Full System Emulation:
Achieving Successful Automated Dynamic Analysis of Evasive Malware

Christopher Kruegel
Lastline, Inc.
chris@lastline.com

1 Introduction

Automated malware analysis systems (or sandboxes) are one of the latest weapons in the arsenal of security vendors. Such systems execute an unknown malware program in an instrumented environment and monitor their execution. While such systems have been used as part of the manual analysis process for a while, they are increasingly used as the core of automated detection processes. The advantage of the approach is clear: It is possible to identify previously unseen (zero day) malware, as the observed activity in the sandbox is used as the basis for detection. A key question is how to build an effective sandbox that reveals as many interesting malware behaviors as possible. We will address this point in the following section. Another question relates to the response from attackers and malware authors. Sandboxes are becoming increasingly popular. We all know that security is an arms race. So, how are the "bad guys" responding, and how can defenders, in turn, react to the trend of evasive malware that becomes increasingly popular.

2 Sandbox Goals and Design Options

A good sandbox has to achieve three goals: Visibility, resistance to detection, and scalability.

First, a sandbox has to see as much as possible of the execution of a program. Otherwise, it might miss relevant activity and cannot make solid deductions about the presence or absence of malicious behaviors. Second, a sandbox has to perform monitoring in a fashion that makes it difficult to detect. Otherwise, it is easy for malware to identify the presence of the sandbox and, in response, alter its behavior to evade detection. The third goal captures the desire to run many samples through a sandbox, in a way that the execution of one sample does not interfere with the execution of subsequent malware programs. Also, scalability means that it must be possible to analyze many samples in an automated fashion.

In this article, we discuss different ways in which a sandbox can monitor the execution of malware that runs in user mode (either as a regular user or administrator). This leaves out malicious code that tampers the kernel, such as rootkits. The vast majority of malware runs as regular user mode processes, and even rootkits typically leverage user mode components to install kernel drivers or to modify the operating system code.

When monitoring the behavior of a user mode process, almost all sandboxes look at the system call interface or the Windows API. System calls are functions that the operating system exposes to user mode processes so that they can interact with their environment and get stuff done, such as reading from files, sending packets over the network, and reading a registry entry on Windows. Monitoring system calls (and Windows API function calls) makes sense, but it is only one piece of the puzzle. The problem is that a sandbox that monitors only such invocations is blind to everything that happens in between these calls. That is, a sandbox might see that a malware program reads from a file, but it cannot determine how the malware actually processes the data that it has just read. A lot of interesting information can be gathered from looking deeper into the execution of a program. Thus, some sandboxes go one step further than just hooking function calls (such as system calls or Windows API functions), and also monitor the instructions that a program executes between these invocations.

Now that we know what information we want to collect, the next question is how we can build a sandbox that can collect this data in a way that makes it difficult for malware to detect. The two main options are *virtualization* and *emulation*.

2.1 Virtualization versus Emulation

An emulator is a software program that simulates the functionality of another program or a piece of hardware. Since an emulator implements functionality in software, it provides great flexibility. For example, consider an emulator that simulates the system hardware (such as the CPU and physical memory). When you run a guest program P on top of this emulated hardware, the system can collect very detailed information about the execution of P. The guest program might even be written for a different CPU architecture than the actual CPU that the emulator runs on. This mechanism allows, for example, to run an Android program, written for ARM, on top of an emulator that runs on an x86 host. The drawback of emulation is that the software layer incurs a performance penalty. The potential performance impact has to be carefully addressed to make the analysis system scalable.

With virtualization, the guest program P actually runs on the underlying hardware. The virtualization software (the hypervisor) only controls and mediates the accesses of different programs (or different virtual machines) to the underlying hardware. In this fashion, the different virtual machines are independent and isolated from each other. However, when a program in a virtual machine is executing, it is occupying the actual physical resources, and as a result, the hypervisor (and the malware analysis system) cannot run simultaneously. This makes detailed data collection challenging. Moreover, it is hard to entirely hide the hypervisor from the prying eyes of malware programs. The advantage is that programs in virtual machines can run at essentially native speed.

As mentioned previously, the task of an emulator is to provide a simulated (runtime) environment in which a malware program can execute. There are two main options for this environment. First, one can emulate the operating system (this is called OS emulation). Intuitively, this makes sense. A program runs in user mode and needs to make system calls to interact with its environment. So, why not simply emulate these systems calls? While the malware is running, one can get a close look at its activity (one can see every instruction). When the malware tries to make a system call, this information can be easily recorded. At this point, the emulator simply pretends that the system call was successfully executed and returns the proper result to the program.

