256...	11/7/2017 10:05 AM	Text Document	137 KB

Figure 18. DrLtrace generates four logs for the initial run of Gootkit

While files attrib.exe and cmd.exe looks like a standard windows processes, files `DrLtrace.mstsc.exe.04076.0000.log,drltrace.477c305741164815485218f165256126bcd3fa6ce305f187aceb08cab89d8f01.003376.000.log` looks suspicious. Let's look in the log file of the main process.

At the beginning we see Gootkit allocates memory with activated PAGE_EXECUTE permission. So, it looks like the sample is preparing place to unpack and execute some code.



```
~~2272~~ KERNEL32.dll!ReadProcessMemory
    arg 0: 0xffffffff (type=HANDLE, size=0x4)
    arg 1: 0x737e0188 => 0x00000000 (type=void*, size=0x0)
    arg 3: 0x8 (type=size_t, size=0x4)
    and return to module id:0, offset:0x4f4c
~~2272~~ KERNELBASE.dll!ReadProcessMemory
    arg 0: 0xffffffff (type=HANDLE, size=0x4)
    arg 1: 0x737e0188 => 0x00000000 (type=void*, size=0x0)
    arg 3: 0x8 (type=size_t, size=0x4)
    and return to module id:0, offset:0x4f4c
~~2272~~ ntdll.dll!RtlEncodePointer
    arg 0: 0x00000000
    arg 1: 0x772d0000
    and return to module id:0, offset:0x4f59
~~2272~~ KERNEL32.dll!GetProcAddress
    arg 0: 0x772d0000
    arg 1: 0x00409972
    and return to module id:0, offset:0x4f7e
~~2272~~ KERNELBASE.dll!GetProcAddress
    arg 0: 0x772d0000
    arg 1: 0x00409972
    and return to module id:0, offset:0x4f7e
~~2272~~ KERNEL32.dll!VirtualAlloc
    arg 0: 0x00000000 => 0x00000000 (type=void*, size=0x0)
    arg 1: 0x688 (type=size_t, size=0x4)
    arg 2: 0x1000 (type=DWORD, size=0x4)
    arg 3: 0x40 (type=DWORD, size=0x4)
    and return to module id:0, offset:0x277f
```


Figure 19. Gootkit dynamically allocates memory with PAGE_EXECUTE_READWRITE permission (arg 3: 0x40)

Next, the sample gets command line and compares itself name with mstsc.exe (standard Windows RDP client).

```
~~2272~~ KERNEL32.dll!GetCommandLineW
    arg 0: 0x00000001 (type=void, size=0x0)
    and return to module id:0, offset:0x1e13
~~2272~~ KERNELBASE.dll!GetCommandLineW
    arg 0: 0x00000001 (type=void, size=0x0)
    and return to module id:0, offset:0x1e13
~~2272~~ SHELL32.dll!CommandLineToArgvW
    arg 0: ..\..\..\477c305741164815485218f165256126bcd3fa6ce305f187aceb08cab89d8f01.exe
    and return to module id:0, offset:0x1e1a
~~2272~~ SHLWAPI.dll!StrStrIW
    arg 0: C:\Users\secuser\Desktop\477c305741164815485218f165256126bcd3fa6ce305f187aceb08cab89d8f01.exe
    arg 1: mstsc.exe (type=wchar_t*, size=0x0)
    and return to module id:0, offset:0x1ed0
```

Figure 20. Gootkit takes itself name from command line and compares with mstsc.exe

Few hundred API-calls after, we see CreateProcess API call where legitimate mstsc.exe is called with an argument that points to the path of our Gootkit.

```
~~2272~~ KERNEL32.dll!lstrcatW
    arg 0: C:\Windows\System32\mstsc.exe "C:\Users\secuser\Desktop\477c305741164815485218f165256126bcd3fa6ce305f187aceb08cab89d8f01.exe
    arg 1: " (type=wchar_t*, size=0x0)
    and return to module id:0, offset:0x9233
~~2272~~ KERNEL32.dll!CreateProcessW
    arg 0: <null> (type=wchar_t*, size=0x0)
    arg 1: C:\Windows\System32\mstsc.exe "C:\Users\secuser\Desktop\477c305741164815485218f165256126bcd3fa6ce305f187aceb08cab89d8f01.exe"
    arg 2: <null> (type=<unknown>*, size=0x0)
    arg 3: <null> (type=<unknown>*, size=0x0)
    arg 4: 0x0 (type=BOOL, size=0x4)
    arg 5: 0x800000c (type=DWORD, size=0x4)
    and return to module id:0, offset:0x9263
```

