# INSECTION

## AWESOMELY EXPLOITING SHARED MEMORY OBJECTS

INFILTRATE 2015

ALEX IONESCU

@AIONESCU

# WHO AM I?

Chief Architect at CrowdStrike, a security startup

Previously worked at Apple on iOS Core Platform Team

Co-author of *Windows Internals 5$^{th}$ and 6$^{th}$ Editions*

Reverse engineering NT since 2000 – main kernel developer of ReactOS

Instructor of worldwide Windows Internals classes

Conference speaking:

- Infiltrate 2015
- Blackhat 2015, 2013, 2008
- SyScan 2015-2012, NoSuchCon 2014-2013, Breakpoint 2012
- Recon 2014-2010, 2006

For more info, see [www.alex-ionescu.com](www.alex-ionescu.com)

# WHAT THIS TALK IS ABOUT

Some Windows Internals, of course ;-)

- Sections, Control Areas, Segments, Address Windowing Extensions (AWE)

Section Object Internals

- What they are
- Different types of shared objects
- How to abuse named section objects

Determining Section Ownership

- Object Type Creator Tracing
- Segment Tracking

Epic WinDBG Scripting!

Fuzzing for Section Vulnerabilities

- Three real-life case studies on my own laptop (LIVE!)

# WHAT THIS TALK IS ALSO ABOUT

KASLR Bypassing Using Windows' Self-Referencing PTE Flat Mapping

- Builds upon previous research, with some new twists
- [ref: https://labs.mwrinfosecurity.com/blog/2014/08/15/windows-8-kernel-memory-protections-bypass/ by Jeremy Fetiveau (@__x86) from MWR]
- [ref: http://hypervsir.blogspot.com/2014/11/page-structure-table-corruption-attacks.html by Kevin Bing (@imsky) from Intel]
- [ref: https://www.syscan.org/index.php/download/get/5ba020d2e4b4817cb1086a0fcc2cea54/SyScan15%20Peter%20Hlavaty%20-%20Back%20To%20The%20CORE.pdf by Peter Hlavaty (@zer0mem) from KeenTeam]

SMEP Bypassing Using AWE

- Two unpublished techniques!

Zero Day Kernel Code Signing Bypass

- This is an OFFENSIVE con, right?

# PART 1 (User) OUTLINE

Windows Internals Primer

- Windows Shared Memory Model
  - Private vs Shared
  - Four Types of Shared Memory
- Windows Object Management
  - Naming and Security
- Going Deep in Sections
  - Control Areas, Segments and Subsections
  - Key Debugger Data

Hunting For Weakly Secured Section Objects

- Determining Ownership
- Fuzzing Techniques

Demo Time [Three Case Studies]

# PART 2 (Kernel) OUTLINE

Tracking Kernel vs User Section Ownership

- Process Attachment Artifacts
- Kernel Object Header Flag

Difficulties of Exploiting Kernel Sections and Workarounds

- SMEP/KASLR
- Write-What-Where Conversion
- Self-Referencing PTE Space
- Address Windowing Extensions

Demo Time

QA & Wrap-Up

# Windows Memory Basics

# Private vs Shared Memory

Windows separates memory into **allocated** or **free** virtual address space

Allocated memory is separated into **committed** and **reserved**

Committed memory is further sub-divided into **private** and **shared**

- **Private**: new/malloc/HeapAlloc/VirtualAlloc/Stack/TEB/PEB/…
- **Shared**: Typically memory-mapped files (DLLs and EXEs)

Allocated memory is represented by a **Virtual Address Descriptor (VAD)**

Shared memory is also represented by a **Section Object**

There are multiple types and usages for shared memory…

# Four Types of Shared Memory

**Memory Mapped File Sections (UNNAMED, FILE-BACKED)**

- Typically associated with dynamic libraries and executables from disk
  - LoadLibrary(), etc…
  - Windows calls these **Image Sections**
- But can also be data files mapped from disk
  - Windows calls these **Data Sections**

- Size comes from the file
- Page protections come from caller (**Data**) or from PE Sections (**Image**)

- They don’t need a name – because they are bound by a file object on disk
  - Cache Manager and Memory Manager take care of finding the section object for an already-mapped file
- **But they CAN actually have a name – this is how KnownDLLs work**
  - Abused by myself at SyScan 2013

# Four Types of Shared Memory

**ALPC Sections** **(UNNAMED, PAGEFILE-BACKED)**

- Used and managed by Advanced Local-Procedure Call kernel IPC mechanism
  - Leveraged by RPC and DCOM, as well as many other internal services
- Size determined by ALPC attribute during ALPC Message Passing
- Protection is either **RO** or **RW**, managed by ALPC Subsystem
  - Protection changes are protected through **SEC_NO_CHANGE** and **MmSecureVirtualMemory**, **MmSecureVirtualMemoryAgainstWrites**
- They don't need a name because ALPC maps the server/client ends as needed during message passing
  - No way to get access to underlying object
- Abused by myself at SyScan 2014

# Four Types of Shared Memory

**Non-ALPC Sections (UNNAMED, PAGEFILE-BACKED)**

- Pretty much the same as the ALPC Sections, but not managed through ALPC
  - Object Manager returns **handle** to the Section Object to the caller
- Caller must find a way to pass the handle to the process it wants to share the data with
  - **Inheritance:** Caller spawns a new child process, and the handles are inherited
  - **Forking:** Caller uses **fork**() API (HI BEN!!!), and the memory address space is inherited
  - **Duplication:** Caller uses **DuplicateHandle**() API, and the handle is duplicated to another process
- Unnamed Section Objects **do not have security checks enabled** (no ACLs)
  - Not an issue for ALPC Sections, because user-mode code doesn't have the **handle**
  - But big problem for non-ALPC Sections
  - [ref: http://googleprojectzero.blogspot.com/2014/10/did-man-with-no-name-feel-insecure.html James Forshaw (@tiranid) from Google]

# Four Types of Shared Memory

**Named Sections** **(NAMED, PAGEFILE-BACKED)**

- Very similar to the previous type (identical at the memory manager level)
- Caller uses the Object Manager to assign a **name** to the section object
- Size and protection both specified by the creator
  - Interesting side-effect w.r.t memory consumption – creating but not mapping leaks memory!
- Any process on the system can now attempt to **open** the section by **name**
  - ACLs are critical to ensure only designated processes receive a handle to the section
  - ACLs can enforce **READ** vs **WRITE** permissions on the shared memory itself
  - Windows ACL Rule: No DACL (NULL) means **everyone has access**
- Nobody has yet systematically abused these yet!

# Which to Use?

If sharing with only one process, consider **COM/RPC** (ALPC) if the communication model makes sense

- Takes care of the **handle** passing, **RO** vs **RW** semantics, and **synchronization**

Otherwise, consider security boundary needs and use **Non-ALPC Sections**

- If child/other process is expected to have **limited** rights, **do not use!**

If separate rights are needed, consider a **named section**

- Make sure to **secure** it!
- Incorrectly secured object can lead to **arbitrary** processes having **RW** access to **privileged** processes

# “Objects”

Windows kernel tracks all of its shared, user-accessible resources, through a generalized Object Manager mechanism

- Also used for some kernel-internal resources (easy abstraction, so why not)
- Other kernel-mode components implement similar ideas too (Window Mgr.)