This sounds simple enough in theory, but it is not quite as easy in practice. One problem is that the (native) system call interface in Windows is not documented, and Microsoft reserves the right to change it at will. Thus, an emulator would typically target the Windows API, a higher-level set of library functions on top of the native system calls. Unfortunately, there are tens of thousands of these Windows API functions. Moreover, the Windows OS is a huge piece of software, and emulating it faithfully requires an emulator that has a comparable complexity of Windows itself! Since faithful emulation is not practical, emulators typically focus on a popular subset of functionality that works "reasonably well" for most programs. Of course, malware authors know about this. They can simply invoke less frequently used functions and check whether the system behaves as expected (that is, like a real Windows OS). OS emulators invariably fail to behave as expected, and such sandboxes are quite easy for malware to detect and evade. Security vendors that leverage OS emulation are actually well aware of this limitations. They typically include OS emulation only as one part of their solution, complemented by other detection techniques.

As the second option for an emulator, one can simulate the hardware (in particular, CPU and physical memory). This is called (whole) system emulation. System emulation has several advantages. First, one can install and run an actual operating system on top of the emulator. Thus, the malware is executed inside a real OS, making the analysis environment much more difficult to detect for malware. The second advantage is that the interface offered by a processor is (much) simpler than the interface provided by Windows. Yes, there are hundreds of instructions, but they are very well documented, and they essentially never change. After all, Intel, AMD and ARM want an operating system (or application) developer to know exactly what to expect when she targets their platform. Finally, and most importantly, a system emulator has great visibility. A sandbox based on system emulation sees every instruction that a malware program executes on top of the emulated processor, and it can monitor every single access to emulated memory.

Virtualization platforms provides significantly fewer options for collecting detailed information. The easiest way is to record the system calls that programs perform. This can be done in two different ways. First, one could instrument the guest operating system. This has the obvious drawback that a malware program might be able to detect the modified OS environment. Alternatively, one can perform system call monitoring in the hypervisor. System calls are privileged operations. Thus, when a program in a guest VM performs such an operation, the hypervisor is notified. At this point, control passes back to the sandbox, which can then gather the desired data. The big challenge is that it is very hard to efficiently record the individual instructions that a guest process executes without being detected. After all, the sandbox relinquishes control to this process between the system calls. This is a fundamental limitation for any sandbox that uses virtualization technology.

2.2 Lastline's Full System Emulation Approach

We think that more visibility is better, especially facing malware that becomes increasingly aware of virtual machines and sandbox analysis. We have seen malware that tries to detect the presence of VMware for many years. Even if one builds a custom sandbox based on virtualization technology, the fundamental visibility limitations remain. Of course, when a malware program checks for specific files or processes that a well-known hypervisor like VMware introduces, these checks will fail, and the custom sandbox will be successful in seeing malicious activity. However, virtualization, by definition, means that malicious code is run directly on the underlying hardware. And while the malicious code is running, the sandbox is paused. It is only woken up at specific points, such as system calls. This is a problem, and a major reason why we decided to implement our sandbox as a system emulator.

Why does not everybody use system emulation, since it seems such a great idea? The reason is that one needs to overcome two technical challenges to make a system emulator work in practice. One challenge is called the semantic gap, the other one is performance. The semantic gap is related to the problem that a system emulator sees instructions executed on the CPU, as well as the physical memory that the guest OS uses. However, it is not immediately clear how to connect CPU instructions and bytes in memory to objects that make sense in the context of the guest OS. After all, we want to know about the files that a process creates, or the Windows registry entries that it reads. To bridge the semantic gap, one needs to gain a deep understanding into the inner workings of the guest operating system. By doing this, we can then map the detailed, low level view of our system to high level information about files, processes and network traffic that are shown in our report.

