ILLUSION TRICKS:
Manipulating Sandbox Reports for Fun and Profit

Viviane Zwanger, Dr. Elmar Padilla

Fraunhofer-Institut für Kommunikation, Informationsverarbeitung und Ergonomie FKIE

Bonn, April 2018

Content

1
Introduction

Commercial sandbox systems are used for different purposes. Presumably well-known is their use in the field of malware analysis. Analysts run unknown malware, check what the malware is doing and analyze the observed behavior. Given this, analysts can conclude whether this is malware, the kind of malware, and maybe even the identification of the precise family. This is a very important and basic step in malware analysis.

However, sandbox systems are not only used in malware analysis. Several commercial sandbox systems are sold as solutions for a company's network to inspect traffic or as email filter system. Websites can be opened in advance in a sandbox with the company's default browser to see if anything suspicious happens, such as spawning unrelated processes and similar bad behavior. Files appended to emails can be opened with standard applications or run within the sandbox solution to see if bad things take place.

Surprisingly, everybody seems to trust sandbox systems, i.e. considers sandbox reports the truth. So it was a bit of a surprise (as malware analysts) when we found the reports might be faked without much effort, and do not necessarily mirror the truth.
Like any other tool, sandboxes can be fooled and circumvented by attackers. But still we were surprised discovering how easily some sandbox reports could be tricked into presenting false data. Thus, our conclusion is that one should not trust sandbox reports too easily – they might have been manipulated by the running malware. More generally spoken, sandboxes should be seen as one part of the solution when fighting against malware attacks, but not as the single solution.

1.1
Background

In the first days of October 2017, we wrote an internal tool to monitor, observe and change IOCTLs and their associated data[1] values used in device I/O to tell the kernel to execute certain functions without requiring a kernel driver. The idea was to catch the IOCTL codes originating from malware and change the associated data in input and output buffer in userspace.

The tool worked as intended, but as it was run in a sandbox, we observed that the report was wrong. It reported the original programmed actions the malware intended to do, but we knew it could not be true, since the actions had been changed in the meantime by our post-modification code and we were also able to observe the effect of our modification in reality, e.g., a connect to a server controlled by us or a file created on the hard disk.

---

[1] I/O Control Codes are numerical values used in device I/O to tell the kernel to execute certain mostly device-related functions.

We wrote several proof of concept binaries and began to test in a more systematic fashion, which will be detailed out in the following. We realized that this kind of post-modification might not affect only device I/O, but other types of actions such as opening, reading and writing files or read/write to registry. Later, we checked other commercial sandbox systems and positively found that a number of them were affected as well, where »affected« means the sandbox report lists actions that never happened, but does *not* list actions, which really happened. In other words, the exact opposite of a working sandbox.

1.2 Contacting the Vendors

We checked 8 sandbox vendors (BlueCoat/Symantec, VM-Ray, Lastline, Huawei, ThreatTrackSecurity/Vipre, VxStreams, JoeSandbox and Cuckoo as the only non-commercial sandbox) to which we had access to and reported them our findings on 11 October 2017.

4 sandbox vendors were vulnerable, one was partially affected, and 3 were unaffected. Only one of the vulnerable vendors fixed the bug in February 2018. We were also interested in testing the sandbox solutions of Fortinet, Barracuda, Fireeye, and Palo Alto, but when contacting them, they did not respond or were not interested in checking their sandbox solutions, insisting that their sandbox solutions worked correctly.

In all the time, we received very different reactions from the vendors and as mentioned, from others we never received any reply. We promised the contacted vendors to not divulge who was vulnerable and who not. We do publish our proof of concept code (see chapter 5 »Provided Material«) and we explain the technique. This makes it possible for everybody to take our code, run it on a sandbox and check the sandbox reports. Thus, everybody can check whether their sandbox is affected.

