The Threat of Evasive Malware

Lastline Labs

labs@lastline.com

February 25, 2013

Key Points

- Malware authors exploit the limited visibility of traditional malware analysis systems (sandboxes) to evade detection
- Stalling code is a novel evasion technique that delays the malware execution without invoking any system calls, thereby timing out existing sandboxes
- The problem of evasive code is increasing

Introduction

The fight against malicious code is an arms race. Whenever defenders introduce novel detection techniques, attackers strive to develop new ways to bypass them. Automated malware analysis systems (or sandboxes) are one of the latest weapons in the defenders' arsenal. 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 core of the detection process. The advantage of the approach is clear: It is possible to identify previously unseen (zero day) malware, as only the observed activity in the sandbox is used for detection.

As dynamic analysis systems have become more popular, malware authors have responded by devising 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.

Security vendors reacted by adding some counter-intelligence of their own to their systems. They added hooks that would identify cases where malware queries for well-known keys, and they would force a program to wake up after it calls sleep. This approach worked reasonably well for a while, although it is fundamentally reactive in nature. That is, the malware analysis system needs to be manually updated to handle each new, evasive trick. As a result, malware authors who create zero day evasions can bypass detection until the sandbox is upgraded. Unfortunately, malware authors have recently introduced an evasive technique that can no longer be handled by current sandboxes (even if the trick is known). This new evasive technique, which we refer to as stalling code, delays the execution of malicious code so that a sandbox times out. However, to do this, the malware does not simply sleep. Instead, the program performs some (useless) computation that gives the appearance of activity. Hence, there is no way for the sandbox to wake up the program. Also, there are no checks for artifacts in the environment that might reveal any evasive behavior. The program simply executes, and from the point of view of the malware analysis system, everything is normal.

The key problem, and the reason for the fundamental limitation of current sandboxes, is their lack of visibility into the execution of a malware program. A good sandbox has to achieve two goals: Visibility and stealth. That is, a sandbox has to see as much as possible of the execution of a program. Moreover, it has to do this in a stealthy fashion. Otherwise, it is easy for malware to detect the presence of the sandbox and alter its behavior (as discussed above).

Current sandbox implementations typically rely on a virtual environment that contains the guest operating system. Sometimes, a sandbox runs the operating system directly on a real machine. The malware program is started inside the guest OS. To monitor a program's activity, a sandbox introduces hooks. These hooks can be inserted directly into a program to get notifications (callbacks) for function or library calls. The problem with direct hooks is that the program code needs to be modified, and this can be detect by malware or interfere with dynamic code generation (unpacking). Most frequently, sandboxes hook system calls to monitor the interaction between a program and the operating system. This is quite stealthy, especially for user-mode malware. Moreover, system calls capture all interactions between a program and its environment (e.g., when files are read, registry keys are written, and network traffic is produced). The key problem with hooking system calls (or library functions) is that the sandbox is blind to everything that happens in between calls. That is, a traditional sandbox cannot see any instruction that the malware executes between calls. This is a significant blind spot that malware authors can target; and they do so with stalling code, which is code that runs between system calls.

An alternative approach to increase the visibility into malicious code execution is to use a debugger. A debugger has the advantage that it can see every instruction that a program executes. Typically, these tools used by a human analyst to manually step through the code in order to understand its functionality. Unfortunately, debuggers are not stealthy, and malware programs already employ many techniques to detect running debuggers. Also, debuggers are mostly used for manual analysis, which does not scale to the number of samples that vendors receive.

What is needed is an automated malware analysis system that delivers the visibility of a debugger, but that is as stealthy and easy to manage as a virtual execution environment (a traditional sandbox). To achieve this goal, Lastline relies on system emulation. With a system emulator, we gain the advantages of a virtual execution environment, but retain the ability to see every instruction. This is crucial to be able to automatically handle evasive checks as well as stalling code. Figure 1 shows an overview of the ability of different malware analysis techniques. In the following sections, we discuss in more depth the two main evasive techniques that malware used in the wild: environmental checks and stalling code.