Handles **allocation** and **destruction** (with a simple reference-based GC)

Implements naming scheme (allows **hierarchical namespace** and lookup)

Leverages Security Reference Monitor (SRM) for both **discretionary access control** and **mandatory integrity control**

- Uses familiar ACL model just like for Files / Registry Keys

Exposes underlying kernel object through a per-process **handle** (index)

- Handle will **cache** the requested access rights at **creation time** and reuse them if granted

# Section Objects

The **Section Object** represents the runtime semantics of a piece of shared memory

- Virtual Address, Flags (**Image** vs **Data**), Size, Session ID, Initial Protection

The actual memory backing semantics are actually held by a **Control Area**

- **Multiple section** objects can point to the **same control area** (multiple mappings of the same file)
- **Multiple section** objects can exist for a **file** (mapped once as data, and again as image)
- **Multiple control areas** can exist for the **same section** object (ask me offline if you really care how this can happen)

The **Control Area** points to a **Segment** and is made up of **Subsections**

# Segment and Subsections

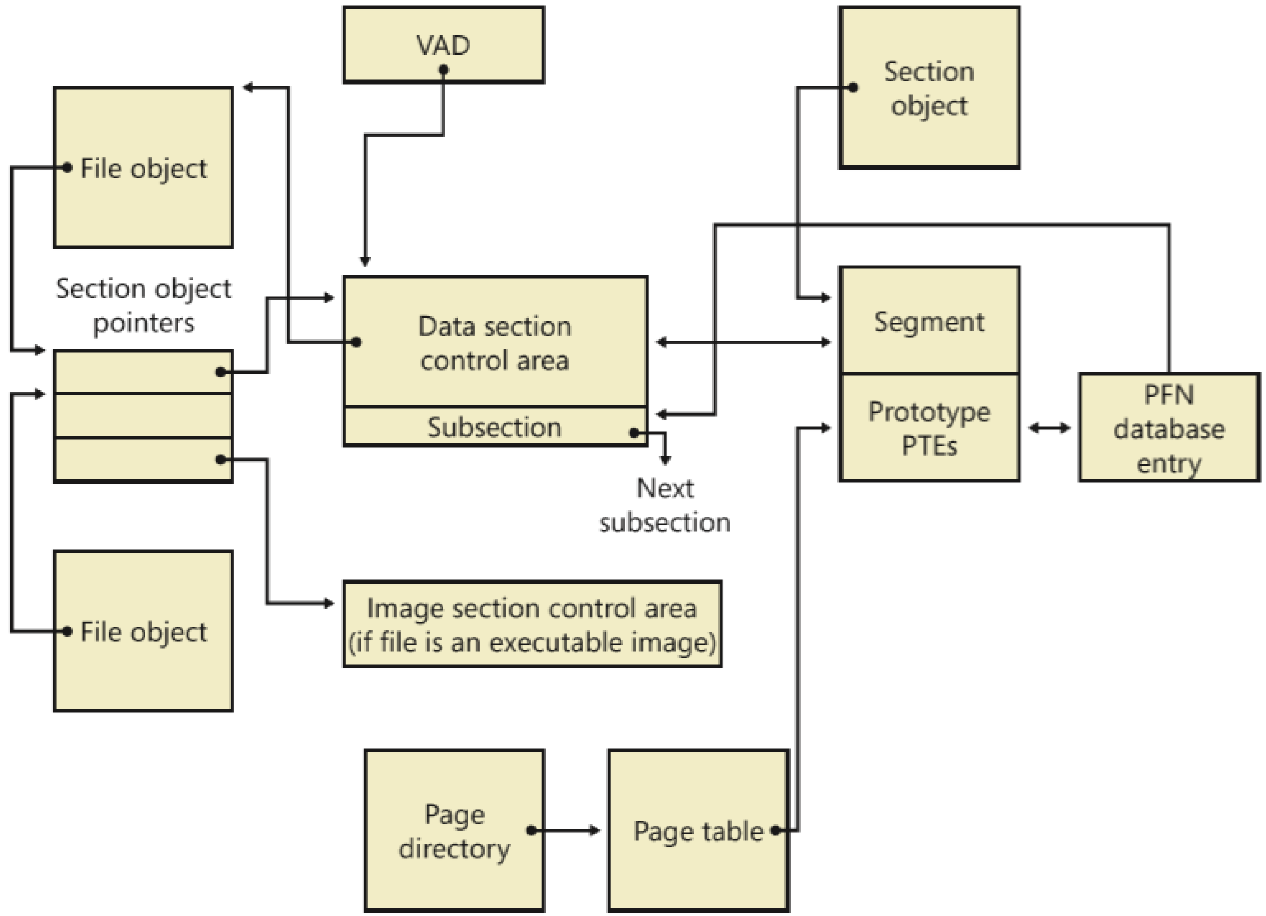
**Subsections** follow the **Control Area** and describe different pieces of the mapped file

- For **Image** file-backed sections, the subsections are created based on the PE sections
- For **Data** file-backed sections, the subsections are created if multiple sparse offsets of the file are mapped with different protections
- For pagefile-backed (i.e.: not file-backed) sections, there is only one **subsection**

The **Segment** describes additional internal memory-manager flags for the **Control Area**

- Points to the **Creator Process** that owns the pagefile-backed section
- Also stores address of the **first** virtual address **mapping** (if any)
- Stores the “**Prototype PTEs**” which define up the physical location of the data

# So… you got all that, right?



# Hunting For Insecure Sections

# Finding (Un)Secured Objects

**SysInternals** has two great tools for looking at the Object Manager **namespace**

WinObj is a GUI Tool. Double-clicking will display a **properties** dialog, with a GUI ACL **Editor**

- However, it doesn't for objects that Mark hasn't written custom code for – cannot show ACLs for arbitrary objects (works for Sections, however)
- **Hard** to **automate**

AccesChk is a CLI Tool. Can **dump** a particular object, or **recursively** dump an object directory.

- Understands all object types
- Has flags to filter by object type, and output is **easy** to send to a **parser**

# Quick Demo…

To dump all named section object ACLs, one can do

- **accesschk –s –o –t Section > acl.txt**
  - Parse with some tool to figure out what's unsecured

Can also just do quick checks:

- **accesschk –s –o –t Section | findstr Everyone**

```
\BaseNamedObjects\__ComCatalogCache__
  Type: Section
  R  Everyone
  RW NT AUTHORITY\SYSTEM
  RW BUILTIN\Administrators
  R  APPLICATION PACKAGE AUTHORITY\ALL APPLICATION PACKAGES
\BaseNamedObjects\GDA:  ESENT Performance Data Schema Version 265
  Type: Section
  RW NT AUTHORITY\SYSTEM
  RW BUILTIN\Administrators
  RW NT AUTHORITY\SERVICE
  RW BUILTIN\Performance Monitor Users
  RW BUILTIN\Performance Log Users
\BaseNamedObjects\Wmi Provider Sub System Counters
  Type: Section
  RW NT AUTHORITY\SYSTEM
  RW NT AUTHORITY\NETWORK SERVICE
  RW NT AUTHORITY\LOCAL SERVICE
  R  Everyone
```

# Issues with the Tools

Starting with Windows Vista, the kernel has **private namespaces**

Documented on MSDN, can be used by user-mode applications to **hide** their objects

- Won’t show up in any user-mode API
- Won’t show up in !object in the Debugger
  - With private symbols + an undocumented flag, you can see them though ;-)
- These aren’t really any more protected than normal objects, they’re just hidden

Also, the tools only show you the security descriptor, but not who **owns** the object, who **created** it, who is **accessing** it, etc..