Figure 21. Gootkit starts a new process. Arg 5 equals CREATE_NO_WINDOW|CREATE_SUSPENDED|DETACHED_PROCESS

Since mstsc.exe was started suspended, we can expect process hollowing technique further. Figure 22 and 23 proves our guess. The sample takes context of a remote thread, creates a new section, maps the section in memory, writes the code in the remote process and resume the thread.

```
2777 ~~2272~~ KERNEL32.dll!GetThreadContext
2778     arg 0: 0x108 (type=HANDLE, size=0x4)
2779     arg 1: 0x0012fc30 (type=<unknown>*, size=0x0)
2780     and return to module id:0, offset:0x92a1
2893 ~~2272~~ ntdll.dll!ZwCreateSection
2894     arg 1: 0xf001f (type=unsigned int, size=0x4)
2895     arg 2: 0x0012fae8 (type=OBJECT_ATTRIBUTES*, size=0x4)
2896     arg 3: 0x0012fb10 (type=LARGE_INTEGER*, size=0x4)
2897     arg 4: 0x40 (type=unsigned int, size=0x4)
2898     arg 5: 0x8000000 (type=unsigned int, size=0x4)
2899     and return to module id:0, offset:0xa013
3463 ~~2272~~ ntdll.dll!ZwMapViewOfSection
3464     arg 0: 0x114 (type=HANDLE, size=0x4)
3465     arg 1: 0x10c (type=HANDLE, size=0x4)
3466     arg 2: 0x0012fbc0 => 0x00000000 (type=void **, size=0x4)
3467     arg 3: 0x0 (type=unsigned int, size=0x4)
3468     arg 4: 0x0 (type=unsigned int, size=0x4)
3469     arg 5: 0x0012fb24 (type=LARGE_INTEGER*, size=0x4)
3470     and return to module id:0, offset:0x9d42
```

Figure 22. Gootkit begins process hollowing

```
3721 ~~2272~~ KERNEL32.dll!WriteProcessMemory
3722     arg 0: 0x10c (type=HANDLE, size=0x4)
3723     arg 1: 0x00095ae2 => 0x00000000 (type=void*, size=0x0)
3724     arg 2: 0x0012fb80 => 0x00000000 (type=void*, size=0x0)
3725     arg 3: 0x1 (type=size_t, size=0x4)
3726     and return to module id:0, offset:0x94fc
3727 ~~2272~~ KERNELBASE.dll!WriteProcessMemory
3728     arg 0: 0x10c (type=HANDLE, size=0x4)
3729     arg 1: 0x00095ae2 => 0x00000000 (type=void*, size=0x0)
3730     arg 2: 0x0012fb80 => 0x00000000 (type=void*, size=0x0)
3731     arg 3: 0x1 (type=size_t, size=0x4)
3732     and return to module id:0, offset:0x94fc
3733 ~~2272~~ KERNEL32.dll!ResumeThread
3734     arg 0: 0x108 (type=HANDLE, size=0x4)
3735     and return to module id:0, offset:0x9512
```

Figure 23. Remote code injection in mstsc.exe

The rest of the main log file is not interesting. Let's switch to the log of mstsc.exe.

