Abstract

A reverse engineer trying to understand a protected binary is faced with avoiding detection by anti-debugging protections. Advanced protection systems may even load specialized drivers that can re-flash firmware and change the privileges of running applications, significantly increasing the penalty of detection. Hades is a Windows kernel driver designed to aid reverse engineering endeavors. It avoids detection by employing intelligent instrumentation via instruction rerouting in both user and kernel space. This technique allows a reverse engineer to easily debug and profile binaries without fear of invoking protection penalties.

Introduction

Hades is a tool for dynamic application analysis. It has function hooking capabilities similar to those of Microsoft Detours and WinAPIOverride (WAO), and it can also function as a debugger. It was developed to allow analysis of various malware binaries that were able to detect Detours and WAO. Both of these tools operate by injecting a DLL into a target binary; the DLL places hooks in specific sets of functions and logs information when those functions are called. The malware I was examining could detect that unauthorized DLLs were being loaded into the current process space. To avoid detection, I created an instrumentation tool based on instruction rerouting (to avoid most debugger detection techniques) that runs from the kernel rather than using DLL injection (which avoids DLL detection). It was then relatively straightforward to extend the tool's functionality to allow debugging of hooked executables.

Basic Operation

First, a target executable is identified for instrumentation. The Hades driver registers a callback function using *PsSetLoadImageNotifyRoutine* to detect when the target executable is loaded. When the target is loaded, but before it begins executing, Hades sets up a system call hook that will allow control to pass from the target to the Hades driver. First, a system call is hooked (any will do). Then a trampoline to a shared area of memory (described in section *Transition from user to kernel space*) is created. Finally, an instruction rerouting hook (a JMP to the trampoline code) is installed in the process at a user-specified virtual address and target execution is resumed.

Once the instruction pointer hits our rerouting hook, control is passed to the trampoline, which invokes an interrupt that will send execution to our hooked system call, where the Hades driver takes control. The driver will save the context (registers, stack, etc.) and display it, change any registers specified by the user, execute the original function bytes, and return control to the process at a point just after the rerouted instruction (virtual address + <JMP size>).

Transition from user to kernel space

Transitioning from user space to kernel space is achieved by trampolining through the system call dispatcher, which has memory accessible to both kernel and user code. The trampoline is installed in the *SharedUserData* memory area, which Windows uses as an efficient way to provide processes with certain frequently requested information. Hades

uses this area as a scratch space and to host its code for transitioning to the kernel from user space. The trampoline code is installed at offset 0x800 within the SharedUserData area (at address 0x7FFE000 from user space) to place it past Windows function pointers (which are the intended use of this area).

The trampoline is made up of two parts: a hook-specific set of instructions that save the processor state and identify the hooked function, and a generic handler that calls the Hades' hooked system call. Breaking up the trampoline in this way allows us to have multiple function rerouting routines installed in the targeted binary.

```
    pushad              // registers
    pushfd              // flags
    push 0x4098B0   // ID
    jmp dword ptr MyHandler
MyHandler:
    mov eax, 0x61   // ZwLoadDriver ID
    mov edx, esp
    _emit 0x0F      // SYSENTER
    _emit 0x34
```

Figure 1: Trampoline

As Fig. 1 shows, the first part of the trampoline code saves the processor context (pushad, pushfd), saves an identifier (where we came from), and then jumps to the generic code in "MyHandler." The code at MyHandler first moves 0x61 into EAX to identify the system call that should be invoked (it will be the one hooked by Hades). The stack pointer is saved into EDX, giving Hades a pointer to the user process stack. When SYSENTER executes, control is passed to the Windows Kernel and the hooked system call will be executed. At that point, Hades has full control and can be used for debugging, profiling, etc.

Profiler

As a profiler, Hades provides unique capabilities not found in other tools. Unlike most profilers, it can hook internal application functions without requiring source code or debugging symbols. It does not inject a DLL into the target process space like Detours or WAO, making it harder to detect.

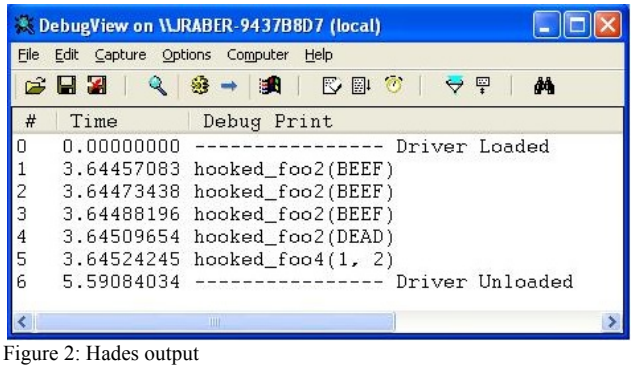

Figure 2: Hades output

Debugger

A slight modification can be made to the Hades driver to turn it into a debugger. Currently it is limited to just one breakpoint. One operational difference between debugging and profiling is that in debugging mode we restore the application bytes that had been used for Hades's JMP hook after the breakpoint is handled. In essence, Hades only supports a one-time breakpoint. It would not be difficult to add functionality to make the breakpoint persistent.

Effectiveness

Hades has several advantages over existing debuggers. Traditional Ring-3 debuggers such as OllyDbg and IDA Pro need to register with the OS to begin debugging a user application; this is easily detectable. Kernel-level debuggers avoid registration-based debugger checks but still use other standard debugging features such as INT 3's and hardware debug registers. They may also require the system to boot in debug-mode or require a second PC to control the system being debugged. Hades does not use standard OS debugging features, and it does not require any special system preparation.

Conclusion

We have found that reverse engineering advanced malware and software protections is often painfully inefficient with commonly available analysis tools. Hades uses non-standard debugging techniques such as instruction rerouting to avoid most detection techniques. Its ability to smoothly transition between user and kernel mode allows it to give the reverse engineer a comprehensive view of both the user- and kernel-mode code that is executed while a target process executes. We have made the source code to Hades available in the hopes that others may find it useful in their own projects.