- All this information is in the kernel debugger, however…

# Object Creator Type Tracking

Windows has a poorly documented internal flag which changes the Object Manager behavior in three ways:

- It stores the **creator process** (EPROCESS Object) for each object
- It **links all the objects** of the same type together (no need for memory scanning or fingerprint)
- It (optionally) takes a complete **stack backtrace** at object creation time

You can enable this really useful behavior with the **Global Flags Utility**

- Set the option "**maintain a list of objects for each type**"
- Can also use **!gflag +otl** in the kernel debugger (sets flag 0x4000)

This is exposed in two ways

- A new **NtQueryObject** information class is exposed, dumps all objects of a given type
- A new capability is added to the **!object** WinDBG extension to do the above too

# Dump all the things!

Once the Global Flag is set, you can use the following syntax in WinDBG:

- **!object 0 Section**

This will dump *all* section objects

- We only care about the ones that have names…
- No way to filter

Well… we can filter with the magic "**.foreach**" WinDBG scripting command

- Allows parsing output from the debugger, and choosing/skipping which **tokens** to use as **input**

But instead of using a WinDBG extension and then a filtering script…

- Let's just write our own script

# Linked Objects

All objects of the same type have an **OBJECT_TYPE** structure which owns the **type** of the object (Process vs Thread vs Section)

When the **+otl** flag is enabled, a **TypeList** doubly-linked circular list is used to link the objects of the same type together

- dt nt!_OBJECT_TYPE TypeList

Every unique object has a **main header** and a possible set of **extra headers** that define the attributes for the particular object

- Name, memory quota consumption, etc…
- When **+otl** is used, a new **creator info header** is allocated for each object
- dt nt!_OBJECT_HEADER_CREATOR_INFO

The object creator info header is always **on top of the main header**

- dt nt!_OBJECT_HEADER

# Enumerating a Linked List

The debugger has a **!list** extension to iterate over a linked list

Each iteration can execute a command typed between the **–x "…"** input

- The current entry being iterated is stored in a **@$extret** register variable

How do we get the linked list for the section object?

- Each object has an exported symbolic name
- Section: **nt!MmSectionObjectType**

This will iterate over every section object:

- `r? @$t0 = *(nt!_OBJECT_TYPE**)@@(nt!MmSectionObjectType)`
- `!list -x "?? @$extret + sizeof(nt!_OBJECT_HEADER_CREATOR_INFO) + sizeof(nt!_OBJECT_HEADER)" @@(&@$t0->TypeList)`

# Filtering Named Objects

If an object has a name, it has an **object name info header**

If an object has any type of **extra headers**, the **main header** describes that by setting the corresponding **bit** into the **InfoMask** bitfield

- Bit Value **0x2** is used for the name info header
- dt nt!_OBJECT_HEADER InfoMask

We can filter named objects by doing something like this:

```
$$ T1 is the Object Header
.if (@@((@$t1->InfoMask & 0x2) != 0))
{
     $$ This is a named object
}
```

# Finding the Security Descriptor

If an object is protected by a security descriptor, the **SecurityDescriptor** field in the **main header** will point to it (almost…)

- dt nt!_OBJECT_HEADER SecurityDescriptor

The **!sd** extension will then display the DACL and SACL in the security descriptor

- Add **flag 0x1** after the pointer and you'll get SID->Name translation too

But the pointer is actually a **fast reference**

- Bottom 3 (x86/ARM) or 4 (x64/ARM64) bits store a reference count

We must align it first:

- ```
  $$ T1 is the Object Header
  !sd @@(((unsigned int64)@$t1->SecurityDescriptor & ~0xF)) 1
  ```

We are particularly interested in DACLs that are either **empty**, or which grant "**Everyone**" RW access

- No way to filter at the extension level, we need **.foreach** or a custom script
  ```

# Finding the Owner

When we talk about the “owner” of an object, there’s really two things we could be talking about

The “Owner SID”, which is a security concept, stored in the security descriptor

- The owner **does not imply creatorship**. It is merely used to allow certain additional permissions to the SID when object access is attempted. **Anyone can set the owner to someone else when creating a new object.**

The actual “**Creating Process**” which Windows does not normally track…

- But we have the **+otl** flag!

But remember that for section objects, the **Segment** does also independently track the creator!

# Creator Lookup with Header

Get the **Object Header** for the **Section**, get the **Object Creator Info**, and finally get the **creator PID**, feeding it into **!process**:

- ```
  $$ Section Object in T0
  !process @@(((nt!_OBJECT_HEADER_CREATOR_INFO*)((unsigned int64)#CONTAINING_RECORD(@$t0, nt!_OBJECT_HEADER, Body) - sizeof(nt!_OBJECT_HEADER_CREATOR_INFO)))->CreatorUniqueProcess) 0
  ```

Or if you already have the header…

- ```
  $$ Section Object Header in T1
  !process @@(((nt!_OBJECT_HEADER_CREATOR_INFO*)((unsigned int64)@$t1- sizeof(nt!_OBJECT_HEADER_CREATOR_INFO)))->CreatorUniqueProcess) 0
  ```

# Creator Lookup with Segment

Get the **Segment** from the **Section Object,** then dereference **u1.CreatingProcess**

- dt nt!_SEGMENT u1.CreatingProcess

In **Windows 10**, the **Section Object** has a pointer to the **Control Area** at **u1.ControlArea**

- dt nt!_SECTION u1.ControlArea

So in Windows 10:

- ```
  $$ Assumes Section Object in T0
  !process @@(@$t0->u1.ControlArea->Segment->u1.CreatingProcess) 0
  ```

In Windows <10, the **Section Object** has a pointer to the Segment

- And the nt!_SECTION type is not in the symbols!
- You can use nt!_SECTION_OBJECT, and dereference the **Segment** field
- **CAREFUL:** Type is nt!_SEGMENT, not nt!_SEGMENT_OBJECT as shown
  ```

# Security Descriptor's Owner

Get the **Owner** from the **Security Descriptor,** then add the value to the Security Descriptor pointer

- This is because it is stored in **relative** form
- dt nt!_SECURITY_DESCRIPTOR_RELATIVE Owner

The resulting pointer is a **SID** structure

- dt nt!_SID

We can use the **!sid** extension to dump it, and pass flag **0x1** to perform SID->Name conversion

- ```
  $$ Section Object Header in T1
  !sid @@(((nt!_SECURITY_DESCRIPTOR_RELATIVE*)((unsigned int64)@$t1->SecurityDescriptor & ~0xF))->Owner + ((unsigned int64)@$t1->SecurityDescriptor & ~0xF)) 1
  ```