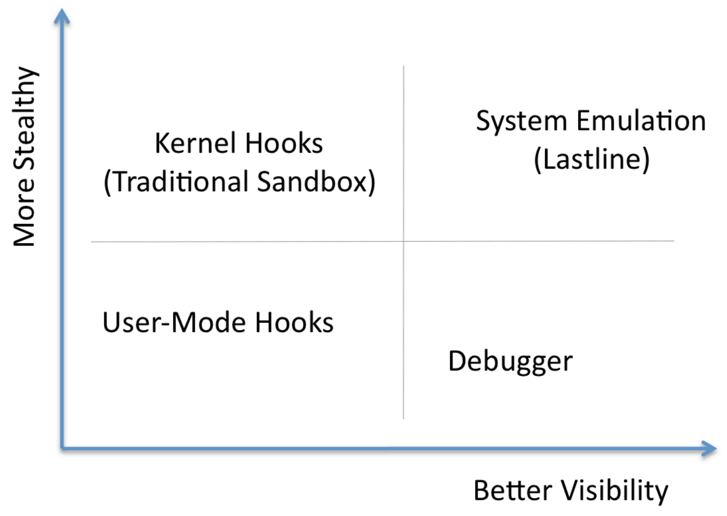
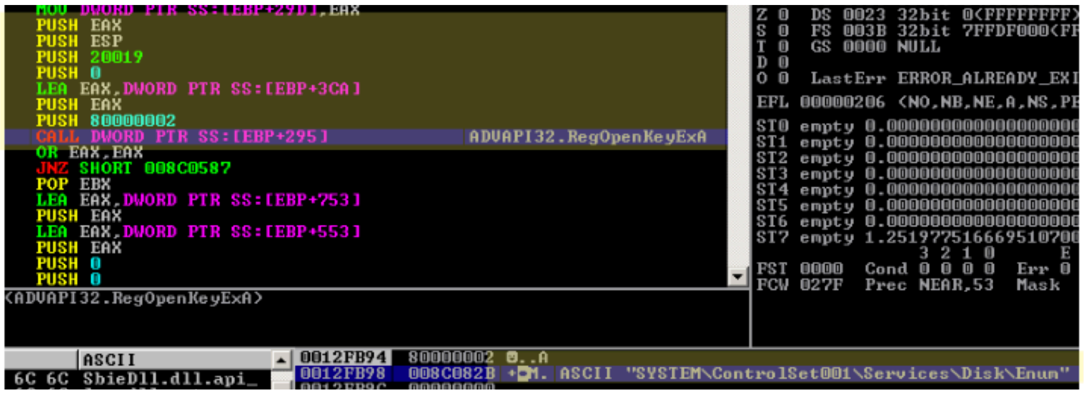

More Stealthy
Kernel Hooks (Traditional Sandbox)
System Emulation (Lastline)
User-Mode Hooks
Debugger
Better Visibility

Figure 1: Visibility versus stealth for different malware analysis approaches

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.

