Project Zero

News and updates from the Project Zero team at Google

Tuesday, April 16, 2019

Windows Exploitation Tricks: Abusing the User-Mode Debugger

Posted by James Forshaw, Google Project Zero

I've recently been adding native user-mode debugger support to NtObjectManager. Whenever I add new functionality I have to do some research and reverse engineering to better understand how it works. In this case I wondered what access you need to debug an existing running process and through that noticed an interesting security mismatch between what the user-mode APIs expose and what the kernel actually does which I thought would be interesting to document. What I'll describe is not a security vulnerability, but it is useful for exploiting certain classes of bugs, which is why it fits in my series on exploitation tricks.

I'm not going to go into any great depth about how the user-mode debugger works under the hood -- if you want to know more Alex Ionescu wrote 3 whitepapers (1, 2, 3) over 12 years ago about the internals on Windows XP, and the internals haven't really changed much since. Given that observation, while I'm documenting the behavior on Windows 10 1809 I'm confident these techniques work on earlier releases Windows.

Attaching to a Running Process

Debugging on modern versions of Windows NT is based around the *Debug* kernel object, which you can create by calling the *NtCreateDebugObject* system call. This object acts as a bridge to assign processes for

Search This Blog

Search

Pages

- Working at Project Zero

Archives

2019

- Windows Exploitation Tricks: Abusing the User-Mode... (Apr)
- Virtually Unlimited Memory: Escaping the Chrome Sa... (Apr)
- Splitting atoms in XNU (Apr)
- Windows Kernel Logic Bug Class: Access Mode Mismat... (Mar)
- Android Messaging: A Few Bugs Short of a Chain (Mar)
- The Curious Case of Convexity Confusion (Feb)
- Examining Pointer Authentication on the iPhone XS (Feb)
- voucher_swap: Exploiting MIG reference counting in... (Jan)

debugging and wait for debug events to be returned. The creation of this object is hidden from you by the Win32 APIs and each thread has its own debug object stored in its TEB.

If you want to attach to an existing process you can call the Win32 API DebugActiveProcess which has the prototype:

```
BOOL DebugActiveProcess(_In_ DWORD dwProcessId);
```

This ultimately calls the native API, *NtDebugActiveProcess,* which has the following prototype:

```
NTSTATUS NtDebugActiveProcess(
    _In_ HANDLE ProcessHandle,
    _In_ HANDLE DebugObjectHandle);
```

The *DebugObjectHandle* comes from the TEB, but requiring a process handle rather than a PID results in a mismatch between the call semantics of the two APIs. This means the Win32 API is responsible for opening a handle to a process, then passing that to the native API.

The question which immediately come to mind when I see code like this is what access does the Win32 API require and then what does the kernel API actually enforce? This isn't just idle musing, a common place for security vulnerabilities to manifest is in mismatched assumptions between interface layers. A good example is the discovery that NTFS hardlinks required a write access when being created from a Win32 application using CreateHardlink due to the user-mode API opening the target file with *WRITE_ATTRIBUTES* access. However, the kernel APIs didn't require any access (see my blog post about the hard link issue here) allows you to hardlink to any file you can open for any access. To find out what the Win32 API and Kernel APIs enforce we need to look at the code in a disassembler (or look at Alex's original write ups which has the code RE'd as well). The code in *DebugActiveProcess* calls a helper, *ProcessIdToHandle* to get the process handle which looks like the following:

```
HANDLE ProcessIdToHandle(DWORD dwProcessId) {
 NTSTATUS Status;
 OBJECT_ATTRIBUTES ObjectAttributes;
 CLIENT_ID ClientId;
 HANDLE ProcessHandle;
 DWORD DesiredAccess = PROCESS_CREATE_THREAD |
                       PROCESS_VM_OPERATION |
                       PROCESS_VM_READ |
```


- Taking a page from the kernel's book: A TLB issue ... (Jan)