Might display either a **user** (Bob) or a **group** (Administrators)
  ```

# Creator Process To Account

The creator process is interesting for many purposes, but doesn't quickly tell us if the section object is worth attacking

- Process could "sound" important but run with limited privileges, etc…

If we take a look at the **Token** of the Process, this will tell us the **User Account** that it's running as

- The **Token** field in **EPROCESS** stores a **fast reference** to the token object

The **!token** extension will display and format a token, and the **–n** flag will convert SIDs to names

- We're interested in SYSTEM/LOCAL SERVICE, for starters

Again, filtering is not possible through the extension, so we have to use **.foreach** or write a custom script

- The **UserAndGroups** field in the **Token** stores the SID of the user account
- `!sid @@(@$t1->UserAndGroups[0].Sid) 1`

# Good Script Starting Point…

Assuming **+otl** was enabled in the Global Flags, use **!list** to enumerate over all the section objects

For each entry:

- Filter out **named** sections only
- Filter out **Image**, **Data**, or **Physical** sections (they don't have an owner process and are not relevant)
- Print **Object Creator** PID (from **Header**)
- Print **Section Creator** PID (from **Segment**)
- Print **User Account** of Section Creator PID (from **Token**)
- Check if **DACL** is NULL (from **Security Descriptor**) and print a **WARNING**
- If **DACL** is != NULL, parse each **Access Control Entry (ACE)**
  - For each **ACE**, print the **SID**
  - Check if the **ACE** is S-1-1-0 (**Everyone**), and if the access is **SECTION_ALL_ACCESS** (0xF001F) and print a **WARNING**

# Time To Make Your Eyes Bleed

```
r? @$t0 = (nt!_OBJECT_HEADER_CREATOR_INFO*)@$extret
r? @$t1 = (nt!_OBJECT_HEADER*)(@$t0 + 1)

.if (@@(@$t1->InfoMask & 0x2))
{
    r? @$t2 = (nt!_OBJECT_HEADER_NAME_INFO*)@$t0 - 1
    r? @$t4 = (nt!_SECURITY_DESCRIPTOR_RELATIVE*)((unsigned int64)@$t1->SecurityDescriptor & ~0xF)
    r? @$t5 = (nt!_SECTION*)&@$t1->Body

    .if (@@((@$t5->u1.ControlArea->u.Flags.Image == 0) && \
            (@$t5->u1.ControlArea->u.Flags.File == 0) && \
            (@$t5->u1.ControlArea->u.Flags.PhysicalMemory == 0)))
    {
        r? @$t6 = @$t5->u1.ControlArea->Segment->u1.CreatingProcess
        r? @$t7 = (nt!_TOKEN*)(@$t6->Token.Value & ~0xF)

        .printf /D "\nOBJECT: %p [<b>%msu</b>] K: %d\n", @$t1, @@(&@$t2->Name), @@(@$t1->KernelObj
        .printf /D "\tPID: <b>%lx</b> EPROCESS_PID: <b>%lx</b>\n", @@(@$t0->CreatorUniqueProcess),
        .printf /D "\tOWNER "
        !sid @@((unsigned int64)@$t4 + @$t4->Owner) 1
        .printf /D "\tTOKEN\t"
        !sid @@(@$t7->UserAndGroups[0].Sid) 1

        .if (@@(@$t4->Dacl == 0))
        {
            .printf /D "<col fg=\"changed\">\t<b>WARNING!!! WARNING!!! WARNING!!!</b></col>\n"
        }
        .else
        {
            r? @$t8 = (nt!_ACL*)((unsigned int64)@$t4 + @$t4->Dacl)

            $$>< c:\insection\aclcheck.txt
        }
    }
}
```

# Live Results…

```
OBJECT: ffffc0016fbac5a0 [RTKAPO_NSM_AEC{d9945156-9691-442f-99a2-1b94e0005708}]
        PID: 1024 EPROCESS_PID: 1024
        OWNER SID is: S-1-5-19 (Well Known Group: NT AUTHORITY\LOCAL SERVICE)
        TOKEN  SID is: S-1-5-19 (Well Known Group: NT AUTHORITY\LOCAL SERVICE)
        DACL SID is: S-1-1-0 (Well Known Group: localhost\Everyone) with ACCESS: f001f
        WARNING!!! WARNING!!! WARNING!!!
```

```
OBJECT: ffffc0016d9a24e0 [FloresvilleIWMSInterface_Vista]
        PID: 6cc EPROCESS_PID: 6cc
        OWNER SID is: S-1-5-18 (Well Known Group: NT AUTHORITY\SYSTEM)
        TOKEN  SID is: S-1-5-18 (Well Known Group: NT AUTHORITY\SYSTEM)
        WARNING!!! WARNING!!! WARNING!!!
```

```
OBJECT: ffffc0016dadeed0 [FoxitCloudUpdateSvcShareMemoryForPhantomPDF]
        PID: 878 EPROCESS_PID: 878
        OWNER SID is: S-1-5-32-544 (Alias: BUILTIN\Administrators)
        TOKEN  SID is: S-1-5-18 (Well Known Group: NT AUTHORITY\SYSTEM)
        DACL SID is: S-1-1-0 (Well Known Group: localhost\Everyone) with ACCESS: f001f
        WARNING!!! WARNING!!! WARNING!!!
```