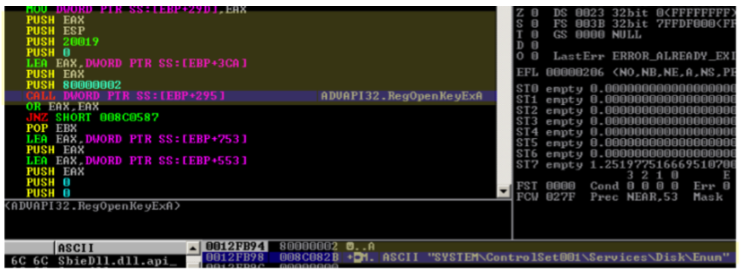
The second question is about performance. Isn't emulation terribly slow? The answer is yes, if implemented in a naive way. If we emulated every instruction in software, the system would indeed not scale very well. However, one can implement many clever things to speed up emulation, to a level where it is (almost) as fast as native execution. For example, one does not need to emulate all code. A lot of code can be trusted, such as Windows itself. Well, we can trust the kernel most of the time – of course, it can be compromised by rootkits. Only the malicious program (and code that this program interacts with) needs to be analyzed in detail. Also, one can perform dynamic translation. With dynamic translation, every instruction is examined in software once, and then translated into a much more efficient form that can be run directly.

3 Evasive Code

The fight against malicious code is an arms race. Whenever defenders introduce novel detection techniques, attackers strive to develop new ways to bypass them. Given the popularity of sandboxes, it is not surprising that malware authors have started to devise evasive techniques to ensure that their programs do not reveal any malicious activity when executed in such an automated analysis environment. Clearly, when malware does not show any unwanted activity during analysis, no detection is possible. Simple evasive techniques have been known for quite a while. For example, malware might check for the presence of a virtual machine, or it might query well-known Windows registry keys or files that reveal a particular sandbox. Other malware authors instructed their malware to sleep for a while, hoping that the sandbox would time out the analysis before anything interesting is happening.

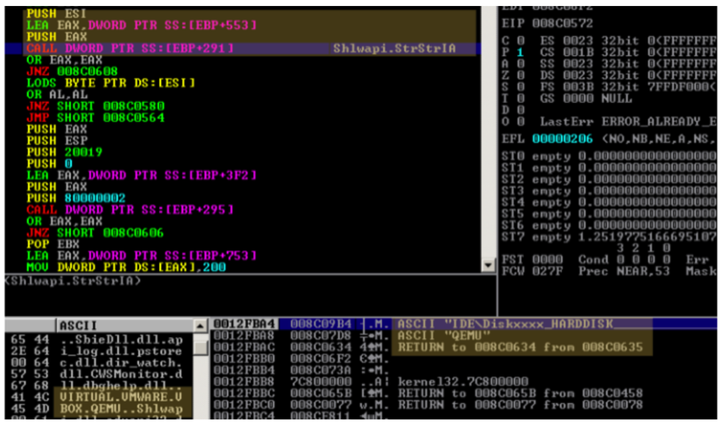
3.1 Environmental Checks

Malware programs frequently contain checks that determine whether certain files or directories exist on a machine and only run parts of their code when they do. Others require that a connection to the Internet is established or that a specific `mutex` object does not exist. In case these conditions are not met, the malware may terminate immediately. This is similar to malicious code that checks for indications of a virtual machine environment, modifying its behavior if such indications are present in order to make its analysis in a virtual environment more difficult. Other functionality that is not invoked on every run are malware routines that are only executed at or until a certain date or time of day. Functionality can also be triggered by other conditions, such as the name of the user or the IP address of the local network interface. Environmental checks have been discussed among security vendors in the past, and malware authors share well-known checks on hacker forums. As an example for such an environmental check, consider the example in Figure 1 and Figure 2. Here, we see malicious code querying for the names of the attached disks, and checking these names for the presence of the string `QEMU`. If this comparison is true, the malware knows that it is running inside the Qemu virtual environment.



```
MOV DWORD PTR SS:[EBP+29D],EAX
PUSH EAX
PUSH ESP
PUSH 20019
PUSH 0
LEA EAX,DWORD PTR SS:[EBP+3CA]
PUSH EAX
PUSH 80000002
CALL DWORD PTR SS:[EBP+295]                    ADVAPI32.RegOpenKeyExA
OR EAX,EAX
JNZ SHORT 008C0587
POP EBX
LEA EAX,DWORD PTR SS:[EBP+753]
PUSH EAX
LEA EAX,DWORD PTR SS:[EBP+553]
PUSH EAX
PUSH 0
PUSH 0
<ADVAPI32.RegOpenKeyExA>

Z 0   DS 0023 32bit 0(FFFFFFFF
S 0   FS 003B 32bit 7FFDF000(FF
T 0   GS 0000 NULL
D 0
O 0   LastErr ERROR_ALREADY_EXI
EFL 00000206 (NO,NB,NE,A,NS,PE
ST0 empty 0.0000000000000000000
ST1 empty 0.0000000000000000000
ST2 empty 0.0000000000000000000
ST3 empty 0.0000000000000000000
ST4 empty 0.0000000000000000000
ST5 empty 0.0000000000000000000
ST6 empty 0.0000000000000000000
ST7 empty 1.2519775166695107000
               3 2 1 0       E
FST 0000  Cond 0 0 0 0  Err 0
FCW 027F  Prec NEAR,53  Mask

      ASCII
6C 6C SbieDll.dll.api_
0012FB94  80000002 ☻..A
0012FB98  008C082B +☻M. ASCII "SYSTEM\ControlSet001\Services\Disk\Enum"
```