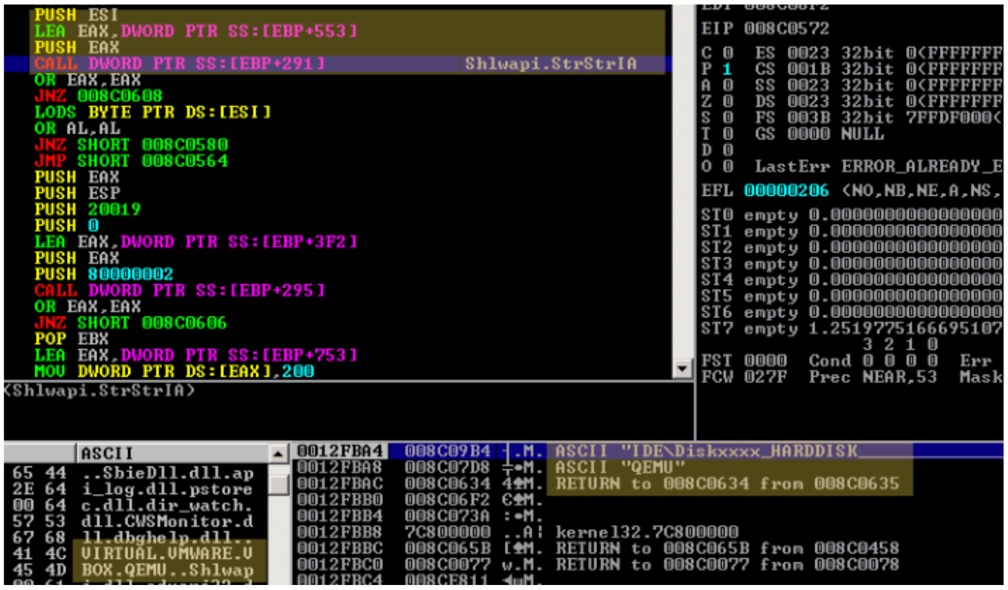
Environmental checks have been discussed among security vendors in the past, and malware authors share well-known checks on hacker forums. As an example for such an environmental check, consider the example in Figure 2 and Figure 3. Here, we see malicious code querying for the names of the attached disks, 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            Z 0  DS 0023 32bit 0(FFFFFFFF)
PUSH EAX                                  S 0  FS 003B 32bit 7FFDF000(FF
PUSH ESP                                  T 0  GS 0000 NULL
PUSH 20019                                D 0
PUSH 0                                    O 0  LastErr ERROR_ALREADY_EXI
LEA EAX,DWORD PTR SS:[EBP+3CA]
PUSH EAX                                  EFL 00000206 (NO,NB,NE,A,NS,PE
PUSH 80000002
CALL DWORD PTR SS:[EBP+295]     ADVAPI32.RegOpenKeyExA   ST0 empty 0.0000000000000000000
OR EAX,EAX                                ST1 empty 0.0000000000000000000
JNZ SHORT 008C0587                        ST2 empty 0.0000000000000000000
POP EBX                                   ST3 empty 0.0000000000000000000
LEA EAX,DWORD PTR SS:[EBP+753]            ST4 empty 0.0000000000000000000
PUSH EAX                                  ST5 empty 0.0000000000000000000
LEA EAX,DWORD PTR SS:[EBP+553]            ST6 empty 0.0000000000000000000
PUSH EAX                                  ST7 empty 1.2519775166695107000
PUSH 0                                                  3 2 1 0      E
PUSH 0                                    FST 0000  Cond 0 0 0 0  Err 0
(ADVAPI32.RegOpenKeyExA)                  FCW 027F  Prec NEAR,53  Mask