```
OBJECT: ffffc0016ef03570 [S-1-5-21-3704014639-4232294138-2895164880-1001UserDataHostPidMemory]
        PID: 1150 EPROCESS_PID: 1150
        OWNER SID is: S-1-5-21-3704014639-4232294138-2895164880-1001 (User: SAMMY-PC\Alex Ionescu)
        TOKEN  SID is: S-1-5-21-3704014639-4232294138-2895164880-1001 (User: SAMMY-PC\Alex Ionescu)
        DACL SID is: S-1-5-18 (Well Known Group: NT AUTHORITY\SYSTEM) with ACCESS: f001f
        DACL SID is: S-1-1-0 (Well Known Group: localhost\Everyone) with ACCESS: f001f
        WARNING!!! WARNING!!! WARNING!!!
        DACL SID is: S-1-15-2-1 (Well Known Group: APPLICATION PACKAGE AUTHORITY\ALL APPLICATION PACKAGES) with ACCESS: f001f
```

# Fuzzing Sections

# Using WinDBG for Analysis

Easiest way (if it works), is to use the local kernel debugger (we have access to all processes this way) and **fill** the section object's memory with **garbage/41414141/etc**

First, we have to **attach** to the right process:

- `.process /P <creator process>`
- `.reload /user`

Then **dump** the section memory to see what's there:

- `dc <Segment->u2.FirstMappedVa>`

Also use the **!pte** extension to see the **page permissions**

- Bi-winning if the memory is **executable**

This might not work if the creator never **mapped** the section…

- Or has since **unmapped** it (and maybe mapped **elsewhere**)

# Using WinDBG for Fuzzing

We can use the fill (**f**) command in WinDBG to write some garbage…

- `f <virtual address> 41 L1000` to write a page's worth of 414141…

This **won't work** from the local kernel debugger! ☹

- It will work if you're remotely attached, or if you're in user-mode

Workaround is to use **physical** memory address

- Read the page frame number from **!pte** and multiply by **@$pagesize**
- Then use **fp** instead of **f**

This will work and since we're on a live system TLB flush will happen on its own anyway

Check for any **changes** to the process

- Strange memory consumption, **crash**, etc
- Having user-mode WinDBG attached can easily display **exceptions** / DbgPrint

# Caveat Emptor

Do this on most processes and absolutely **nothing** will happen

Put yourself in the software engineer's shoes

- How does the **reader** know that the memory has been **written** to (Hint: the answer is not "hardware memory breakpoints")?
- Likewise, how does the **writer** know that it can write **more** (**reader** is **finished** reading)

Some **synchronization object** must obviously be used

- Except when ALPC is doing it for you – but these are named sections
- Could be a **mutex**, **event**, **semaphore**

If the permissions of the **synchronization object** aren't equally as weak as the section object, you may not be able to achieve **instant** exploitation…

- Or perhaps **no** synchronization object is used at all

# More Complex Scenarios

No **synchronization object**?

- Maybe a **timer** is employed: every n seconds, some thread reads changes in the memory structure
- Maybe some other **external source** (**I/O** is completed, or **window message** is received) is causing the thread to check the memory

No access to **synchronization object**?

- Perhaps you can r**e-create the scenarios** that are normally required by the privileged client to signal the object on your behalf
- This is likely going to **overwrite** your fuzzed data with authentic data, however

However, in the vast majority of cases, there's a **synchronization object**, and it's usually as secure as the section object itself

# Live Case Studies

# Floresville Interface

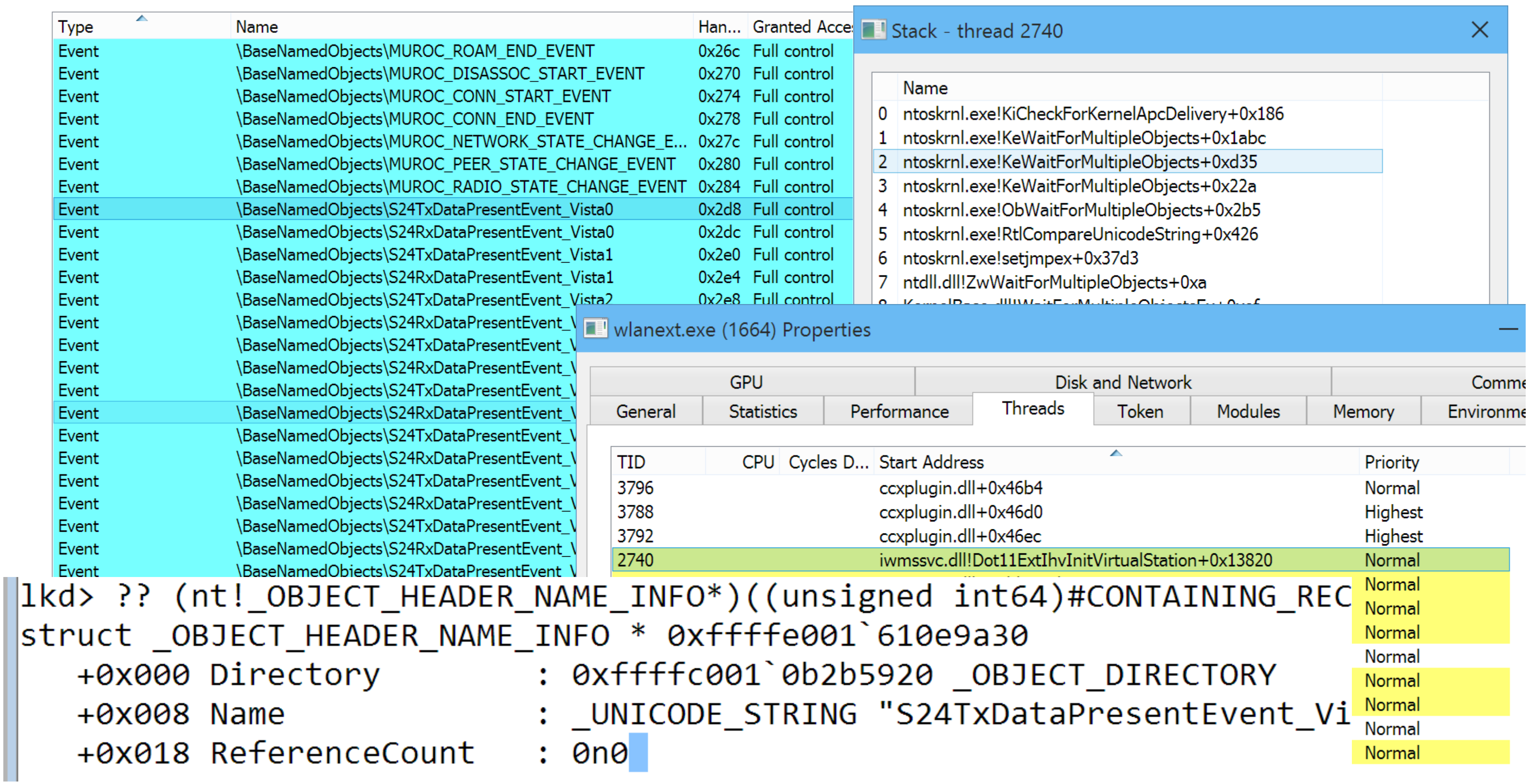
Owned by **wlanext.exe** (“Windows Wireless LAN Extension Host”)
- However, not a Windows/Microsoft bug per se – this is Intel’s Extension

Appears to contain **encrypted data**

Reverse engineering shows:
- **10 equally-sized buffers**, each one associated with its own event
- One thread **waits** on all the 10 events
- Each buffer is **~1794 bytes**, and the entire shared memory is allocated x10 that size
- Thread copies from offset SignaledEvent*1794, and always copies 1794 bytes
- Sends a **window message**, goes back to **waiting**

Another thread wakes up to **handle the window message**
- **Decrypts** the shared memory buffer (bails out if decryption failed)
- **Parses** it based on various encoded **commands** (each with their payload)

# Floresville Interface



# Floresville Interface

May allow for **information disclosure**

**May** have subtle parsing bugs

But ultimately requires a lot of **effort** to fuzz since it requires **encrypting** the payload (otherwise it gets dropped)

Let's see if we have more success with **other** sections…

# PhantomPDF Interface

Owned by **FCUpdateService.exe** (“Foxit Cloud **Safe Update** Service”)

Appears to contain **JSON data** when populated, but **usually empty**

Is present even if user **unchecks** the option to install Phantom PDF (Foxit’s Cloud storage service)

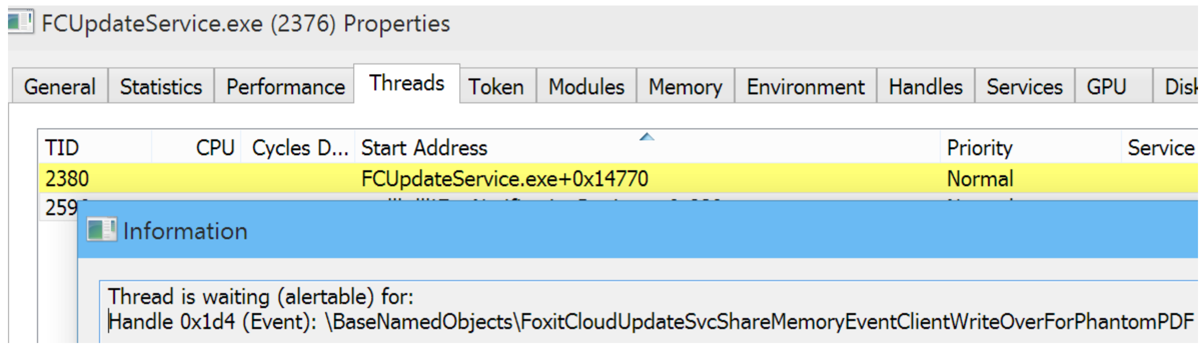
First field is the **size** of the buffer

- Directly **trusts** it and uses it during **memcpy** operations
- Also immediately **assumes** data is correctly **formatted** JSON and begins parsing it

Not only is it **trivial** to **exploit**, but I also wonder if the JSON payloads themselves can be legitimately laid out but malicious in nature

- i.e.: do a JSON payload attack, not fuzzing the JSON data/memory itself

# PhantomPDF Interface




| | | |
|---|---|---|
| Key | HKU\.DEFAULT\Software\Microsoft\Windows\CurrentVersion\Explorer | 0x170 |
| Directory | \BaseNamedObjects | 0xa8 |
| Event | \BaseNamedObjects\FoxitCloudUpdateSvcShareMemoryEventClientWriteOverForPhantomPDF | 0x1d4 |
| Event | \BaseNamedObjects\FoxitCloudUpdateSvcShareMemoryEventServerWriteOverForPhantomPDF | 0x1d8 |
| Section | \BaseNamedObjects\FoxitCloudUpdateSvcShareMemoryForPhantomPDF | 0x1c8 |
| Mutant | \BaseNamedObjects\FoxitCloudUpdateSvcShareMemoryMutexForPhantomPDF | 0x1cc |