At the beginning, we can see our malware joking at us.

```
3306 ~~3852~~ KERNEL32.dll!GetEnvironmentVariableA
3307     arg 0: crackmeololo (type=char*, size=0x0)
3308     arg 2: 0x104 (type=DWORD, size=0x4)
```

Figure 24. Gootkit authors have sense of humor

After a few hundred lines, infected mstsc.exe starts a huge loop where it enumerates all the processes running in the memory looking for specific names for anti-research purposes (see our technical report for more details [12]).

```
~~3852~~ KERNEL32.dll!OpenProcess
    arg 0: 0x400 (type=DWORD, size=0x4)
    arg 1: 0x0 (type=BOOL, size=0x4)
    arg 2: 0xf4 (type=DWORD, size=0x4)

~~3852~~ KERNEL32.dll!WideCharToMultiByte
    arg 0: 0x0 (type=uint, size=0x4)
    arg 1: 0x0 (type=DWORD, size=0x4)
    arg 2: smss.exe (type=wchar_t*, size=0x0)
    arg 3: 0x8 (type=int, size=0x4)
    arg 5: 0x9 (type=int, size=0x4)
```

Figure 25. Gootkit enumerates process names running in the OS

After this long loop, the sample removes itself by dropping bat file 36197379.bat and executing it (our previous intuition about cmd.exe and attrib.exe was right).

```
~~3852~~ USER32.dll!wsprintfA
    arg 1: attrib -r -s -h %%1
:%u
del %%1
if exist %%1 goto %u
del %%0
```

Figure 26. Content of the bat file

```
~~3852~~ KERNELBASE.dll!ExpandEnvironmentStringsW
    arg 0: C:\Users\secuser\Desktop\36197379.bat (type=wchar_t*, size=0x0)
    arg 2: 0x26 (type=DWORD, size=0x4)

~~3852~~ KERNEL32.dll!CreateFileW
    arg 0: C:\Users\secuser\Desktop\36197379.bat (type=wchar_t*, size=0x0)
    arg 1: 0xc0000000 (type=DWORD, size=0x4)
    arg 2: 0x0 (type=DWORD, size=0x4)
    arg 3: <null> (type=<unknown>*, size=0x0)
    arg 4: 0x4 (type=DWORD, size=0x4)
    arg 5: 0x80 (type=DWORD, size=0x4)
~~3852~~ SHELL32.dll!ShellExecuteW
    arg 0: <null> (type=<unknown>, size=0x0)
    arg 1: open (type=wchar_t*, size=0x0)
    arg 2: C:\Users\secuser\Desktop\36197379.bat (type=wchar_t*, size=0x0)
    arg 3: "C:\Users\secuser\Desktop\477c305741164815485218f165256126bcd3fa6ce305f187aceb08cab89d8f01.exe"
    arg 4: <null> (type=wchar_t*, size=0x0)
    arg 5: 0x0 (type=int, size=0x4)
```

Figure 27. Creation and execution of the bat file

It looks like something is wrong with our environment and our "friend" doesn't want to work here. Let's look on output of strings filtering script supplied with DrLtrace (Figure 28).

```
    arg 0: %APPDATA%\Microsoft\Internet Explorer\ (type=wchar_t*, size=0x0)
    arg 0: %APPDATA%\Microsoft\Internet Explorer\ (type=wchar_t*, size=0x0)
    arg 0: %SystemRoot%\Tasks\ (type=wchar_t*, size=0x0)
    arg 0: %SystemRoot%\Tasks\ (type=wchar_t*, size=0x0)
    arg 1: C:\Users\Test\AppData\Roaming\Microsoft\Internet Explorer\ (type=wchar_t*,
    arg 2: ServiceEntryPointThread (type=wchar_t*, size=0x0)
    arg 2: ServiceEntryPointThread (type=wchar_t*, size=0x0)
```

Figure 28. A list of strings generated by filtering script

These strings are used only once in the log file (Figure 29), so, probably the sample compares them with something that was stored before in the memory. The rest of the file are not interesting for us. So, let's try to put our executable in one of these folders and execute it again.

```
~~3852~~ KERNEL32.dll!ExpandEnvironmentStringsW
    arg 0: %APPDATA%\Microsoft\Internet Explorer\ (type=wchar_t*, size=0x0)
    arg 2: 0x104 (type=DWORD, size=0x4)

~~3852~~ KERNELBASE.dll!ExpandEnvironmentStringsW
    arg 0: %APPDATA%\Microsoft\Internet Explorer\ (type=wchar_t*, size=0x0)
    arg 2: 0x104 (type=DWORD, size=0x4)

~~3852~~ KERNEL32.dll!ExpandEnvironmentStringsW
    arg 0: %SystemRoot%\Tasks\ (type=wchar_t*, size=0x0)
    arg 2: 0x104 (type=DWORD, size=0x4)
```

