Just Carry A Ladder – Why Your EDR Let Pikabot Jump Through

vmray.com/cyber-security-blog/why-your-edr-let-pikabot-jump-through/

Just Carry A Ladder:

– Why Your EDR Let Pikabot Jump Through

Download full report
28.02.2024

Table of Contents

Overview

Pikabot has posed significant challenges to many Endpoint Detection and Response (EDR) systems through its employment of an advanced technique to hide its malicious activities known as “indirect system calls” (or “indirect syscalls”). This is only one of multiple techniques this family employs to evade detection: Pikabot distinguishes itself through the use of extensive obfuscation techniques such as inserting irrelevant junk code, hiding strings and even masquerading as benign applications by including their strings. Notably, the extent of obfuscation fluctuates among different samples, with recent instances showing less complexity. However, each variation consistently aims to evade detection by EDR systems.

Recent observations show an increase in the identification of Pikabot samples, a trend partially attributed to the activities of the threat actor group known as TA577, as such we believe it is important to delve into the sophisticated evasion tactics employed by this malware family, with a particular emphasis on its use of indirect syscalls.

You can read our analysis report here:

VMRay Threat Identifiers (19 rules, 72 matches)

	Score	Category	Operation
▸	5/5	Extracted Configuration	Pikabot configuration was extracted
▸	5/5	YARA	Malicious content matched by YARA rules
▸	5/5	Anti Analysis	Makes indirect system call to evade hooking based sandboxes
▸	4/5	Injection	Writes into the memory of another process
▸	4/5	Injection	Modifies control flow of another process
▸	4/5	Reputation	Malicious file detected via reputation
▸	4/5	Reputation	Malicious host or URL detected via reputation
▸	3/5	Network Connection	Uses HTTP to upload a large amount of data.
▸	2/5	Anti Analysis	Tries to detect kernel debugger
▸	2/5	Anti Analysis	Tries to detect debugger
▸	2/5	Discovery	Queries a host's domain name
▸	1/5	Discovery	Enumerates running processes
▸	1/5	Obfuscation	Reads from memory of another process
▸	1/5	Obfuscation	Creates a page with write and execute permissions
▸	1/5	Mutex	Creates mutex
▸	1/5	Network Connection	Tries to connect using an uncommon port
▸	1/5	Obfuscation	Resolves API functions dynamically
▸	1/5	Obfuscation	Overwrites code

Figure 1: Our dynamic, behavioral analysis reveals the malicious behavior of Pikabot.

Hooking

When Windows applications want to perform certain actions requiring interaction with the Windows kernel, for example to start a new process, the operation is similar to Figure 2:

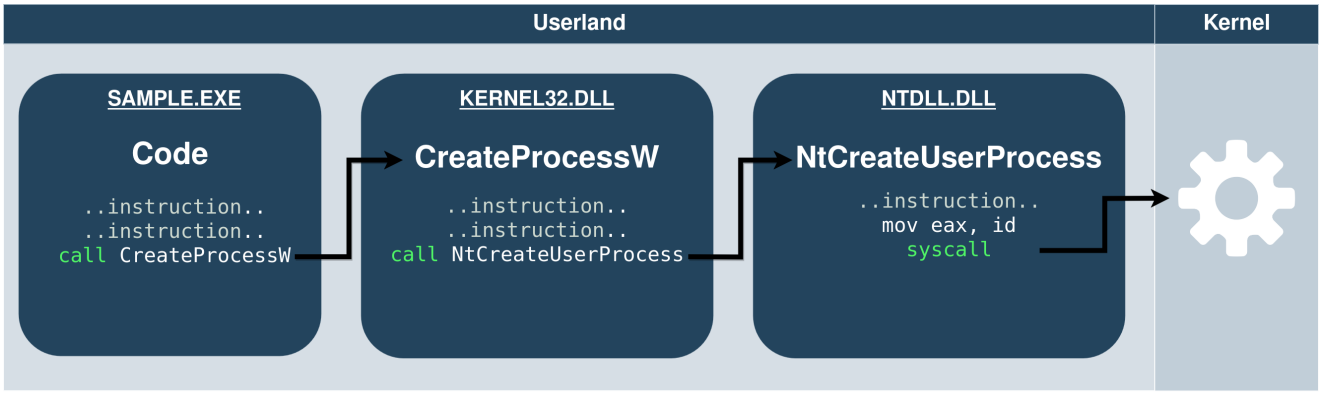

Figure 2: A simplified overview of the call sequence from sample to Windows kernel. Note that many details were excluded and adjusted for didactic purposes, e.g., kernelbase.dll was omitted for space purposes.