Figure 1: Malware enumerating registry keys for the disk.



```
PUSH ESI
LEA EAX,DWORD PTR SS:[EBP+553]
PUSH EAX
CALL DWORD PTR SS:[EBP+291]              Shlwapi.StrStrIA
OR EAX,EAX
JNZ 008C0608
LODS BYTE PTR DS:[ESI]
OR AL,AL
JNZ SHORT 008C0580
JMP SHORT 008C0564
PUSH EAX
PUSH ESP
PUSH 20019
PUSH 0
LEA EAX,DWORD PTR SS:[EBP+3F2]
PUSH EAX
PUSH 80000002
CALL DWORD PTR SS:[EBP+295]
OR EAX,EAX
JNZ SHORT 008C0606
POP EBX
LEA EAX,DWORD PTR SS:[EBP+753]
MOV DWORD PTR DS:[EAX],200
<Shlwapi.StrStrIA>

EIP 008C0572
C 0   ES 0023 32bit 0(FFFFFFF
P 1   CS 001B 32bit 0(FFFFFFF
A 0   SS 0023 32bit 0(FFFFFFF
Z 0   DS 0023 32bit 0(FFFFFFF
S 0   FS 003B 32bit 7FFDF000(
T 0   GS 0000 NULL
D 0
O 0   LastErr ERROR_ALREADY_E
EFL 00000206 (NO,NB,NE,A,NS,
ST0 empty 0.000000000000000000
ST1 empty 0.000000000000000000
ST2 empty 0.000000000000000000
ST3 empty 0.000000000000000000
ST4 empty 0.000000000000000000
ST5 empty 0.000000000000000000
ST6 empty 0.000000000000000000
ST7 empty 1.2519775166695107
               3 2 1 0
FST 0000  Cond 0 0 0 0  Err
FCW 027F  Prec NEAR,53  Mask

      ASCII
65 44 ..SbieDll.dll.ap
2E 64 i_log.dll.pstore
00 64 c.dll.dir_watch.
57 53 dll.CWSMonitor.d
67 68 ll.dbghelp.dll..
41 4C VIRTUAL.VMWARE.V
45 4D BOX.QEMU..Shlwap

0012FBA4  008C09B4  .M.  ASCII "IDE\Diskxxxx_HARDDISK
0012FBA8  008C07D8  •M.  ASCII "QEMU"
0012FBAC  008C0634  4•M. RETURN to 008C0634 from 008C0635
0012FBB0  008C06F2  €•M.
0012FBB4  008C073A  :•M.
0012FBB8  7C800000  ..A! kernel32.7C800000
0012FBBC  008C065B  [•M. RETURN to 008C065B from 008C0458
0012FBC0  008C0077  w.M. RETURN to 008C0077 from 008C0078
```

Figure 2: The same malware checking for the presence of QEMU in the disk name.

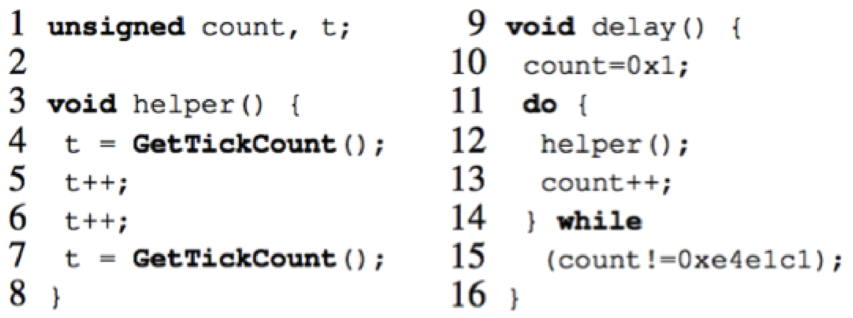
Environmental checks typically require that malware reads some value from the operating system (its runtime environment). These values can be registry keys, such as the name of the disks in the example above. Other values are file names or names of running processes. Whenever a malware program reads some value from the operating system, it has to invoke a system call. A sandbox sees this system call, and hence, can manipulate the return value (typically, it is randomized). Thus, when a specific environmental check becomes known, security vendors can improve their sandbox to watch for it. While this reactive approach works to some extend, it is vulnerable to evasion when malware authors introduce novel (zero day) checks. This requires vendors to patch their sandbox, introducing a window of vulnerability.