2018

- On VBScript (Dec)
- Searching statically-linked vulnerable library fun... (Dec)
- Adventures in Video Conferencing Part 5: Where Do ... (Dec)
- Adventures in Video Conferencing Part 4: What Didn... (Dec)
- Adventures in Video Conferencing Part 3: The Even ... (Dec)
- Adventures in Video Conferencing Part 2: Fun with ... (Dec)
- Adventures in Video Conferencing Part 1: The Wild ... (Dec)
- Injecting Code into Windows Protected Processes us... (Nov)
- Heap Feng Shader: Exploiting SwiftShader in Chrome... (Oct)
- Deja-XNU (Oct)
- Injecting Code into Windows Protected Processes us... (Oct)
- 365 Days Later: Finding and Exploiting Safari Bugs... (Oct)
- A cache invalidation bug in Linux memory managemen... (Sep)
- OATmeal on the Universal Cereal Bus: Exploiting An... (Sep)
- The Problems and Promise of WebAssembly (Aug)
- Windows Exploitation Tricks: Exploiting Arbitrary ... (Aug)
- Adventures in vulnerability reporting (Aug)
- Drawing Outside the Box: Precision Issues in Graph... (Jul)

```
                        PROCESS_VM_WRITE |
                        PROCESS_QUERY_INFORMATION |
                        PROCESS_SUSPEND_RESUME;

 ClientId.UniqueProcess = dwProcessId;
 InitializeObjectAttributes(&ObjectAttributes, NULL, ...)

 Status = NtOpenProcess(&ProcessHandle,
                         DesiredAccess,
                        &ObjectAttributes,
                        &ClientId);
 if (NT_SUCCESS(Status))
   return ProcessHandle;
 BaseSetLastNTError(Status);
 return NULL;
}
```

Nothing shocking about this code, a process is opened by its PID using *NtOpenProcess*. The code requires all the access rights that a debugger would expect:

- Creating new threads, which is how the initial break point is injected.
- Access to read and modify memory to write break points and inspect the running state of the process.
- Suspending and resuming the entire process.

Once the kernel gets the process handle it needs to convert it back to a process object pointer using ObReferenceObjectByHandle. The API takes an desired access mask which is checked against the open handle access mask and only returns the pointer if the check succeeds. Here's the relevant snippet of code:

```
NTSTATUS NtDebugActiveProcess(HANDLE ProcessHandle,
                              HANDLE DebugObjectHandle) {
 PEPROCESS Process;
 NTSTATUS status = ObReferenceObjectByHandle(
            ProcessHandle,
            PROCESS_SUSPEND_RESUME,
            PsProcessType,
```

```
            KeGetCurrentThread()->PreviousMode,
            &Process);
 // ...
}
```

Here's a pretty big mismatch in security expectations. The user mode API opens the process with much higher access than the kernel API requires. The kernel only enforces access to suspend and resume the target process. This makes sense from a kernel perspective (as Raymond Chen might say, looking at it through Kernel Colored Glasses) as the immediate effect of attaching a process to a debug object is to suspend the process. You'd assume that without having other access such as VM Read/Write there's not much debugging going on but from a kernel perspective that's irrelevant. All you need to use the kernel APIs is the ability to suspend/resume the process (through *NtDebugContinue*) and read events from the debug objects. The fact that you might get memory addresses in the debug events which you can't access isn't that important from a design perspective.

Where's the problem you might ask? With only *PROCESS_SUSPEND_RESUME* access you can "debug" a process with limited access, but without the rest of the access rights you'd not be able to do that much. What can we do if we have only PROCESS_SUSPEND_RESUME access?

Accessing Debug Events

The answer to the question is based on the event you receive when you first wait for a debug event, CREATE_PROCESS_DEBUG_INFO. Note the native structure is slightly different but it's close enough for our purposes.

The create process event is received whenever you connect to an active process, it allows the debugger to synchronize its state by providing a set of events such as process creation, thread creation and loaded modules that you'd have received if you were directly starting a process under a debugger. In fact the *NtDebugActiveProcess* calls the method *DbgkpPostFakeProcessCreateMessages* which gives away the status of the debug events.

What's interesting about *CREATE_PROCESS_DEBUG_INFO* is you'll notice there are HANDLE parameters, *hFile*, *hProcess* and *hThread* corresponding to a file handle to the process executable, a handle to the process being debugged and finally a handle to the initial thread. Checking in *DbgkpPostFakeProcessCreateMessages* the objects are captured, however the handles are not generated.