To monitor the behavior of applications, hooking-based EDR's and sandbox solutions insert hooks into Windows libraries or other process components of the malware, allowing them to intercept function calls and extract behavioral, runtime information. As an example, consider a script downloading an executable, placing it in the autostart folder and upon execution, writing code into the memory of another process. If this malware sample is new, traditional techniques based on signatures may fail to detect it, but the idea behind hooking-based EDR's is to capture the live behavior and determine whether the activities are benign or indicative of malicious intent.

However, Pikabot seeks to circumvent this surveillance by executing calls to the Windows kernel in a manner that avoids detection, thus concealing its malicious operations. The strategy involves a sophisticated approach of interacting with system calls to communicate with the Windows kernel indirectly.

In the following sections, we will explore the specifics of Pikabot's evasion techniques, including its use of indirect syscalls, and the broader implications of these tactics for cybersecurity defense strategies. Through this analysis, we aim to provide not only insights into past and current developments, but also how to combat future evasion techniques.

Bypassing Hooking

As malware becomes more sophisticated, so too do the methods it uses to evade detection.

Since the interception mechanism based on userland hooks operates within the malware's own process space, it is inherently more visible—and therefore detectable—by the malware. This visibility allows malware developers to devise methods specifically aimed at identifying

and circumventing these hooks.

Pikabot's evasion strategy exploits this vulnerability in hooking-based EDR systems. By bypassing these hooks, Pikabot can carry out its malicious activities without triggering the behavioral alarms that would normally alert the EDR to its presence. To demonstrate this, a simple EDR using hooks is highlighted in Figure 3:

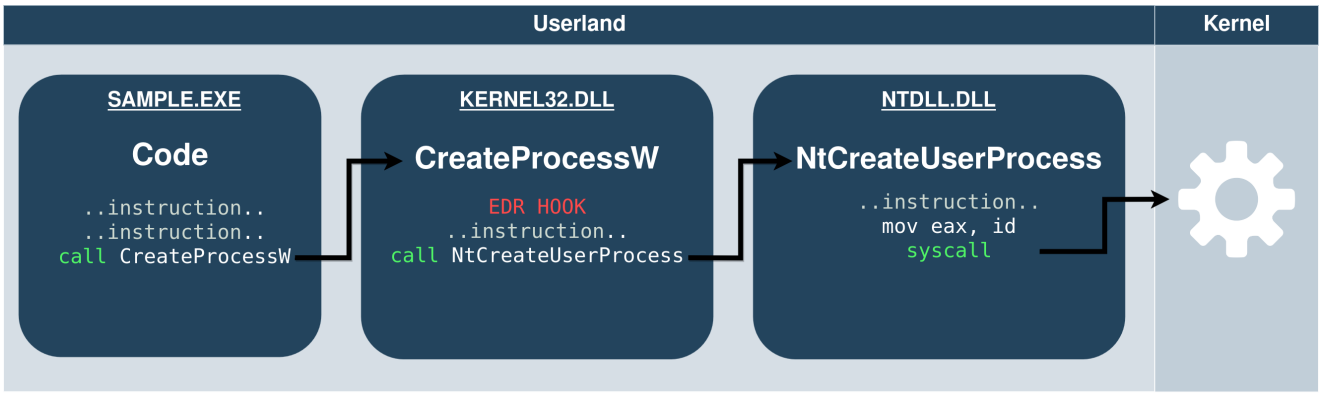

Figure 3: A simple user-land hook installed by an EDR to capture calls to CreateProcessW.

There have been a number of methods employed by malware to avoid detection by userland, hooking-based EDR's.

Detecting the Hooks

As most hooks are installed by redirecting the execution of the Windows API function by replacing the beginning of the function call, known as the prologue, with a jump to the monitoring code of the EDR (see Figure 4), in this method the malware inspects the bytes at the beginning of API call instructions.

```
00007FFDAA386A10   E9 839805C0          jmp 7FFD6A3E0298                LdrLoadDll
00007FFDAA386A15   CC                   int3
00007FFDAA386A16   57                   push rdi
00007FFDAA386A17   41:56                push r14
00007FFDAA386A19   48:81EC D0000000     sub rsp,D0
00007FFDAA386A20   48:8B05 E9DA1600     mov rax,qword ptr ds:[7FFDAA4F4510]
00007FFDAA386A27   48:33C4              xor rax,rsp
00007FFDAA386A2A   48:898424 C0000000   mov qword ptr ss:[rsp+C0],rax
00007FFDAA386A32   4D:8BF1              mov r14,r9
00007FFDAA386A35   49:8BF8              mov rdi,r8
00007FFDAA386A38   4C:8BD2              mov r10,rdx
00007FFDAA386A3B   48:8BF1              mov rsi,rcx
00007FFDAA386A3E   48:85D2              test rdx,rdx
00007FFDAA386A41   0F84 29010000        je ntdll.7FFDAA386B70
00007FFDAA386A47   8B02                 mov eax,dword ptr ds:[rdx]
00007FFDAA386A49   8B0A                 mov ecx,dword ptr ds:[rdx]
00007FFDAA386A4B   83E1 04              and ecx,4
00007FFDAA386A4E   03C9                 add ecx,ecx
00007FFDAA386A50   8BD1                 mov edx,ecx
00007FFDAA386A52   83CA 40              or edx,40
00007FFDAA386A55   24 02                and al,2
00007FFDAA386A57   41:8B02              mov eax,dword ptr ds:[r10]
00007FFDAA386A5A   0F44D1               cmove edx,ecx
00007FFDAA386A5D   44:8BC2              mov r8d,edx
```