To handle the problem of environmental checks, it is crucial to have a more detailed view into the execution of a malware program. In particular, it is necessary to monitor the execution of all instructions. If such a view is available, it is possible to automatically track the values that a program reads and trace how the program processes it. This allows the system to recognize program points where the continuation of the execution depends on previously read input. When such a program point (an environmental check) is encountered, the analysis can explore both possible continuations. In addition, the system can extract the conditions under which the program follows a particular execution path. Using this information, one can determine the circumstances under which a damage routine or a propagation function is executed. This allows the automated identification and bypass of environmental checks, irrespective of the actual check that is used. For an academic writeup of some ideas behind multi-path exploration, check [2].

3.2 Stalling Code

Stalling code is executed before any malicious behavior – regardless of the execution environment. The purpose of such evasive code is to delay the execution of malicious activity long enough so that automated analysis systems give up on a sample, incorrectly assuming that the program is non-functional, or does not execute any action of interest. It is important to observe that the problem of stalling code affects all analysis systems, even those that are fully transparent. Moreover, stalling code does not have to perform any checks.

Stalling code exploits two common properties of automated malware analysis systems: First, the time that a system can spend to execute a single sample is limited. Typically, an automated malware analysis system will terminate the analysis of a sample after several minutes. This is because the system has to make a trade-off between the information that can be obtained from a single sample, and the total number of samples that can be analyzed every day. Second, malware authors can craft their code so that the execution takes much longer inside the analysis environment than on an actual victim host. Thus, even though a sample might stall and not execute any malicious activity in an analysis environment for a long time (many minutes), the delay perceived on the victim host is only a few seconds. This is important because malware authors consider delays on a victim's machine as risky. The reason is that the malicious process is more likely to be detected or terminated by anti-virus software, an attentive user, or a system reboot.



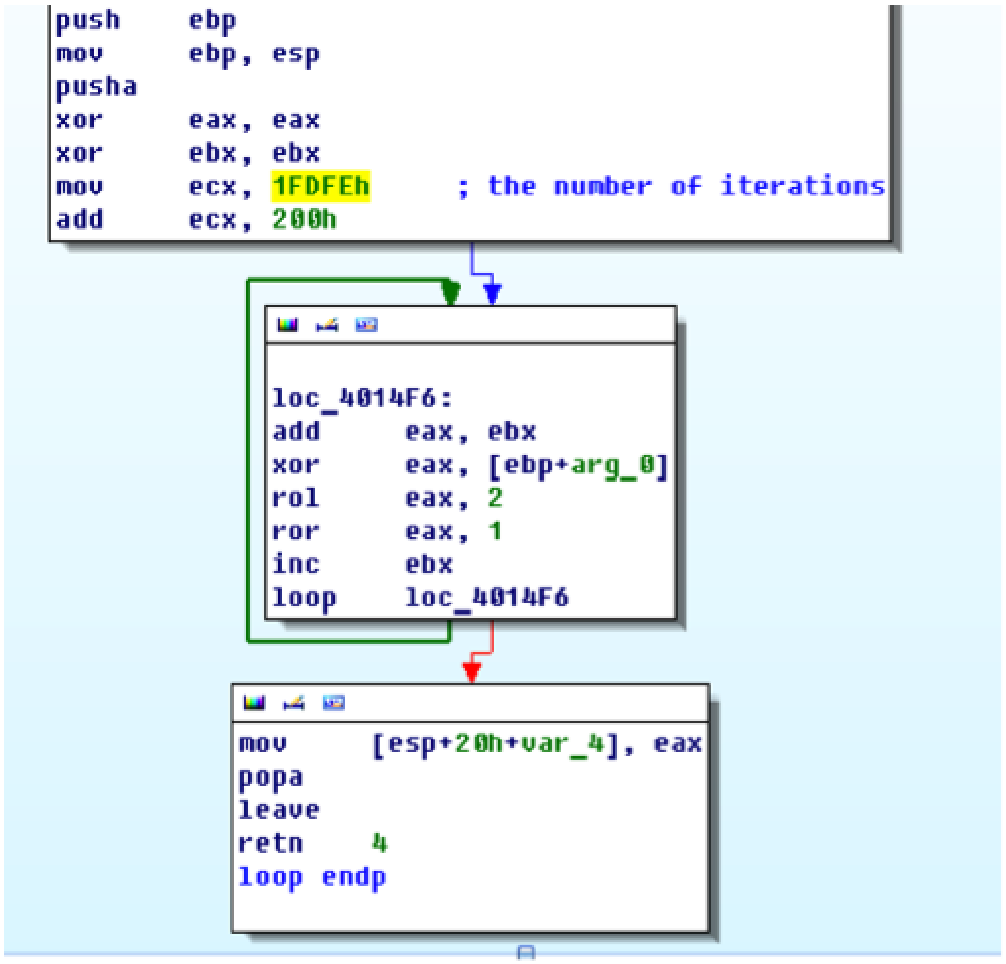
```
unsigned count, t;

void helper() {
 t = GetTickCount();
 t++;
 t++;
 t = GetTickCount();
}
void delay() {
 count=0x1;
 do {
  helper();
  count++;
 } while
  (count!=0xe4e1c1);
}
```