Figure 29. Gootkit expands two potentially interesting strings

In the newly produced log files, the sample again checks for %SystemRoot%\Tasks\ but since the process started from the right place, we can see more behavior. Now, we can see numerous anti-research checks:

1) **VideoBiosVersion** at HKEY_LOCAL_MACHINE\HARDWARE\DESCRIPTION\System\ looking for substring "VirtualBox" using RegOpenKey/RegQueryValueExW.
2) **SystemBiosVersion** at HKEY_LOCAL_MACHINE\HARDWARE\DESCRIPTION\System\ looking for substrings AMI, BOCHS, VBOX, QEMU, SMCI, INTEL–6040000 and FTNT-1 using RegOpenKey/RegQueryValueExW.
3) **DigitalProductId** at HKEY_LOCAL_MACHINE\Software\Microsoft\Windows NT\CurrentVersion looking for 55274-640-2673064-23950 (JoeBox) and 76487-644-3177037-23510 (CWSandbox) product ID keys using RegOpenKey/RegQueryValueExW.
4) **Current Windows user name** is compared with "CurrentUser" and "Sandbox" using GetUserNameA.
5) **Workstation name** is compared with ""SANDBOX" and "7SILVIA" using GetComputerNameA (we may guess that Gootkit's authors owns information about specific configuration details of a specific sandbox which they try to bypass).
6) To avoid execution on servers, the sample looks for "*Xeon*" substring at *ProcessorNameString*[2] (Figure 30).

---

[2]Registry key HKEY_LOCAL_MACHINE\HARDWARE\DESCRIPTION\System\CentralProcessor\0

```
~~1756~~ ADVAPI32.dll!RegOpenKeyW
    arg 0: 0x80000002 (type=<unknown>, size=0x0)
    arg 1: Hardware\DESCRIPTION\System\CentralProcessor\0
~~1756~~ ADVAPI32.dll!RegQueryValueExW
    arg 0: 0x00000118 (type=<unknown>, size=0x0)
    arg 1: ProcessorNameString (type=wchar_t*, size=0x0)
    arg 2: 0x00000000 (type=DWORD*, size=0x4)
    arg 5: 0x01b2f6d4 => 0x200 (type=DWORD*, size=0x4)
~~1756~~ SHLWAPI.dll!StrStrIW
    arg 0: Intel(R) CPU E5-2650 v3 @ 2.30GHz (type=wchar_t*,
    arg 1: Xeon (type=wchar_t*, size=0x0)
```

Figure 30. Gootkit checks for CPU name, comparing result with Xeon

If anti-research checks successfully pass, Gootkit connects to CnC server. We can clearly see API calls used for these purposes (Figure 31).

```
~~2840~~ KERNELBASE.dll!MultiByteToWideChar
    arg 0: 0x0 (type=uint, size=0x4)
    arg 1: 0x0 (type=DWORD, size=0x4)
    arg 2: susiku.info (type=char*, size=0x0)
    arg 3: 0xffffffff (type=int, size=0x4)
    arg 5: 0xc (type=int, size=0x4)

~~2840~~ WINHTTP.dll!WinHttpConnect
    arg 0: 0x003ca440 (type=<unknown>, size=0x0)
    arg 1: susiku.info (type=wchar_t*, size=0x0)
    arg 2: 0x00000050 (type=<unknown>, size=0x0)
    arg 3: 0x0 (type=DWORD, size=0x4)
~~2840~~ WINHTTP.dll!WinHttpOpenRequest
    arg 0: 0x004173a0 (type=<unknown>, size=0x0)
    arg 1: GET (type=wchar_t*, size=0x0)
    arg 2: /rbody320 (type=wchar_t*, size=0x0)
    arg 3: <null> (type=wchar_t*, size=0x0)
    arg 4: <null> (type=wchar_t*, size=0x0)
    arg 5: <null> (type=wchar_t*, size=0x0)

~~2840~~ KERNEL32.dll!GetProcessHeap
    arg 0: 0x00000000 (type=void, size=0x0)

~~2840~~ KERNELBASE.dll!GetProcessHeap
    arg 0: 0x00000000 (type=void, size=0x0)

~~2840~~ KERNEL32.dll!HeapFree
    arg 0: 0x3a0000 (type=HANDLE, size=0x4)
    arg 1: 0x0 (type=DWORD, size=0x4)
    arg 2: 0x003ca9f0 => 0x00000000 (type=void*, size=0x0)

~~2840~~ WINHTTP.dll!WinHttpGetIEProxyConfigForCurrentUser
    arg 0: 0x01b3f534 (type=<unknown>*, size=0x0)

~~2840~~ WINHTTP.dll!WinHttpCrackUrl
    arg 0: https://susiku.info:80/ (type=wchar_t*, size=0x0)
    arg 1: 0x17 (type=DWORD, size=0x4)
    arg 2: 0x0 (type=DWORD, size=0x4)
    arg 3: 0x01b3f2bc (type=<unknown>*, size=0x0)

~~2840~~ WINHTTP.dll!WinHttpOpen
    arg 0: <null> (type=wchar_t*, size=0x0)
    arg 1: 0x1 (type=DWORD, size=0x4)
    arg 2: <null> (type=wchar_t*, size=0x0)
    arg 3: <null> (type=wchar_t*, size=0x0)
    arg 4: 0x0 (type=DWORD, size=0x4)
```