Figure 4: Native function LdrLoadDll is hooked by an antivirus software via a jump instruction placed at the beginning.

By analyzing these CPU instructions, the malware can determine if they have been altered from their original state, which would indicate the presence of hooks inserted by an EDR system. This detection mechanism allows malware to identify and react to the presence of monitoring tools, thereby evading detection.

Removing the Hooks

Another approach taken by malware is to restore the original DLL code, effectively removing the hooks inserted by the EDR system. One such method is to create a new, clean copy of the library by re-loading it into memory, thereby eliminating the monitoring hooks (see Figure 5):

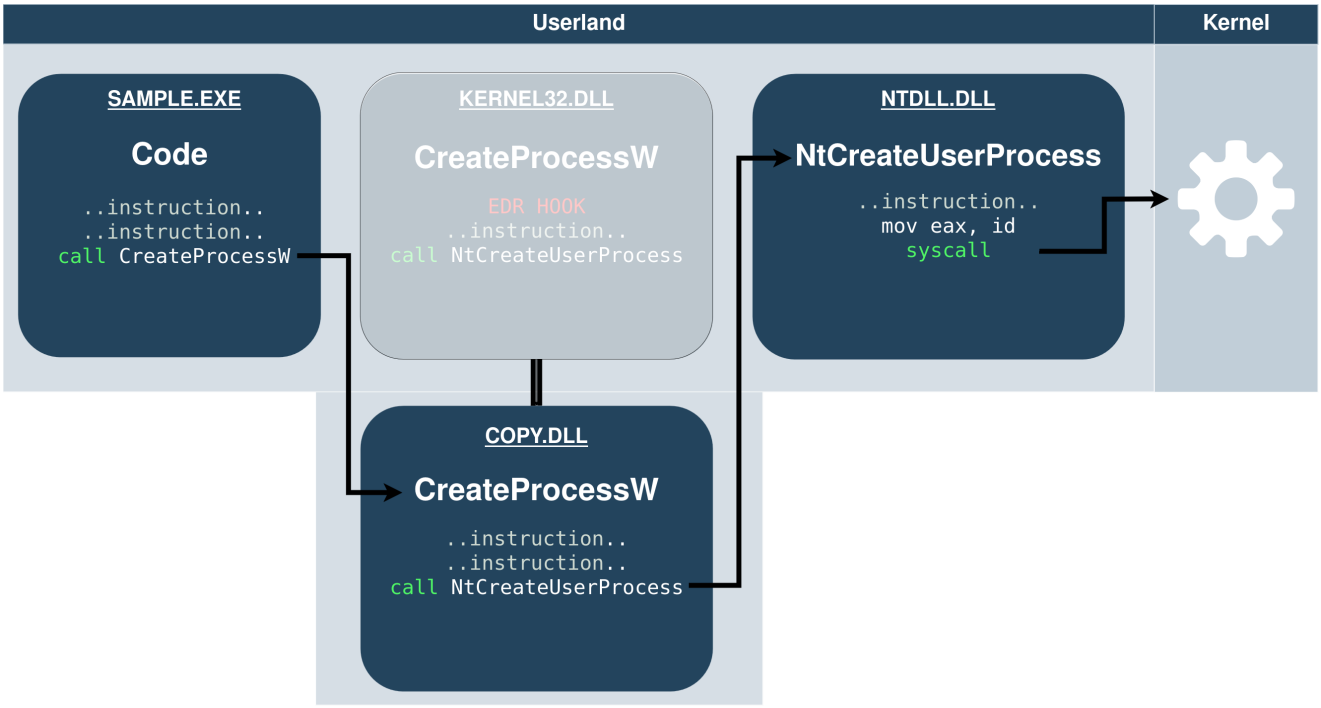

Figure 5: A technique involves loading a clean copy of a hooked DLL to evade the EDR hooking.