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

Figure 2: Malware enumerating registry keys for the disk



```
                                          EIP 008C0572
PUSH ESI
LEA EAX,DWORD PTR SS:[EBP+553]            C 0  ES 0023 32bit 0(FFFFFFFF
PUSH EAX                                  P 1  CS 001B 32bit 0(FFFFFFFF
CALL DWORD PTR SS:[EBP+291]     Shlwapi.StrStrIA   A 0  SS 0023 32bit 0(FFFFFFFF
OR EAX,EAX                                Z 0  DS 0023 32bit 0(FFFFFFFF
JNZ 008C0608                              S 0  FS 003B 32bit 7FFDF000(
LODS BYTE PTR DS:[ESI]                    T 0  GS 0000 NULL
OR AL,AL                                  D 0
JNZ SHORT 008C0580                        O 0  LastErr ERROR_ALREADY_E
JMP SHORT 008C0564
PUSH EAX                                  EFL 00000206 (NO,NB,NE,A,NS,
PUSH ESP
PUSH 20019                                ST0 empty 0.0000000000000000
PUSH 0                                    ST1 empty 0.0000000000000000
LEA EAX,DWORD PTR SS:[EBP+3F2]            ST2 empty 0.0000000000000000
PUSH EAX                                  ST3 empty 0.0000000000000000
PUSH 80000002                             ST4 empty 0.0000000000000000
CALL DWORD PTR SS:[EBP+295]               ST5 empty 0.0000000000000000
OR EAX,EAX                                ST6 empty 0.0000000000000000
JNZ SHORT 008C0606                        ST7 empty 1.2519775166695107
POP EBX                                                 3 2 1 0
LEA EAX,DWORD PTR SS:[EBP+753]            FST 0000  Cond 0 0 0 0  Err
MOV DWORD PTR DS:[EAX],200                FCW 027F  Prec NEAR,53  Mask
(Shlwapi.StrStrIA)

      ASCII               0012FBA4  008C09B4  .M. ASCII "IDE\Diskxxxx_HARDDISK_
65 44 ..SbieDll.dll.ap    0012FBA8  008C07D8  .M. ASCII "QEMU"
2E 64 i_log.dll.pstore    0012FBAC  008C0634  4M. RETURN to 008C0634 from 008C0635
00 64 c.dll.dir_watch.    0012FBB0  008C06F2  €M.
57 53 dll.CWSMonitor.d    0012FBB4  008C073A  :M.
67 68 ll.dbghelp.dll..    0012FBB8  7C800000  ..A| kernel32.7C800000
41 4C VIRTUAL.VMWARE.V    0012FBBC  008C065B  [M. RETURN to 008C065B from 008C0458
45 4D BOX.QEMU..Shlwap    0012FBC0  008C0077  w.M. RETURN to 008C0077 from 008C0078
```

Figure 3: 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.

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.

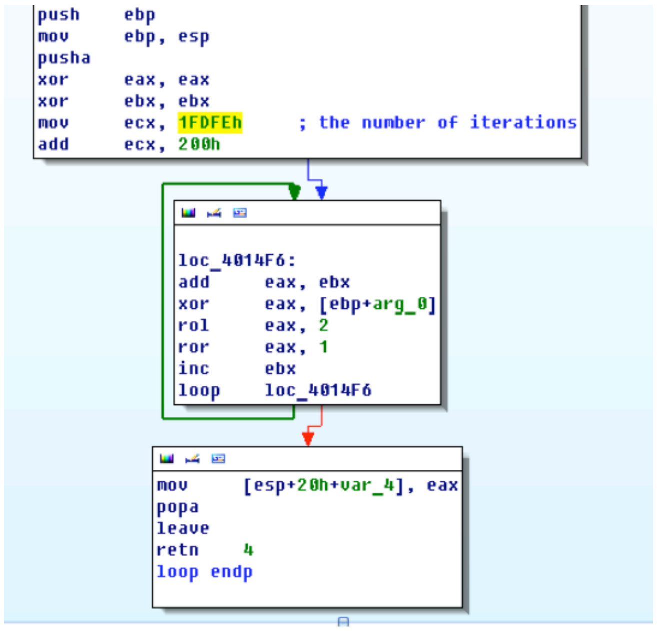
Stalling code exploits two common properties of automated malware analysis systems: First, the time that a system can spend to execute a single sample is limited. Typically, an automated malware analysis system will terminate the analysis of a sample after several minutes. This is because the system 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 4: Stalling code in W32.DelfInj