In the following, we explain how it works, present our two proof of concept samples. We also created several example reports with custom layout, the latter being due to anonymization. The report templates were created from scratch and do not disclose vendor identity, but contain real data.

2
How It Works

What we did is very simple and does not require expert knowledge. We let the application (or malware) make an API call. During the API call, code from several libraries is executed till ntdll. In ntdll, the corresponding system call number is pushed into the register and further execution passes into kernel space. We overwrite the code bytes in ntdll with a jump instruction, which later makes executaion flow jump to our post-modification code. The modification routine changes the parameters as desired, then restores the original bytes on ntdll and jumps back. Normal execution continues and it next passes into kernel space.

Several sandbox systems seem to capture the parameters of the API call only at the point where *the API function is initially called*. If something changes the original parameters unexpectedly during API call processing in the libraries, affected sandboxes won't notice and still print the original values. This is a bad mistake. It allows a malware essentially to control what is printed into the sandbox report.

2.1
Technical Implementation Details

In the following code example (see chapter 5 »Provided Material«), we let a malware/application connect to a fake ip address and a fake port. During API call processing, these parameters are changed post-hoc to the real IP address and the real port number. The actual code execution flow is shown in Fig. 1, together with a detailed walkthrough. The implementation details as given by the walkthrough are not needed for understanding, but to reproduce our proof of concept samples from source. For readers *not* interested in reproduction of our code, reading the walkthrough is not necessary.

main.c

```
// Create a socket

if((s = socket(AF_INET , SOCK_STREAM , 0 )) == INVALID_SOCKET)
{
    printf("Could not create socket : %d\n" , WSAGetLastError());
    return 1;
}

printf("Socket created.\n");

trap_setup(); // <== This sets up the trap for post-hoc modification

server.sin_family = AF_INET;

server.sin_addr.s_addr = inet_addr(fakeIP);    // uses fake IP address
server.sin_port = htons(fakePort);  // uses fake port number

// Connect to remote server
if (connect(s , (struct sockaddr *)&server , sizeof(server)) < 0)
{
    printf("Connect error.\n");
    return 1;
}

printf("Connected.\n");

[...]
```

In *main.c*, the program intends to connect to fakeIP and fakePort. The parameters will be changed to different values during the API call processing, e.g., to realIP and realPort. To achieve this, code in ntdll is modified. *trap_setup()* is a routine which sets up a shellcode trap triggered in ntdll (see trap_setup() pseudocode below). The 6 byte shellcode trap will be removed afterwards, which makes it a one-time event. Despite of using only a single call in the *main()*, the parameter-changing code can bootstrap itself multiple times afterwards by shifting portions of shellcode through various locations without requiring any new call in *main()*. This is further explained in Fig. 1 and walkthrough.

trap_setup()

```
// Prepare shellcode buffer. Buffer starts with 'ff 15' (call dword_address).
memcpy(&(buffer_shellcode[2]),&ptr_ephemeral_call);
// fills in address, which is the ephemeral ntdll twin function.
// Optional: do the same for the trap_setup shellcode.

// This saves the original 6 code bytes of ntdll function in ntdll to a buffer.
memcpy(buffer_org_xxfunction, targetedntdllfunction, 6);

Some virtual Protect (+RWE)
// make ntdll (and optionally winsock) writable.

// Overwrite original code bytes in ntdll with 6 byte shellcode call.
memcpy(targetedntdllfunction, buffer_shellcode, 6);

Set back Virtual Protect settings (original setting);

if called by shellcode: subtract 6 from return address;
```

After *trap_setup() is* called in *main()*, a 6 byte call instruction exists in the associated ntdll function, which in case of *winsock.connect()* would be *NtDeviceIoControlFile()*, a 10 parameter function. The target address of the call instruction points to our ephemeral post-modification routine, a twin of the correspondent ntdll function, which in our running example takes 11 (1+10) parameters. (See below for comparison).