```
(948.a24): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
FCUpdateService+0x14a97:
00984a97 f3a5            rep movs dword ptr es:[edi],dword ptr [esi]
0:001:x86> db edi
018fc018  00 00 00 00 9c 00 93 00-9c 00 93 00 00 00 00 00  ................
018fc028  00 00 00 00 00 50 41 41-00 c0 41 41 d2 2f 7d c4  .....PAA..AA./}.
```

# RTK APO NSM AEC Interface

Owned by **Audiodg.exe** (“Windows Audio Device Graph Endpoint”)
- However, not present by default… not a Windows component

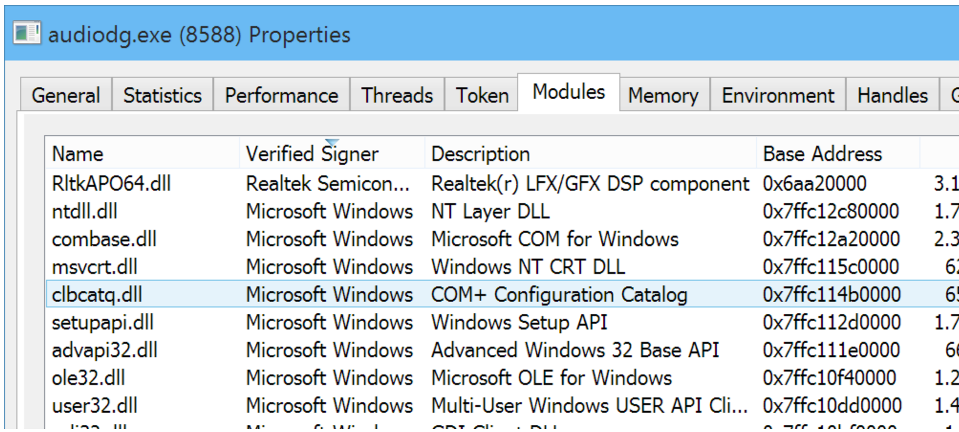
**APO** = Audio Processing Objects
- **COM Interface** through which 3rd party sound card vendors can apply DSP
- Likely my **sound card manufacturer** (Hint: R***t*k)
  - Or maybe it’s a Rootkit

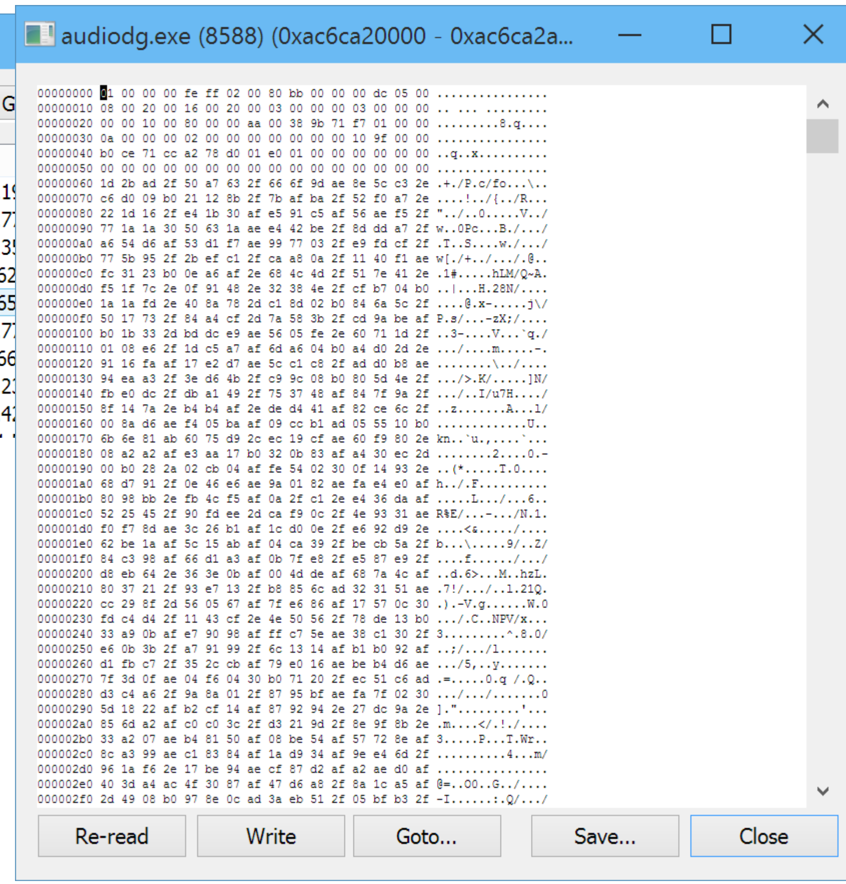
Couldn’t find an obvious **synchronization object** that it’s associated with

The related threads in Audiodg.exe are each waiting on 64, 48, 36 threads, respectively (and there are two pairs of each)

So, use another technique to discover use:
- Set **hardware breakpoint** on memory and see when it’s used!

# RTK APO NSM AEC Interface