Figure 31. Gootkit tries to establish connection with CnC

Since, the CnC is not available, we don't see downloading of an actual payload of our sample. The rest of the log file doesn't have interesting behavior.

Example 3. NotPetya

Inspired by WannaCry ransomware campaign, NotPetya was first discovered in June 2017 and attracted huge media attention. According to different data, it caused $892.5m [13] damage to the world economic. The authors employed two schemes of propagation: the same EternalBlue/EternalRomance exploits used in WannaCry along with embedded Mimikatz for stealing user and admin credentials.

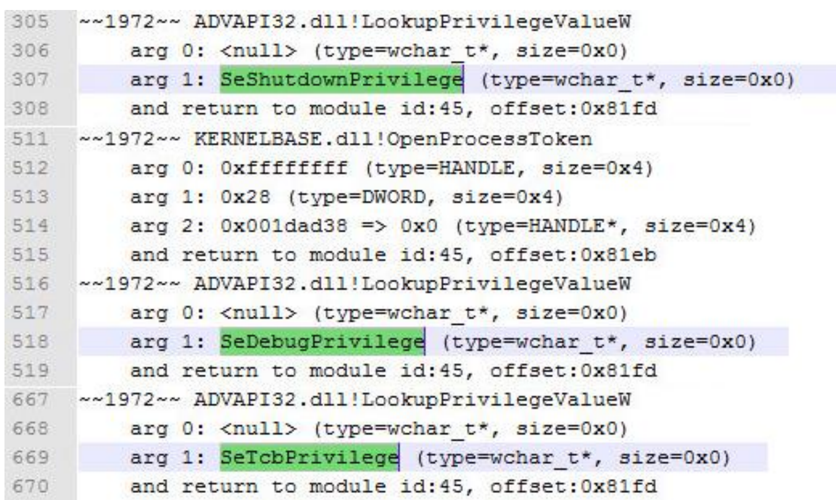
Let's try to get some technical details of this sample using DrLtrace:

```
drltrace.exe -logdir . -print_ret_addr – rundll32.exe perfc.dll,#1
```

The NotPetya main module is distributed as a DLL, we can simply execute it using rundll32.exe and apply an external script to select API calls that belongs to our DLL by specifying unique module id, listed in the module table at the end of the log file.

```
python filter_dlls.py drltrace.exe.rundll32.exe.00336.0000.log 45
```