`// Called in CDECL form.`
 	
`void __cdecl ephemeral_call`
`(`
`__in DWORD retAddr,`
`__in HANDLE FileHandle,`
`__in_opt HANDLE Event,`
`__in_opt PVOID ApcRoutine,`
`__in_opt PVOID ApcContext,`
`__out PIO_STATUS_BLOCK IoStatusBlock,`
`__in ULONG ControlCode,`
`__in PVOID InputBuffer,`
`__in ULONG InputBufferLength,`
`__out PVOID OutputBuffer,`
`__in ULONG OutputBufferLength`
`);`
	`DWORD WINAPI`
	`NtDeviceIoControlFile`
	`(`
	
	`__in HANDLE FileHandle,`
	`__in_opt HANDLE Event,`
	`__in_opt PVOID ApcRoutine,`
	`__in_opt PVOID ApcContext,`
	`__out PIO_STATUS_BLOCK IoStatusBlock,`
	`__in ULONG IoControlCode,`
	`__in PVOID InputBuffer,`
	`__in ULONG InputBufferLength,`
	`__out PVOID OutputBuffer,`
	`__in ULONG OutputBufferLength`
	`);`

Ephemeral twin function	NtDeviceIoControlFile

The ephemeral ntdll twin function can be generated automatically and adapts the typedef of its ntdll pendant function, with two notable exceptions (see comparison above): the first difference is the additional first parameter being the return address of the caller of the ntdll function, i.e. the caller of *NtDeviceIoControlFile()*. The second difference is the function is declared void return type. This allows for clean and elegant automatic stack restoration and works for any targeted ntdll function, such as NtCreateFile() or even ZwFsControlFile().

With the trap set up, the malware/application can now safely call its API functions on a vulnerable sandbox. At ntdll level, it will jump into the ephemeral ntdll twin function, where it changes the fake parameters into whatever real parameters are desired, returns and continues normal code execution including the now restored code bytes formerly overwritten by *trap_setup()*.

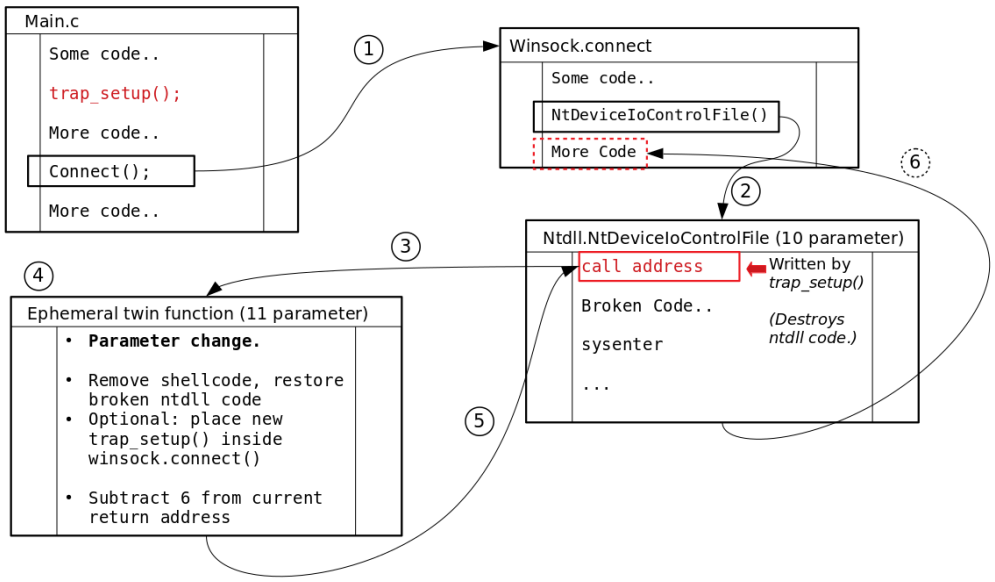

Fig. 1: Code execution flow. The action locations are referenced by numbers and explained in the walkthrough below.