Instead the handles are created inside the *NtWaitForDebugEvent* system call, specifically in *DbgkpOpenHandles*. See if you can spot the problem with the following code snippet from that function:

```
NTSTATUS DbgkpOpenHandles(PDEBUG_EVENT Event,
                          EPROCESS DebugeeProcess,
                          PETHREAD DebugeeThread) {

 // Handle other event types first...
 if (Event->DebugEventCode == CREATE_PROCESS_DEBUG_EVENT) {
   if (ObOpenObjectByPointer(DebugeeThread, 0, NULL,
         THREAD_ALL_ACCESS, PsThreadType,
         KernelMode, &Event->CreateProcess.hThread) < 0) {
     Event->CreateProcess.hThread = NULL;
   }

   if (ObOpenObjectByPointer(DebugeeProcess, 0, 0,
         PROCESS_ALL_ACCESS, PsProcessType,
         KernelMode, &Event->CreateProcess.hProcess < 0) {
     Event->CreateProcess.hThread = NULL;
   }

   ObDuplicateObject(PsGetCurrentProcess(),
                     Event->CreateProcess.hFile,
                     PsGetCurrentProcess(),
                    &Event->CreateProcess.hFile, 0, 0,
                     DUPLICATE_SAME_ACCESS, KernelMode);
 }
 // ...
}
```

Did you spot the problem? This code is using the ObOpenObjectByPointer API to convert the debugee process and thread objects back to handles, that in itself is fine. However the problem is the API is called with the access mode set to *KernelMode* which means the call does not perform any access checking. That's not great but again wouldn't be an issue if it wasn't also requesting additional access above *PROCESS_SUSPEND_RESUME*. This code is effectively giving all access rights to the caller of *NtWaitForDebugEvent* to the debugged target process.

- FireEye Exploitation: Project Zero's Vulnerability... (Dec)
- Between a Rock and a Hard Link (Dec)
- Windows Sandbox Attack Surface Analysis (Nov)
- Hack The Galaxy: Hunting Bugs in the Samsung Galax... (Nov)
- Windows Drivers are True'ly Tricky (Oct)
- Revisiting Apple IPC: (1) Distributed Objects (Sep)
- Kaspersky: Mo Unpackers, Mo Problems. (Sep)
- Stagefrightened? (Sep)
- Enabling QR codes in Internet Explorer, or a story... (Sep)
- Windows 10^H^H Symbolic Link Mitigations (Aug)
- One font vulnerability to rule them all #4: Window... (Aug)
- Three bypasses and a fix for one of Flash's Vector... (Aug)
- Attacking ECMAScript Engines with Redefinition (Aug)
- One font vulnerability to rule them all #3: Window... (Aug)
- One font vulnerability to rule them all #2: Adobe ... (Aug)
- One font vulnerability to rule them all #1: Introd... (Jul)
- One Perfect Bug: Exploiting Type Confusion in Flas... (Jul)
- Significant Flash exploit mitigations are live in ... (Jul)
- From inter to intra: gaining reliability (Jul)
- When 'int' is the new 'short' (Jul)
- What is a "good" memory corruption vulnerability? (Jun)

The result of this behavior is given a process handle with *PROCESS_SUSPEND_RESUME* it's possible to use that to get full access to the process and initial thread even if the objects wouldn't grant the caller that access. It might be argued that "You've attached the debugger to the process, what did you expect?". Well I would expect the caller would need to have opened suitable process and thread handles before attaching the debugger and use them to access the target, or if the kernel has to create new handles at least do an access check on them. This leads us to our first exploitation trick:

Exploitation trick: *given a process handle with PROCESS_SUSPEND_RESUME access you can convert it to a full access process handle through the debugger APIs.*