The first part of the log, shows that NotPetya tries to adjust high level privileges (via AdjustTokenPrivileges API call) in the OS. The sample needs Shutdown, Debug and TCB privileges (to be able use low level OS features to re-write MBR).



```
~~1972~~ ADVAPI32.dll!LookupPrivilegeValueW
    arg 0: <null> (type=wchar_t*, size=0x0)
    arg 1: SeShutdownPrivilege (type=wchar_t*, size=0x0)
    and return to module id:45, offset:0x81fd
~~1972~~ KERNELBASE.dll!OpenProcessToken
    arg 0: 0xffffffff (type=HANDLE, size=0x4)
    arg 1: 0x28 (type=DWORD, size=0x4)
    arg 2: 0x001dad38 => 0x0 (type=HANDLE*, size=0x4)
    and return to module id:45, offset:0x81eb
~~1972~~ ADVAPI32.dll!LookupPrivilegeValueW
    arg 0: <null> (type=wchar_t*, size=0x0)
    arg 1: SeDebugPrivilege (type=wchar_t*, size=0x0)
    and return to module id:45, offset:0x81fd
~~1972~~ ADVAPI32.dll!LookupPrivilegeValueW
    arg 0: <null> (type=wchar_t*, size=0x0)
    arg 1: SeTcbPrivilege (type=wchar_t*, size=0x0)
    and return to module id:45, offset:0x81fd
```

Figure 32. NotPetya is looking for high privileges

The next part of the DrLtrace log file allows us to easily find a famous kill-switch for NotPetya. The sample will stop if `PathFileExistsW(C:\Windows\perfc.dll)` will return true.

```
~~1972~~ SHLWAPI.dll!PathFindFileNameW
    arg 0: C:\Users\secuser\Desktop\perfc.dll (type=wchar_t*,

~~1972~~ SHLWAPI.dll!PathCombineW
    arg 1: C:\Windows\ (type=wchar_t*, size=0x0)
    arg 2: perfc.dll (type=wchar_t*, size=0x0)

~~1972~~ SHLWAPI.dll!PathFindExtensionW
    arg 0: C:\Windows\perfc.dll (type=wchar_t*, size=0x0)

~~1972~~ SHLWAPI.dll!PathFileExistsW
    arg 0: C:\Windows\perfc (type=wchar_t*, size=0x0)

~~1972~~ KERNEL32.dll!CreateFileW
    arg 0: C:\Windows\perfc (type=wchar_t*, size=0x0)
    arg 1: 0x40000000 (type=DWORD, size=0x4)
    arg 2: 0x0 (type=DWORD, size=0x4)
    arg 3: <null> (type=<unknown>*, size=0x0)
    arg 4: 0x2 (type=DWORD, size=0x4)
    arg 5: 0x4000000 (type=DWORD, size=0x4)
```

Figure 33. NotPetya kill-switch shown by DrLtrace

The actual malicious behavior starts few hundred lines after kill-switch. The sample opens PhysicalDrive0 and overwrite MBR (Figure 34).

```
~~1972~~ KERNEL32.dll!CreateFileA
    arg 0: \\.\PhysicalDrive0 (type=char*, size=0x0)
    arg 1: 0xc0000000 (type=DWORD, size=0x4)
    arg 2: 0x3 (type=DWORD, size=0x4)
    arg 3: <null> (type=<unknown>*, size=0x0)
    arg 4: 0x3 (type=DWORD, size=0x4)
    arg 5: 0x0 (type=DWORD, size=0x4)

~~1972~~ KERNEL32.dll!SetFilePointerEx
    arg 0: 0x1a8 (type=HANDLE, size=0x4)
    arg 1: <null> (type=<unknown>, size=0x0)
    arg 3: 0x0 (type=DWORD, size=0x4)

~~1972~~ KERNEL32.dll!WriteFile
    arg 0: 0x1a8 (type=HANDLE, size=0x4)
    arg 1: 0x004484f8 => 0x00000000 (type=void*, size=0x0)
    arg 2: 0x200 (type=DWORD, size=0x4)
    arg 4: <null> (type=<unknown>*, size=0x0)
```