Detailed Walkthrough

1. *Winsock.connect()* is called from *main()* with fake parameters.

2. *Winsock.connect()* calls *ntdll.NtDeviceIoControlFile().*

3. The 6-byte shellcode earlier placed by *trap_setup()* is executed.

4. The ephemeral twin function changes the parameters as desired. This is actually the easy part. Then, it needs to restore the original code bytes of ntdll before returning to the position where the 6 byte shellcode started, (i.e. return address – 6). Since the twin function has a compiler-generated stack frame, we can conveniently subtract the missing 6 bytes from EBP+4 (return address) before returning.

   Now comes the interesting part. Since the ntdll twin function is always declared *__cdecl* form and as a *void* returning function with **n+1** parameters, the return address to *winsock.connect()*, pushed on the stack when *NtDeviceIoControlFile()* was called, is interpreted as first input parameter, followed by the normal **n** parameters (i.e., 10 for *NtDeviceIoControlFile)* of the ntdll function. This is very important, as it serves two purposes.

   The first purpose is the function receives *all* n parameters of the original ntdll function. Mathematically, this combination also restores the stack to the state before the shellcode execution, as if our call never happened, except for the return address, which needs to be set back for 6 bytes. Thus, all that needs to be done in total is copying the 6 missing original ntdll bytes back into place and subtracting 6 from the return address on the stack (at EBP+4). After returning, no signs will be left and everything will be as if our actions never took place.

The second (optional) purpose takes effect when we want to change parameters a second (or third) time. Our function will use the first parameter to place a different 6 byte shellcode on the return code within *winsock.connect()*, which calls *trap_setup()*. This will trigger yet another parameter changing event the next time *connect()* is called by the program. If not for the optional feature, all that needs to be done in total is copying the 6 missing original ntdll bytes back into place and subtracting 6 from the return address at EBP+4.

5. The ntdll function, i.e. *NtDeviceIoControlFile()*, is executed with changed parameters and follows its normal execution path.

6. (Optional) If a new trap was set up, on return to the higher level API library, *winsock.connect()* in our running example, the 6 byte shellcode is executed and *trap_setup()* is called again, which places another 6 byte shellcode into the ntdll function … etc. etc. The *trap_setup()* function receives optional context knowledge saved by the ntdll twin function, which provides information about how it is being called, normally or by shellcode. If called by a 6 byte shellcode call, it will need to restore the stack in similar manner[2] as the ephemeral ntdll twin function. In case it was called by the *main()* as normal call, nothing needs to be done additionally.

On vulnerable sandboxes, malware can therefore essentially redefine what is written to the sandbox report.

For checking purposes, we created two different proof of concept binaries, which we offer along with their source code (see chapter 5 »Provided Material«). They can be executed in a sandbox to quickly check whether a given sandbox works correctly or is affected.

---

[2] Questionably, we resorted to using ESP to restore the stack. In principle, this would be avoidable.

2.2 Our Two Proof of Concept Samples

Our API tricksery would not work in a case where non-sandbox methods are used for generating the report data. Some sandboxes might use pcap data for network-related report data and some sandboxes might do a complete diff of the whole filesystem partition by comparing the starting snapshot against the resulting snapshot. Therefore, we provide at least two independent and different proof of concept examples for testing. The mentioned external methods of measurements must be kept in mind when looking at the reports of a sandbox, especially if only one of the proof of concept samples works.

Along with our proof of concept samples, we provide the *reference versions*, which do *no* post-modification at all (trap_setup() is disabled entirely). The reference version show what the report should have shown if the sandbox would have worked correctly. In source code, this is achieved by a REFERENCE precompiler switch, which can be set in *config.h*. The switch will also comment the post-modification related functions. There is no way of accidentally triggering the post-modification when compiled for the reference version.