Calling Native Functions Directly

Some malware families, such as IcedID, employ an evasion technique based on directly invoking the undocumented native functions (such as NtCreateProcess), which are low-level system functions provided by the Windows Native API (see Figure 6). Microsoft did not intend these APIs to be called directly as they are supposed to be only used by Windows libraries internally, therefore they are not officially documented and subject to change, so developers are advised not to rely on them. But malware authors do usually not care about

longevity of their samples, so they call these functions directly to avoid the higher-level API functions that used to be more likely to be monitored and hooked by EDR systems, thereby sidestepping detection mechanisms.

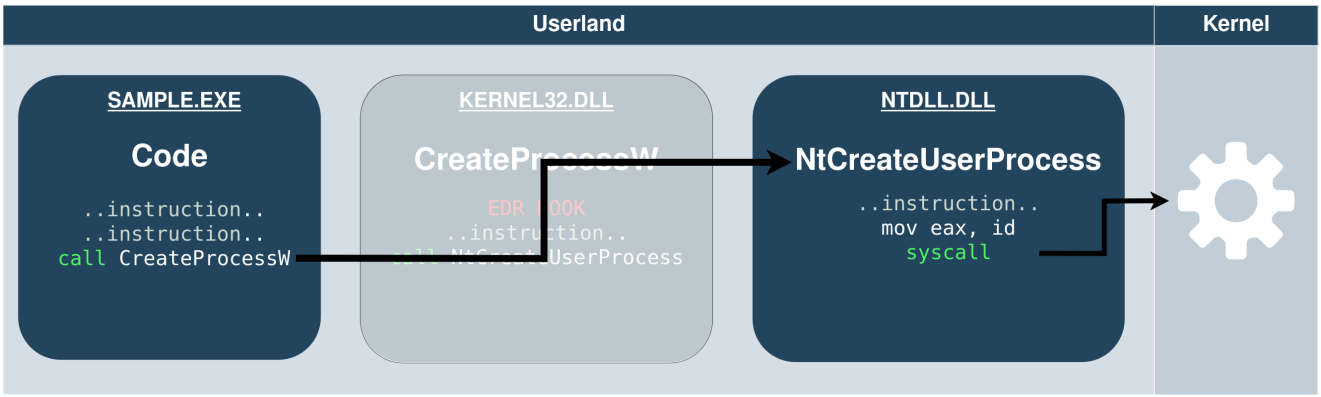

Figure 6: Sample calls native Windows functions directly, sidestepping the hooked Windows API function.

Modern EDR's usually do not just rely on hooks created for non-native functions anymore and instead hook some of the native functions directly.

Calling System Calls Directly

As calling native functions was widely exploited by malware and EDR's started placing hooks into these native functions, a new method to evade these hooks was developed which involves directly executing system calls (syscalls), basically re-implementing the code from the native function in the application itself (see Figure 7):

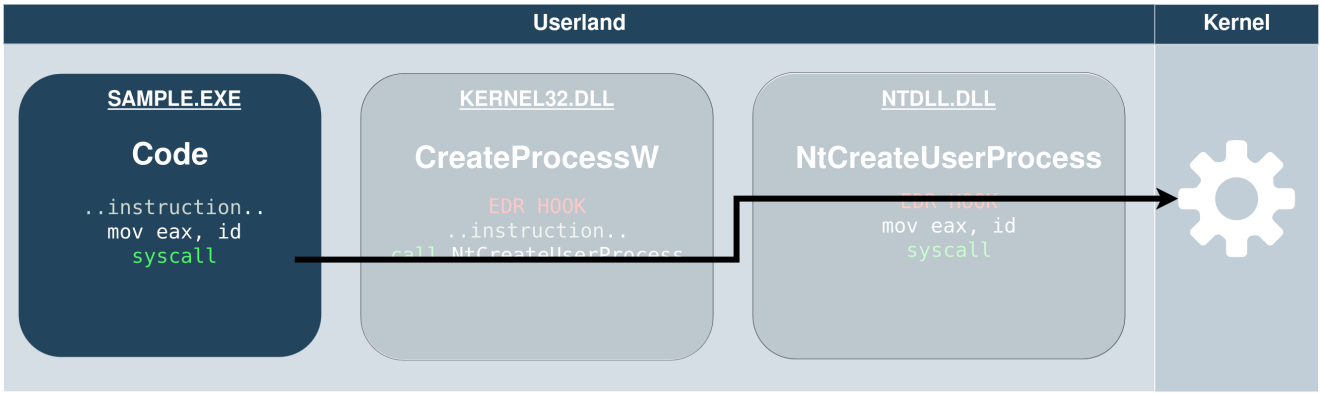

Figure 7: Sample executes syscall instruction directly to create a process, avoiding both the hook in the kernel32.dll library as well as the one in the native function in ntdll.dll.