Figure 34. NotPetya opens PhysicalDrive0 and overwrites MBR

Moreover, NotPetya also encrypts all files stored in all subdirectories of all available drives in the OS starting from C:*. As DrLtrace shows in Figure X, the sample generates a key using standard Windows CryptoAPI and enumerates files via FindFirstFile/FindNextFile.

```
~~3940~~ ADVAPI32.dll!CryptAcquireContextW
    arg 1: <null> (type=wchar_t*, size=0x0)
    arg 2: Microsoft Enhanced RSA and AES Cryptographic Provider
    arg 3: 0x18 (type=DWORD, size=0x4)
    arg 4: 0xf0000000 (type=DWORD, size=0x4)
~~3940~~ ADVAPI32.dll!CryptGenKey
    arg 0: 0x00458188 (type=<unknown>, size=0x0)
    arg 1: 0x0000660e (type=<unknown>, size=0x0)
    arg 2: 0x1 (type=DWORD, size=0x4)
~~3940~~ KERNEL32.dll!FindFirstFileW
    arg 0: C:\* (type=wchar_t*, size=0x0)
~~3940~~ KERNEL32.dll!FindNextFileW
    arg 0: 0x45cac0 (type=HANDLE, size=0x4)
```

Figure 35. Generate a key and start searching for a target

Then if a file extension matches a certain pattern (Figure 36), the encryption takes place. The sample opens a file, maps it in the memory, encrypts it and saves it back on the disk. We can easily find which file extensions our sample wants to encrypt.

```
~~3940~~ SHLWAPI.dll!StrStrIW
    arg 0: .3ds.7z.accdb.ai.asp.aspx.avhd.back.bak.c.cfg.conf.cpp.cs.ctl.dbf.disk.djvu.doc.docx.dwg.eml.fdb.gz.
    h.hdd.kdbx.mail.mdb.msg.nrg.ora.ost.ova.ovf.pdf.php.pmf.ppt.pptx.pst.pvi.py.pyc.rar.rtf.sln.sql.tar.vbox.
    vbs.vcb.vdi.vfd.vmc.vmdk.vmsd.vmx.vsdx.vsv.work.xls.xlsx.xvd.zip.
    arg 1: .rar.

~~3940~~ KERNEL32.dll!CreateFileW
    arg 0: C:\$Recycle.Bin\S-1-5-21-3830209661-1978117751-3110444055-1001\cache.rar
    arg 1: 0xc0000000 (type=DWORD, size=0x4)
    arg 2: 0x0 (type=DWORD, size=0x4)
    arg 3: <null> (type=<unknown>*, size=0x0)
    arg 4: 0x3 (type=DWORD, size=0x4)
    arg 5: 0x0 (type=DWORD, size=0x4)

~~3940~~ KERNEL32.dll!GetFileSizeEx
    arg 0: 0x2f4 (type=HANDLE, size=0x4)

~~3940~~ KERNEL32.dll!CreateFileMappingW
    arg 0: 0x2f4 (type=HANDLE, size=0x4)
    arg 1: <null> (type=<unknown>*, size=0x0)
    arg 2: 0x4 (type=DWORD, size=0x4)
    arg 3: 0x0 (type=DWORD, size=0x4)
    arg 4: 0x230 (type=DWORD, size=0x4)
    arg 5: <null> (type=wchar_t*, size=0x0)

~~3940~~ KERNEL32.dll!MapViewOfFile
    arg 0: 0x2f8 (type=HANDLE, size=0x4)
    arg 1: 0x6 (type=DWORD, size=0x4)
    arg 2: 0x0 (type=DWORD, size=0x4)
    arg 3: 0x0 (type=DWORD, size=0x4)
    arg 4: 0x220 (type=size_t, size=0x4)

~~3940~~ ADVAPI32.dll!CryptEncrypt
    arg 0: 0x0045c9c0 (type=<unknown>, size=0x0)
    arg 1: <null> (type=<unknown>, size=0x0)
    arg 2: 0x1 (type=BOOL, size=0x4)
    arg 3: 0x0 (type=DWORD, size=0x4)
    arg 4: 0x00180000 => 0x1 (type=BYTE*, size=0x1)
    arg 5: 0x008ee390 => 0x220 (type=DWORD*, size=0x4)
```

Figure 36. NotPetya encryption APIs

Future work

1. While DrLtrace is not detectable by standard anti-research tricks, DBI-engine itself can be detected as shown in these works [14] [15]. Making DynamoRIO resistant against these tricks is important path for future work.