Figure 4 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 5).



```
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 5: Stalling loop without any function or system call

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.

To this end, we introduce techniques to detect when a malware sample is not making sufficient progress during analysis. When such a situation is encountered, our system automatically examines the sample to identify the code regions that are likely responsible for stalling the execution. To this end, our 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 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.

Summary and Trends

Malware authors have utilized simple evasion tricks for many years. These tricks mostly targeted public sandboxes and frustrated automated analysis by anti-virus vendors (who leverage sandboxes in the backend to aid signature generation). When sandboxes were introduced as the core of next generation APT detection offerings, attackers responded by developing novel evasive techniques, such as stalling code. Stalling code and novel environment triggers exploit the limited visibility of sandboxes and ensure that targeted attacks and zero day exploits remain successful. Figure 6 shows the fraction of samples with evasive behavior that we collected in our global analysis infrastructure over the last year. The growing trend is clearly visible.

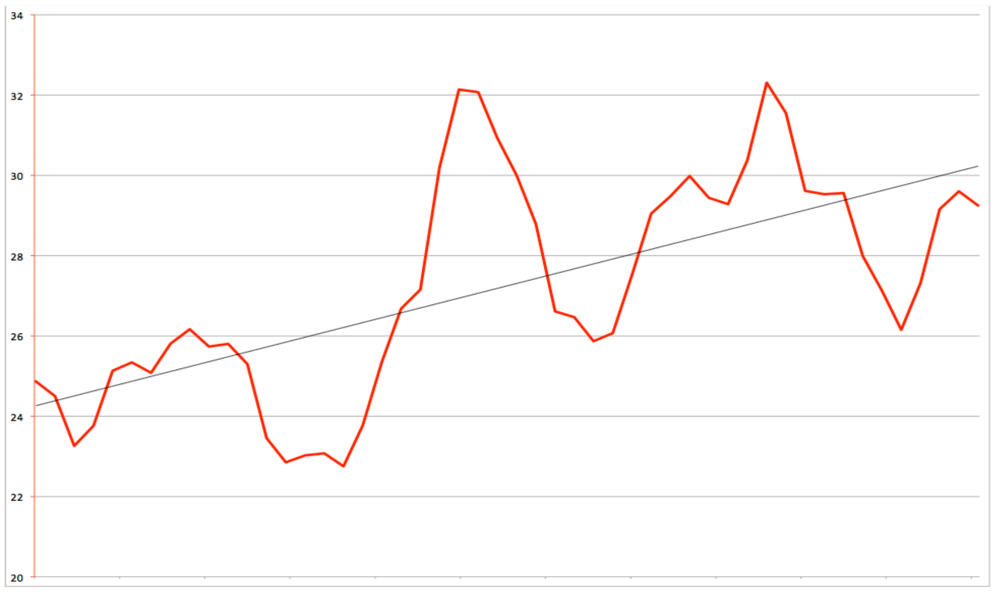


Figure 6: Increase of samples with evasive behaviors in 2012

Lastline, Inc.

Lastline, Inc., was founded in 2011 by university researchers Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Lastline's security products synthesize and bring to commercial standards the founders' award-winning, world-renowned academic research on malware analysis and attack countermeasures. The founders of Lastline are also the founders of iSecLab, one of the top malware research organizations in the world, and are considered to be today's thought leaders on automated high-resolution malware analysis and detection.

The founders are well-known for their development of Anubis and Wepawet, which are cloud-based malware analysis tools with a user base of thousands of corporations, government institutions, and security vendors.

Lastline is focused on real-time analysis of advanced malware and on tracking the Internet's malicious infrastructure (the Malscape™). Lastline leverages this threat intelligence to create advanced malware defenses for companies of all sizes.

By focusing on cloud-based automated systems and processes, Lastline has developed the technology to analyze advanced malware at an unprecedented speed and volume. This gives Lastline the ability to analyze binaries and web content as it enters enterprise networks, as well as the ability to map the Malscape™ at a level of accuracy and relevance previously not available. As a result, Lastline's technology is able to detect 0-day, targeted attacks and provides actionable threat intelligence to address the advanced malware problem.

For more information send email to: info@lastline.com

Lastline, Inc.
6950 Hollister Avenue, Suite 101, Goleta CA 93117 USA
lastline.com +1 (805) 456-7076