Figure 3: Stalling code in `W32.DelfInj`.

Figure 3 shows a stalling loop implemented by real-world malware. As the sample was only available in binary format, we reverse engineered the malware program and manually produced equivalent C code. Since the executable did not contain symbol information, we introduced names for variables and functions to make the code more readable. While this malware calls functions as part of the loop, this does not have to be the case (as shown in Figure 4).

Stalling code can only be recognized by analysis systems that have visibility into all instructions that a malware program executes. There are no obvious checks that can be seen at the system call level. To automatically detect stalling loops, and to ensure forward progress within the amount of time allocated for the analysis of a sample, one can use the following three-step approach.

First, we need techniques to detect when a malware sample is not making sufficient progress during analysis. When such a situation is encountered, the system can automatically examine the sample to identify the code regions that are likely responsible for stalling the execution. To this end, the system starts to dynamically record information about the addresses of instructions (code blocks) that are executed. Using these addresses, we build a (partial) control flow graph (CFG) of the non-progressing thread. This CFG is



```
push    ebp
mov     ebp, esp
pusha
xor     eax, eax
xor     ebx, ebx
mov     ecx, 1FDFEh       ; the number of iterations
add     ecx, 200h
```

```
loc_4014F6:
add     eax, ebx
xor     eax, [ebp+arg_0]
rol     eax, 2
ror     eax, 1
inc     ebx
loop    loc_4014F6
```

```
mov     [esp+20h+var_4], eax
popa
leave
retn    4
loop endp
```

Figure 4: Stalling loop without any function or system call.

then searched for loops. For these code regions (and these regions only), costly logging is disabled. When this is not sufficient, we force the execution to take a path that skips (exits) the previously identified stalling code. In that case, we need to be careful, since the program could be in an inconsistent state. Malware authors could leverage these inconsistencies to expose the analysis system. To overcome this problem, we mark variables that are touched by the loop as potentially inconsistent. When such a variable is later used, we compute the proposer value on demand, by extracting a program slice.

Again, the key insight that allows us to automatically detect and mitigate stalling code is the fact that we see all instructions that are executing. Hence, we can identify stalling loops and actively interrupt their execution. For an academic writeup of some ideas behind the detection and acceleration of stalling loops, check [1].

4 Summary

A sandbox offers the promise of zero day detection capabilities. As a result, most security vendors offer some kind of sandbox as part of their solutions. However, not all sandboxes are alike, and the challenge is not to build a sandbox, but rather to build a good one. Most sandboxes leverage virtualization and rely on system calls for their detection. We believe that this is not enough, since these tools fundamentally miss a significant amount of potentially relevant behaviors. Instead, a sandbox should be an analysis platform that sees all instructions that a malware program executes. This allows the system to see and react to attempts by malware authors to fingerprint and detect the runtime environment. It also enables a number of techniques to combat evasion approaches that rely on environment checks, triggers, and stalling loops.

References

[1] Clemens Kolbitsch, Engin Kirda, and Christopher Kruegel. The Power of Procrastination: Detection and Mitigation of Execution-Stalling Malicious Code. In *18th ACM Conference on Computer and Communications Security (CCS)*, 2011.

[2] Andreas Moser, Christopher Kruegel, and Engin Kirda. Exploring Multiple Execution Paths for Malware Analysis. In *IEEE Symposium on Security and Privacy*, 2007.