```
Breakpoint 0 hit
msvcrt!memmove+0x1cf:
00007ffc`11633c0f 75ef            jne     msvcrt!memmove+
0:006> dd 00000072`26840000
00000072`26840000  fffffffe ffffffff ffffffff ffffffff
00000072`26840010  ffffffff ffffffff ffffffff ffffffff
00000072`26840020  ffffffff ffffffff ffffffff ffffffff
00000072`26840030  ffffffff ffffffff ffffffff ffffffff
```

# RTK APO NSM AEC Interface

Ultimately was able to achieve a **crash**

- But **not** under **repeatable** circumstances

Only discovered this interface by chance on the plane on the way here…

- Doesn’t show up until you actually **play music** using Windows Media Player
  - Or **watch a movie**

Very interested to pursue research on this particular object, since it could allow for attack against **protected processes**

- See my blog if unfamiliar with the concept of **Protected Process Light**
- No symbols for latest Windows 10 build are making things hard
- And can’t exactly run this in a virtual machine

But ultimately, we are going to look at an even **more interesting** class of section objects, so let’s leave **DRM** world behind for now…

# Closing Remarks on Methodology

The script currently only looks for “**Everyone**” being granted **SECTION_ALL_ACCESS** or for a **NULL DACL**

These are the most egregious ways of having insecure section objects, but not the only ones

- Granting **WRITE_DAC**, **SECTION_MAP_WRITE** is just as bad
- Granting access to the “**Users**” group is also bad
  - Or “**Remote Users**” or “**Authenticated Users**”…
- Even granting access to the **current user** can be bad

We didn’t look for **ALL_APPLICATION_PACKAGES** either

- This is the “**Everyone**” of the **AppContainer** world (Windows 8 **sandboxes**)
- AppContainers are vulnerable to **NULL DACL**, but not to “**Everyone**” alone

With some improvements, you can probably find more vulnerabilities

# Kernel Exploitation

# Kernel Owned Section Objects

The API to create named section objects can also be used by **drivers**

- Will typically be owned by **SYSTEM** in the **Security Descriptor**

How can we identify such **kernel-created** section objects?

- Looking for **PID 4** ("System Process") is one way, but the Windows Kernel runs in **arbitrary** process address spaces when syscalls or interrupts happen
- So a driver could've created the section object while in **another context**
- Instead, we can use a few **artifacts** of kernel-created sections…

Note in the script output, some **EPROCESS_PIDs** did not match **PIDs**

- Why would the object header have a different PID than the segment?
  - **Segment** tracks **EPROCESS**, object **header** tracks **PID**…
- Kernel code can "**attach**" to a process to access its memory, handle table, etc

**PsGetCurrentProcess** returns "Current" (**Attached**) Process, **PsGetCurrentProcessId** returns thread's "Owner" (**Original**) Process ID.

- **ReactOS gets this wrong!!!**

# Kernel Created Objects

But the kernel doesn't necessarily have to be attached, so the artifact may not be **visible**

And the artifact can also false-positive if the EPROCESS is **re-used** after the creator process has died but the section has been **kept around** by another mapper

- **PID** of **dead** process and **PID** of **new** EPROCESS will now **mismatch**
- In pathological cases, both PID and EPROCESS will still match but point to an unrelated process

A better way is to use the Object Header's **KernelObject** flag

- Set **automatically** whenever a component in kernel-mode is creating an object

We can extend the hunting script to also **identify** kernel-created Objects

- Reveals the existence of a **\Win32kCrossSessionGlobals** structure

But only allows **SYSTEM/Administrators** to modify it. Is this a problem?

# The Debate

A company spends millions of dollars, resources, and lines of code to create a system where

- Kernel code must be **signed** (by the **company itself**, in Windows 10!)
  - With a **SHA-256 EV Certificate**, in Windows 10
- BIOSes are deprecated in favor of UEFI with **Secure Boot**
  - Preventing custom OS/boot loaders from working, even if you have physical access to machine
- Certain processes are **protected** so that not even an **Administrator** can modify them
- On certain platforms, **no 3rd party** user-mode code is **allowed** to run

….and many other mitigations to **prevent Administrators** from **arbitrarily** executing kernel-mode code without TPM measurement, ELAM authorization, SHA-256 EV Code Signing from Microsoft Root CA

And then you **find a way** to get arbitrary kernel-mode code execution which bypasses every single protection put into place

- **Is it a bug**? If not, then why all these mechanisms? Only admins can load drivers. All these mechanisms are meant to **prevent admins** from doing that.

# The "Cross-Session Globals"

Defined by a structure which contains 3 main storage containers

Network Font Table and Semaphore Lock
- Used whenever a font image file is being loaded from a remote (UNC) share
- Triggered when the kernel font mapper parses fonts or the font cache
- Couldn't find obvious way to trigger from user mode

**Trusted Font Table** and Semaphore Lock
- Used whenever a **font** is being **added** to the current process' resource table
- **Inserted** into when the undocumented "**Trusted Font**" flag is used, **queried** when the flag is not used
- Easy to trigger both paths by using **NtGdiAddFontResourceW** exported as **GdiAddFontResourceExW**

Session Pool Tracker Table
- Off by default, can be enabled with undocumented registry key
- Not really exploitable, but can leak addresses if turned on

# Windows AVL Tables

Windows implements a "Run-time Library", or Rtl which provides, among other things, an **AVL Tree/Table Package**

- Self-balancing binary search tree using the AVL algorithm

An **AVL Table** is described by the following structure:

```
lkd> dt nt!_RTL_AVL_TABLE
   +0x000 BalancedRoot      : _RTL_BALANCED_LINKS
   +0x020 OrderedPointer    : Ptr64 Void
   +0x028 WhichOrderedElement : Uint4B
   +0x02c NumberGenericTableElements : Uint4B
   +0x030 DepthOfTree       : Uint4B
   +0x038 RestartKey        : Ptr64 _RTL_BALANCED_LINKS
   +0x040 DeleteCount       : Uint4B
   +0x048 CompareRoutine    : Ptr64     _RTL_GENERIC_COMPARE_RESULTS
   +0x050 AllocateRoutine   : Ptr64     void*
   +0x058 FreeRoutine       : Ptr64     void
   +0x060 TableContext      : Ptr64 Void
```

# Overwriting Compare Callback

With RW access to the section object, we can **replace** the Trusted Font Table with our **own** table

- Ours will contain a **custom compare** callback function, which we control
- The kernel's AVL Table functions will **call** it at **Ring 0** as soon as we try to **load** a font
- WIN!

Except for **SMEP (Supervisor Mode Execution Prevention)**

- Expressly designed to handle lazy weaponization of similar exploits
- CPU will **not allow execution** of CPL 0 code stored in CPL 3 pages (#PF raised)

We need to find a way around SMEP on modern processors

- And traditional way of using **ROP Gadgets** won't help here, as we don't have **stack control**
- **Heap-spraying** or heap **control** works nicely on Windows 7 or x86 Windows 8 by leveraging Executable Big Pool (NonPaged/SessionPaged), but **not x64**.

# Convert CodeExec into W-w-w

The “lazy way” is often the “wrong way” with SMEP.

We need to convert our code execution into a **Write-what-where**, and we don’t have ROP gadgets.

Instead, we use an **existing** function to perform a controlled write
- Can make things more complicated and use multiple functions that are called in order by re-exploiting the bug
- In some rare cases, can even use gadgets through **relative offsets (“JOP”)**
  - However, **CFG** will **protect** against that on **Windows 10**

Difficulty is that AVL Compare Routine has a specific set of **parameters**, and must return a **specific** set of possible **return** values
- Function must not only perform the required write, but also return correctly

**KASLR** is not an issue here: kernel base is **leaked**, and kernel file can be mapped in user-mode

# Finding the "right" Compare

The **compare routine** has the following signature

```
RTL_GENERIC_COMPARE_RESULTS
(*PRTL_AVL_COMPARE_ROUTINE) (
    __in struct _RTL_AVL_TABLE  *Table,
    __in PVOID  FirstStruct,
    __in PVOID  SecondStruct
    );
```

We control **Table**, and we control **SecondStruct**

- How? Compare routine always starts at the **tree root** (controlled in **Table**) and always first starts by taking its right child (i.e.: **Table->Root.RightChild**)
- Returning **2+** means "**Node Found**", and the **FirstStruct** will be returned.
- Returning **1** means "walk tree down on the **right side**", so **LeftChild** will be dereferenced as the next **SecondStruct**, and function is called again
- Returning **0** means "walk tree down on the **left side**", so **opposite** of above

**FirstStruct** is, in this case, the string containing the font name

# Making things hard…

So Table must have **NumberOfGenericElements** set to 1, and **BalancedRoot.RightChild** must be set to an interesting value.

Preferably, then, the function does something like:

- **mov [SecondStruct], [Table->SomeField]**

Which really expands into

- **mov [Table->Root.RightChild.UserData], [Table->SomeField]**
- i.e.: it's important to make sure that **SomeField** is not already being used to maintain the tree consistent.

Return value is important too: if the function returns **2+**, we are golden, because **win32k!IsTrustedFontPath** only checks for !=NULL, doesn't actually read the data

- But if the function returns **1**, we must make **Table->Root.LeftChild == NULL**
- If the function returns 0, we have a problem, because it will read **RightChild**

# Best we can get…