Syscalls are the fundamental interface between an application and the operating system kernel, and by invoking them directly, malware can perform system-level operations without going through the API functions that are typically monitored by EDR systems.

This method is exemplified by the XLoader malware family, as demonstrated in this analysis report (see Figure 8):

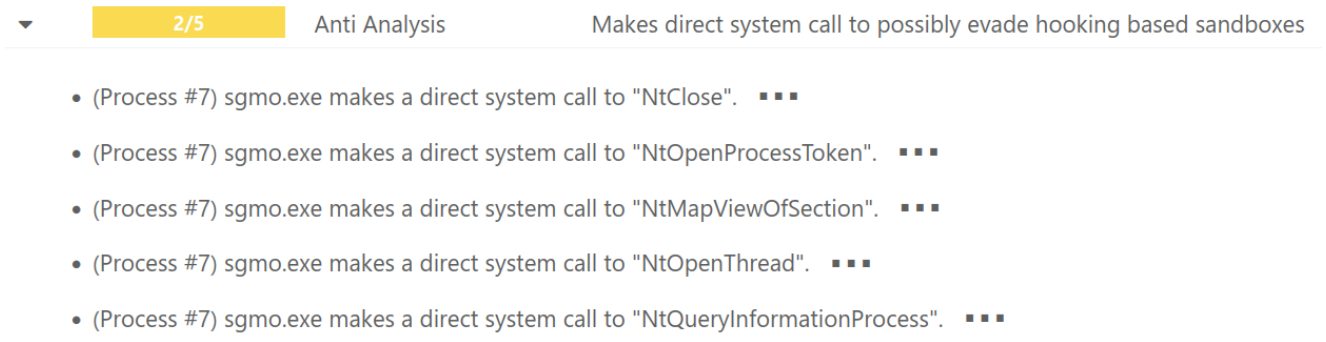

Figure 8: Direct system calls executed by an XLoader sample and detected by VMRay.

More recently, a trend towards even more sophisticated evasion techniques has been observed in the wild: **indirect syscalls**.

Indirect Syscalls

This approach involves the malware making system calls in such a way that they appear to originate from legitimate system components, like ntdll.dll, rather than from the malware itself. By doing so, the activity blends in with legitimate system operations, making it far more challenging for EDR systems to distinguish between benign and malicious behavior.

Indirect syscalls represent a significant evolution in malware evasion techniques. They not only complicate the process of detection but also underscore the necessity for continuous innovation in cybersecurity defense mechanisms. As malware developers refine their strategies to exploit the intricacies of operating systems and detection tools, the cybersecurity community must respond with equally sophisticated solutions to protect users and infrastructure from these evolving threats. This ongoing battle highlights the importance of advanced monitoring techniques, such as those based on recording transitions, which offer a more resilient defense against the clever evasion tactics employed by malware like Pikabot.

Diving deeper into the evasion techniques employed by the Pikabot malware family, we reach a critical aspect of its strategy: the use of indirect syscalls. This method represents a sophisticated approach to evade detection mechanisms that are designed to monitor and analyze system calls. Understanding this technique sheds light on the lengths to which malware authors will go to hide their malicious activities.

Detection of Direct Syscalls

Typically, direct syscalls are a straightforward method for executing system-level operations. However, cybersecurity solutions have adapted to this by implementing mechanisms to hook and monitor these calls directly. By analyzing the call stack, it becomes apparent when a syscall is being made directly by a suspicious sample rather than through legitimate system libraries like ntdll.dll. This direct approach, while effective in performing its intended operation, leaves a clear trail that security tools can follow to identify malicious activities.

The Shift to Indirect Syscalls

In response to the detection capabilities of modern cybersecurity tools, malware developers have evolved their techniques. Indirect syscalls emerge as a cunning solution to this challenge. By executing a jump to a syscall instruction within ntdll.dll, malware can make it appear as though the system call is originating from this legitimate library (see Figure 9). This method effectively masks the true origin of the call, blending the malicious operation with normal, expected system behavior.