2. Currently, DrLtrace prints a raw log and provides several scripts to print important strings and library calls. In future, we plan to add heuristics (probably by applying YARA rules) to be able to select indicative behavior from malware automatically.

2. Currently, DynamoRIO has beta support of ARM architecture, testing and porting DrLtrace on ARM is required.

3. DrLtrace doesn't support situation when malware injects code in a remote process. In such cases, it is possible to tell DynamoRIO inject DrLtrace in all newly created processes (`-syswide_on` option of drrun.exe). However, in future, it is necessary to implement a special support in DrLtrace for such situations.

Conclusion

Thus, in our talk, we demonstrated that modern approaches are significantly limited by so-called semantic gap problem. While system-calls tracing allows us to trace executable without runtime overhead, we miss a lot of details required for efficient malware analysis. In turn, emulation can be used to achieve even full visibility towards malicious sample. However, the approach is extremely slow, and malware can easily take advantage from this.

Library calls tracing might be a reasonable trade-off; however, existent approaches usually modify code of system libraries which breaks transparency towards malware. We demonstrated that dynamic binary instrumentation easily solves this problem and can be used for malware analysis.

Based on dynamic binary instrumentation, we presented a novel solution for API calls tracing called DrLtrace. DrLtrace allowed us to revel in several minutes a lot of internal technical details about several sophisticated samples without even starting IDA or debugger.

DrLtrace is available at https://github.com/mxmssh/drltrace and distributed under BSD license.

References

[1] S. Mariani, L. Fontana, F. Gritti and S. D'Alessio., "PinDemonium. A DBI-based Generic Unpacker for Windows Executables.," in *BlackHat USA*, 2016.

[2] J. W. Oh, "Vulnerability Analysis and Practical Data Flow Analysis & Visualization," in *CanSecWest*, 2012.

[3] C. Kruegel, "Full system emulation: achieving successful automated dynamic analysis of evasive malware," in *BlackHat USA Security Conference Proceedings*, 2014.

[4] PANDA, "PANDA open-source repository," [Online]. Available: https://github.com/panda-re/panda.

[5] M. Lorenzo, "Testing System Virtual Machines," in *ACM Proceedings of The 19th International Symposium on Software Testing and Analysis*, 2010.

[6] Z. Liang, H. Yin and D. Song, "HookFinder: Identifiying and Understanding Malware Hooking Behavior," Carnegie Mellon University, 2008. [Online]. Available: http://repository.cmu.edu/cgi/viewcontent.cgi?article=1004&context=ece.

[7] L. Jennings, "API Hooking. Evading Traditional Detection with Stealthy New Techniques," InfoSecurity Magazine, 2016. [Online]. Available: https://www.infosecurity-magazine.com/opinions/api-hooking-evading-detection/.

[8] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation.," *ACM Sigplan Notices,* vol. 42, no. 6, 2007.

[9] C.-K. Luk and et.al, "Pin: building customized program analysis tools with dynamic instrumentation.," *Acm Sigplan Notices,* vol. 40, no. 6, 2005.

[10] D. Bruening, Efficient, transparent, and comprehensive runtime code manipulation., PhD Dissertation, Massachusetts Institute of Technology,, 2004.

[11] L. Kessem, "The Brazilian Malware Landscape a Dime a Dozen and Going Strong," July 2016. [Online]. Available: https://securityintelligence.com/the-brazilian-malware-landscape-a-dime-a-dozen-and-going-strong/.

[12] M. Shudrak, C. Eisner and L. Kessem, "Unraveling GootKit's Stealth Loader," 2017. [Online]. Available: https://securityintelligence.com/news/unraveling-gootkits-stealth-loader/.

[13] F. O'Connor, "NotPetya's Fiscal Impact Revised 892,5 Million and Growing," September 2017. [Online]. Available: https://www.cybereason.com/blog/blog-notpetyas-fiscal-impact-revised-892-5-million-and-growing.

[14] K. Sun, X. Li and Y. Ou, "Break Out of the Truman Show. Active Detection and Escape of Dynamic Binary Instrumentation.," in *Blackhat Asia*, 2016.

[15] F. Falcon and N. Riva, "Detecting Binary Instrumentation Frameworks.," in *ReCon*, 2012.