Sample 1: Network I/O based Proof of Concept
In the default source code, the sample fake-connects to the official University Bonn website, later this is changed to the real IP and address of a Fraunhofer FKIE controlled server, which returns a small ASCII art depicting a cat.

The reference version directly connects to the Fraunhofer FKIE controlled server without detours.

Sample 2: File-based Proof of Concept
In the default source code, the sample fake-creates a file *e.txt*, which is later changed to *x.exe* during API call processing. The file in question is created in the Documents folder of the user's home directory.

In the reference version, file *x.exe* is created directly without detours.

We also created a *filesystem control based* proof of concept, but this might be too unintuitive for standard purposes. More important, many sandboxes miss filesystem control based actions entirely. It might be noteworthy to point out that by using filesystem control, many sandboxes (and possibly, AV software) are bypassed *per se*. In future, we might investigate this possibility. The mentioned sample is available on demand.

3
Reports

As mentioned, we promised to not publish any data stating, which vendor was vulnerable and which not. It is not trivial to show an example of a vulnerable report in such a case. We were able to create custom report templates and fill them with the real data. We publish these reports along with our technical report to give a better impression how a vulnerable report looks like and what to look for when checking a sandbox report.

A last thing to mention: concerning the vulnerable sandboxes, there were further surprises. When we tested different sandbox solutions, we found two commercial solutions, which were dubbed »super-vulnerable«. Not only did they print actions, which never happened, but it seems they also lost track of the further associated execution flow, causing the malware's follow-up actions to be missing in the report entirely. As an example, if we would change the IP address and port unexpectedly, they would lose track of the entire connection, such as data sent and received, whereas »default vulnerable« sandboxes would simply print wrong data in the report, which seems less bad.

The sandbox reports can be found in the appended material, together with the source code and compiled test binaries.

4
Final Words

We unintentionally discovered the possibility of easily manipulating the sandbox report as malware running on the sandbox. Our results indicate that putting too much trust in a sandbox might be a dangerous idea. We also suggest more public product tests of sandbox solutions as it was for example done in 2016[3].

Our testing was rather incomplete and only included 8 well-known sandbox vendors. However, half of the tested sandbox vendors were vulnerable to relatively simple manipulation. We have no idea whether malware authors already use this kind of exploit for hiding their actions and we did not search for samples in the wild. We furthermore do not know why some sandboxes are affected and others not, or what kind of logic exists behind the sandbox software, as we have no insights in the proprietary code of any commercial sandbox solution. We are however working on a fix for the open source Cuckoo sandbox.

---

[3] https://www.future-security.org/content/dam/future-security/de/doc/2016/FuSec2016_Proceedings-E-Book.pdf

5
Provided Material

We provide following material, which can be downloaded as zip archive at https://www.fkie.fraunhofer.de/content/dam/fkie/de/documents/Presse/AushebelnSandbox/sandbox_publication.zip.

Sha256:
e7d64828891448fabd2deb61752b468015be5b08c602209e3a69fda0ac6d0e64
(sandbox_publication.zip).

It contains:

1. A readme file, containing the password for the source code and binaries. The binaries might produce false alerts in antivirus programs, for example due to our ubiquitous VirtualProtect's against ntdll.

2. The compiled executables for checking the sandbox.

3. The complete source code for building the executables, which has been documented and referenced in this whitepaper. We provide a makefile for the WDK 7600, which uses the same (nmake) compiler as Visual Studio. Visual Studio can be used as well, but requires the source code to be imported into Visual Studio. Other than that, there should be no problems, especially since the WDK has been incorporated into Visual Studio recently. The file *config.h* contains the REFERENCE precompiler switch for either compiling the reference version or the »evil« version.

4. The artificial reports, provided to show the exact difference between a vulnerable sandbox and an unaffected sandbox.

6 Acknowledgements

We thank the Bundeswehr Cyber Security Centre for their support. We also thank our fellow worker Martin Clauß for many useful comments and suggestions as well as his pseudo C&C.