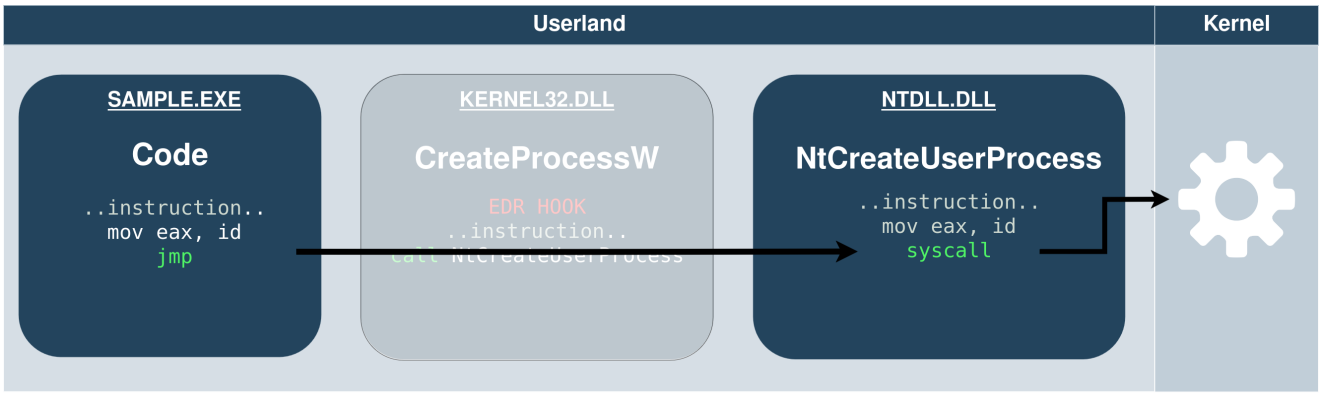

Figure 9: Sample jumps to syscall instruction in ntdll.dll, thus executing an indirect syscall.

This approach significantly complicates the task for detection tools. Since ntdll.dll is a critical component of the Windows operating system, used extensively by legitimate applications, distinguishing between benign and malicious use of its syscalls becomes a complex, nuanced task. The implication is that malware using indirect syscalls can operate under the radar of many detection mechanisms that rely on distinguishing between normal and abnormal syscall patterns.

A practical illustration of this technique in action is provided in our analysis, as highlighted in our VMRay report (see Figure 10):