It's probably rare you'll encounter this type of bug as *PROCESS_SUSPEND_RESUME* is considered a write access to a process. Anything which would leak this access to a privileged process would also have leaked the other write accesses such as *PROCESS_CREATE_THREAD* or *PROCESS_VM_WRITE* and it'd be game over. To prove this exploit trick works the following is a simple PowerShell script which takes a process ID, opens the process for *PROCESS_SUSPEND_RESUME*, attaches it to a debugger, steals the handle and returns the full access handle.

```
param(
    [Parameter(Mandatory)]
    [int]$ProcessId
)

# Get a privileged process handle with only PROCESS_SUSPEND_RESUME.
Import-Module NtObjectManager

Use-NtObject($dbg = New-NtDebug -ProcessId $ProcessId) {
    Use-NtObject($e = Start-NtDebugWait $dbg -Infinite) {
        $dbg.Detach($ProcessId)
        [NtApiDotNet.NtProcess]::DuplicateFrom($e.Process, -1)
    }
}
```

What about the third handle in *CREATE_PROCESS_DEBUG_INFO*, the handle to the process executable file? This has a different behavior, instead of opening a raw pointer it duplicates an existing handle. If you look at the code it seems to be duplicating from the current caller's process and back again, why would it

need to do the duplication if it was already in the debugger process? The key is the final parameter, again it's passing *KernelMode*, which means *ObDuplicateObject* will actually duplicate a kernel handle to the current process. The file handle is opened when attaching to the process and uses the following code:

```
HANDLE DbgkpSectionToFileHandle(PSECTION Section) {
 HANDLE FileHandle;
 UNICODE_STRING Name;
 OBJECT_ATTRIBUTES ObjectAttributes;
 IO_STATUS_BLOCK IoStatusBlock;

 MmGetFileNameForSection(Section, &Name);

 InitializeObjectAttributes(&ObjectAttributes,
     &Name,
     OBJ_CASE_INSENSITIVE |
     OBJ_FORCE_ACCESS_CHECK |
     OBJ_KERNEL_HANDLE);

 ZwOpenFile(&FileHandle, GENERIC_READ | SYNCHRONIZE,
            &ObjectAttributes, &IoStatusBlock,
            FILE_SHARE_ALL, FILE_SYNCHRONOUS_IO_NONALERT);
 return FileHandle;
}
```

This code is careful to pass *OBJ_FORCE_ACCESS_CHECK* to the file open call to ensure it doesn't give the debugger access to arbitrary files. The file handle is stored away to be reclaimed by the later call to *NtWaitForDebugEvent*. This leads us to our second, and final exploitation trick:

Exploitation trick: *with an arbitrary kernel handle closing bug you can steal kernel handles.*

The rationale behind this exploitation trick is that once the handle is captured it's stored indefinitely, at least while the process still exists. The handle can be retrieved at any arbitrary point in time. This gives you a much bigger window to exploit a handle closing bug. An example of this bug class was one I found in a Novell driver, issue 274. In this case the driver wouldn't check whether *ZwOpenFile* succeeded when writing a log entry and so would reuse the handle value stored in the stack when it called *ZwClose*. This results in an arbitrary kernel handle being closed. To exploit the Novell bug using the debugger you'd do the following:

- Mac OS X and iPhone sandbox escapes (Jul)
- pwn4fun Spring 2014 - Safari - Part I (Jul)
- Announcing Project Zero (Jul)

1. Generate a log entry to create a kernel handle which is then closed. The value on the stack is not overwritten.
2. Debug a process to get the file handle allocated. Handle allocation is predictable so there's a good chance that the same handle value will be reused as the one used with the bug.
3. Trigger the handling closing bug, in this case it'll close the existing value on the stack which is now allocated by the debugger resulting in a dangling handle value.
4. Exercise code in the kernel to get the now unused handle value reallocated again. For example *SepSetTokenCachedHandles* called through *NtCreateLowBoxToken* will happily duplicate other kernel handles (although since I reported issue 483 there are fairly strict checks on what handles you can use.
5. Get the debugger to return you the handle.

Handle closing bugs do exist, though they're perhaps rare. Also you have to be careful, as typically closing an already closed kernel handle can result in a bug check.

Wrapping Up

The behavior of user-mode debugging is another case where there are unexpected consequences to the design of the functionality. Nothing I've described here is a security vulnerability, but the behavior is interesting and it's worth looking out for cases where it could be used.