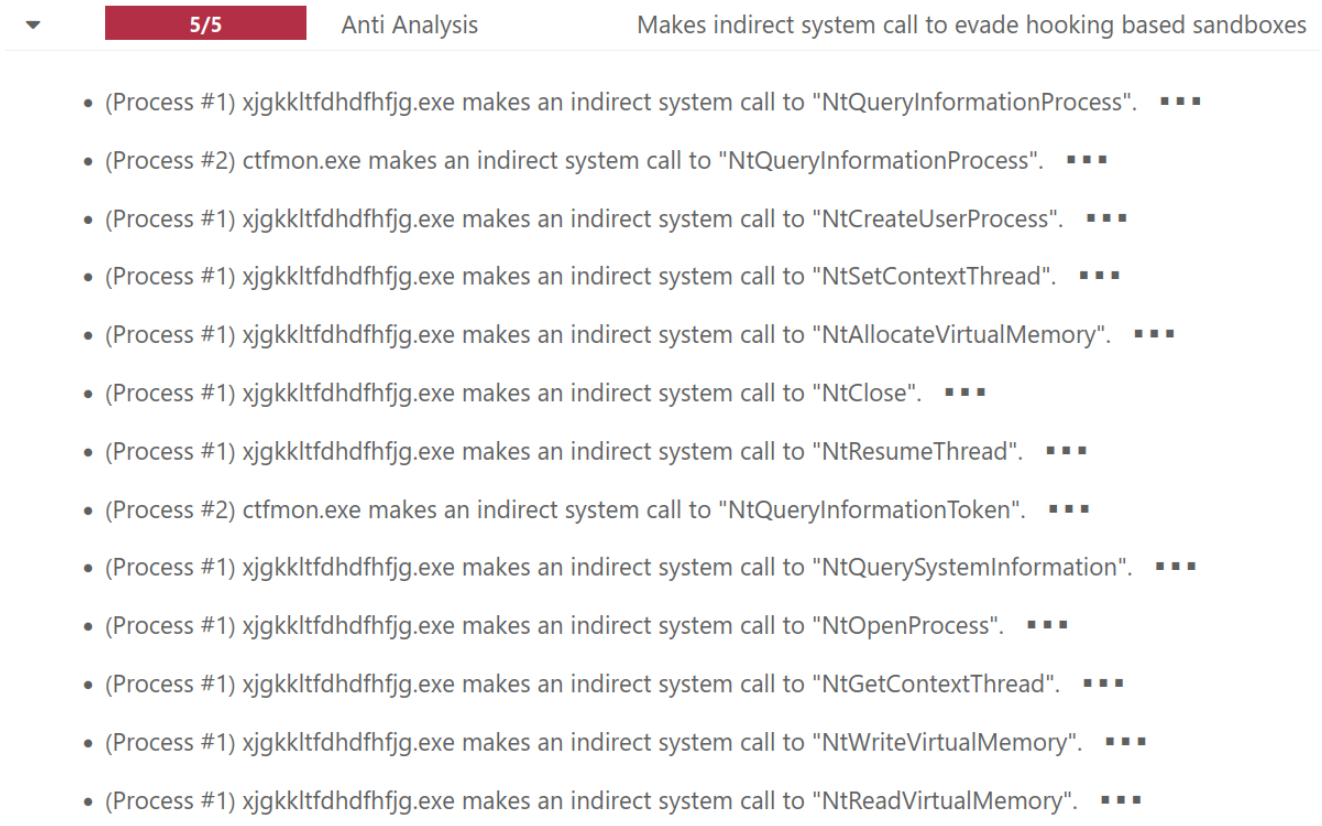

Figure 10: Indirect syscalls executed by Pikabot detected by VMRay Platform.

This evidence confirms that Pikabot employs indirect syscalls as part of its evasion strategy, showcasing the malware's sophisticated design aimed at circumventing traditional detection methods.

Implementation

The intricate evasion mechanisms of the Pikabot malware become even more apparent when examining the specifics of its implementation of indirect syscalls. This method showcases a high level of sophistication in avoiding detection. The process involves several steps, each designed to further obfuscate the malware's activities and complicate the task of cybersecurity defenses.

API Hashing and Selection of Zw* Functions

Initially, Pikabot undertakes the collection of Zw* native functions from ntdll.dll. Instead of referencing these functions directly, it employs API hashing. This technique involves calculating a unique hash for each API function name, which the malware then uses to identify and call the desired functions (see Figure 11). This method of indirect reference makes static analysis and detection significantly more challenging, as the actual function names do not appear in the malware's code.

```
int __cdecl mw_hash_api(_BYTE *ptr_buffer_api_name)
{
  int result; // eax
  int v3; // ecx

  result = 0x21229B0D;
  while ( *ptr_buffer_api_name )
  {
    v3 = *(unsigned __int16 *)ptr_buffer_api_name++;
    result ^= __ROR4__(result, 8) + v3;
  }
  return result;
}
```

Figure 11: API hashing code found in Pikabot sample.

Random Stub Selection for Evasion

To further enhance its evasion capabilities, Pikabot doesn't statically rely on a single native function for its operations. Instead, it randomly selects from the pool of collected Zw* functions (see Figure 12). This variability adds another layer of complexity for analysis tools, as it may appear as if the behavior is changing between executions, making the malware's footprint harder to identify.

```
if ( a1 )
{
  instruction_byte = 0;
  if ( a1 == 1 )
    instruction_byte = 0xBA;
  offset = 5 * (a1 == 1);
}
else
{
  instruction_byte = 0xE8;
  offset = 5;
}
Seed = mw_get_current_time();
srand(Seed);
do
  result = (unsigned __int8 *)(func_addresses_array[2 * (rand() % (unsigned int)(list_len[0] + 1))]
                             + offset
                             + array_base);
while ( *result != instruction_byte );
```

Figure 12: Pikabot randomly selects a function whose syscall instruction will be used.

Indirect Syscall Execution

Once a suitable Zw* function is selected, Pikabot meticulously prepares the correct syscall ID required for the intended system operation. It then jumps into the middle of the native function, skipping the syscall ID preparation code as this was already done by the malware, and finally lets ntdll.dll execute the syscall instruction (see Figure 13 for an overview). This indirect execution of the syscall, bypassing the higher-level API layers, minimizes the malware's trace within the system and evades detection mechanisms designed to monitor API calls. By doing so, Pikabot effectively operates beneath the radar, carrying out its malicious activities while blending in with legitimate system processes.

```
int __cdecl mw_call_indirect_syscall_wow32reserved(int arg_api_hash_value)
{
  int v2; // [esp-8h] [ebp-8h]
  int api_hash_value; // [esp-4h] [ebp-4h]
  int retaddr; // [esp+0h] [ebp+0h]

  dword_4050AC = v2;
  ret_address = retaddr;
  var_api_hash_value = (int)&arg_api_hash_value;
  g_syscall_stub_address = mw_get_syscall_id(api_hash_value);
  g_syscall_stub_offs_addr = (int)mw_get_syscall_stub_offs_addr(NtCurrentTeb()->WOW32Reserved != 0);
  ((void (*)(void))g_syscall_stub_offs_addr)();
  return ((int (*)(void))ret_address)();
}
```

Figure 13: Main execution of Pikabot regarding syscalls: getting the syscall ID for the API function it wants to call and jumping into the middle of a function that uses a syscall.

Transition-based monitoring

Whenever the malware sample and the EDR run on the same system within reach of each other, there is room for evasion. That is why in contrast to hooking-based EDR systems or sandboxing solutions, our method is therefore based on recording transitions, a technique detailed in our whitepaper here. Unlike hooking, which involves inserting small pieces of code (hooks) into the process of the malware itself, modular transitions are monitored from a more detached, outside perspective (see Figure 14). When the code executes a call that moves from one memory region to another—effectively transitioning across boundaries—this activity triggers a recording mechanism. This process occurs outside the observable range of the malware's processes, even outside the virtual machine, rendering it invisible and, consequently, more difficult for the malware to detect and evade.

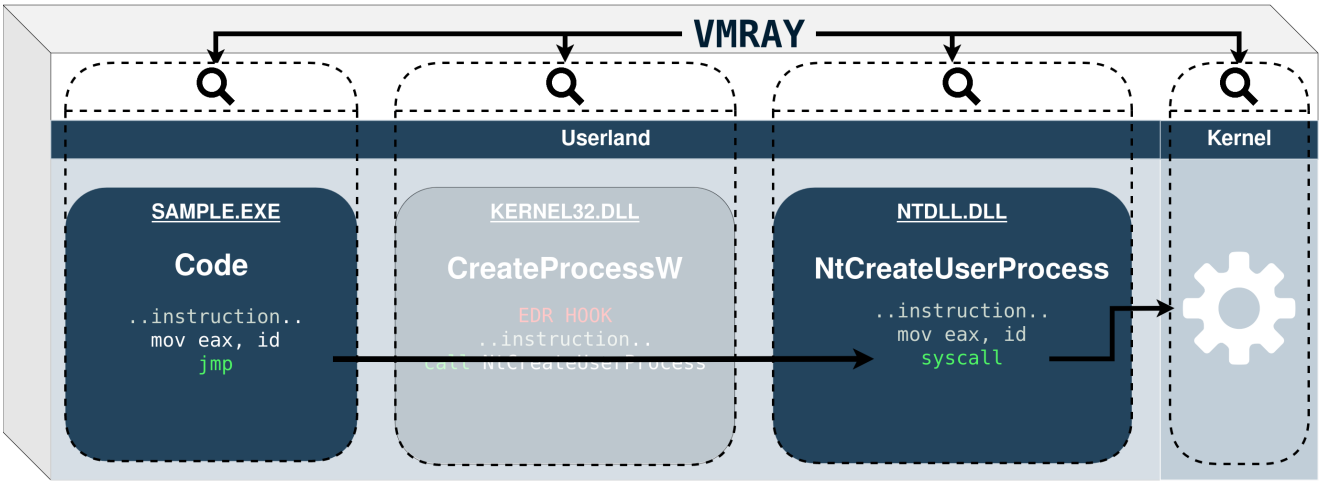

Figure 14: VMRay Platform watches the behavior from the outside without using hooks, thus avoiding detection and also allowing us to detect indirect syscalls as malicious behavior.

Conclusion

While not all EDRs are affected by the techniques outlined here, and some have the ability to detect indirect syscalls, our research shows that there already are works exploring alternatives to indirect syscalls, which means that hooking-based solutions have to consistently be adapted to new techniques. Note that hooking-based sandboxing solutions–those that use similar techniques to the ones outlined above to collect behavioral data about samples–are also plagued by the same issues.

This analysis underscores that no matter the complexity or quantity of evasion tactics employed, they stand little chance against our sophisticated, agentless, hook-free, transition and behavior-based analysis engine. While Pikabot is using sophisticated techniques to evade detection, we leverage this to our advantage by focusing on detecting these evasion attempts, thereby unmasking the true intent of the malware.

To provide more insight into Pikabot samples, we've developed a config extractor now available on our Platform. As adversaries continually seek innovative ways to distribute their malware, the consistent updates to the VMRay Platform ensure we remain ahead, always prepared to counteract the latest trends in malware evolution.

Hash	b025e37611168c0abcc446125a8bd7cb831625338434929febadfcc9cc4c816e
C2 IPs	103.82.243.5:13785
	86.38.225.105:13721
	37.60.242.85:9785
	89.117.23.185:2221

104.129.55.106:13783
86.38.225.106:2221
178.18.246.136:2078
154.12.233.66:2224
85.239.243.155:5000
145.239.135.24:5243
23.226.138.161:5242
104.129.55.105:2223
23.226.138.143:2083
57.128.165.176:13721
89.117.23.186:5632

Emre Güler
IT-Security Researcher | Malware Analysis @VMRay