```
.text:0000000140160F24 ; char __cdecl PopPepStartDevicePowerOnActivity
.text:0000000140160F24 PopPepStartDevicePowerOnActivity proc near
.text:0000000140160F24                                                    ; DATA
.text:0000000140160F24                 mov     dword ptr [r8], 2
.text:0000000140160F2B                 mov     rax, [rcx+20h]
.text:0000000140160F2F                 mov     byte ptr [r8+10h], 1
.text:0000000140160F34                 mov     [r8+8], rax
.text:0000000140160F38                 mov     al, 1
.text:0000000140160F3A                 retn
.text:0000000140160F3A PopPepStartDevicePowerOnActivity endp
```

Reads **Table->OrderedPointer** into RAX

Writes RAX into **SecondStruct+8** (or really **Table->Root.RightChild.UserData + 8**)

Unfortunately, corrupts **SecondStruct+0** and **SecondStruct+16**

# Dealing with Corruption

MWR (first?) publicized a technique leveraging the fact that in Windows, it is **trivial** to find the **paging hierarchy structures** for a given virtual address

- Fixed **PXE_BASE**, **PPE_BASE**, **PDE_BASE**, **PTE_BASE** constants are well known
- The offset be deduced by looking at the virtual address bits
- In fact this is how **!pte** works!

On x64, Windows 8.1 offers **128 TB** of address space, and few processes ever use anywhere close to this

- Meaning that most of the PXE/PML4 entries are **empty** (each PML4 entry covers **512GB**)
- Therefore, by picking an address "**far enough**" in the address space, and making the target of the **Write-what-where** a given **PML4 entry**, nearby **corruption** won't matter as long as nobody attempts to dereference anything within the **adjacent 512GB** blocks
- **Repeat** the process for other entries (PPE, PDE, PTE)

# Improving the Technique

The original technique uses the **Write-what-where four** times for each of the paging structures.

- It describes how **self-referencing** entries work, but **doesn't** actually **leverage** their power!

If we pick a **specially crafted** virtual address, we can make it such that the **PTE** Offset in the Page Table will be **equal** to the **PDE** Offset in the Page Directory, which itself will be **equal** to the **PPE** Offset in the Page Directory Pointer Table, which itself will be **equal** to the **PXE** Offset in the PML4.

For example, **0x40201008000** uses the **universal** offset **0x40**

```
lkd> !pte 0x40201008000
                                           VA 0000040201008000
PXE at FFFFF6FB7DBED040    PPE at FFFFF6FB7DA08040    PDE at FFFFF6FB41008040    PTE at FFFFF68201008040
```

Now only a **single write** is needed, and the PXE will **recursively** be a PTE

# Forging the PTE

In the MWR blog post, they allocate an existing PTE, and flip the required **User** and **NoExecute** bits on/off in order to make the page **Supervisor** and **Executable**

- By using a **misaligned** write, they can flip the NX bit specifically off, while writing the required ON bits at the end of the PTE, with a **single write** that **doesn't corrupt** the page frame number (PFN).

Unfortunately, the particulars of our memory corruption make a **misaligned** address **impossible**

- The writes at **+0x08** and **+0x10** will become **misaligned** writes into our PTE, and **corrupt** it

We can **only** write an 8-byte **aligned** value

- So we have to **forge** a **PTE** from **scratch**, including the **correct PFN** and **Working Set Index**

How do we do that?

# Address Windowing Extensions

Designed for Windows **2000**, which supported **PAE (Physical Address Extension)**

- Intel technology allowed for access of up to **64GB** of RAM on a **32-bit OS**

Virtual address space remained **2GB**, so how could processes map/access the additional RAM?

- API was created to allow **manual** control of **free RAM** pages
- Used by **SQL Server**

Still used **today**, even on 64-bit due to its useful side-effects

- Pages are not in the **working set**, meaning they don't get **paged/cached** or subject to **trimming** (and faster memory operations are allowed)
- Process can fully determine page management (allows for a user-mode memory manager)

**AllocateUserPhysicalPages**, **MapUserPhysicalPages** & **MEM_PHYSICAL**

# How AWE Helps

**AllocateUserPhysicalPages** returns a “**UserPfnArray**”

*UserPfnArray* [out]

A pointer to an array to store the page frame numbers of the allocated memory.

The size of the array that is allocated should be at least the *NumberOfPages* times the size of the **ULONG_PTR** data type.

Do not attempt to modify this buffer. It contains operating system data, and corruption could be catastrophic. The information in the buffer is not useful to an application.

*“MSDN, you just invited Alex Ionescu to take your API apart”*

Bottom line: by **leaking** the address of the **PFNs**, and by **zeroing** out the **working set index**, **AWE** pages’ **PTEs** can be fully **forged** with the needed bits

# Protected Process Bypass DEMO

# There’s One More Thing…

It seems everyone **misunderstands** how **SMEP actually** works

And **AWE** has **one more bug**, which made the entire last 20 minutes useless*

* Except on Windows 10, where it’s **partially fixed**

# What MWR & Others Got Wrong

*"When checking the rights of a page, the kernel will check every PXE involved in the address translation. That means that if we want to check if the U/S flag is set, we will check all entries relating to that page.* ***If any of the entries do not have the supervisor flag set****, any attempt to use this page from kernel mode will trigger a fault.* ***If all of the entries have the supervisor flag set****, the page will be considered a kernel-mode page."*

# What Intel told the NSA…


## Supervisor Mode Execution Protection

Stephen Fischer
Senior Principal Engineer
Intel® Corporation

# Intel Doublespeak

NSA | TRUSTED COMPUTING Conference & Exposition

## SMEP architectural control details

- CR4.SMEP – If 1 and in supervisor mode (CPL<3), instructions may not be executed from a linear address for which the U/S flag is 1 (user mode) in every paging-structure entry controlling the translation for the linear address
    - SMEP U/S paging attribute precedence:
        - Any page level marked as supervisor (U/S=0) will result in treatment as supervisor for SMEP enforcement
            - Existing user/supervisor privileging checking continues to require the more conservative mapping (i.e. execution in user mode (CPL=3) requires all levels to be mapped as U/S=1 (user)

# Just one P*E structure is needed

Because of this, it means that even in a **512GB** address range which is user-mode, and within a **1GB** block that is user-mode, and a **2MB** region that is user-mode, a **single** “**Supervisor**” **4KB** page will **bypass SMEP**

This makes **Write-what-where** vulnerabilities combined with PTE editing an even better deal

- If we didn’t have memory corruption (i.e.: a more standard **Write-what-where**), a single edit at a known **PTE_BASE+PteIndex** offset would be enough to **bypass** SMEP.
- Can easily be done **remotely** as **PTE_BASE** is **known**

But it turns out, thanks to **AWE**, you **don’t** even **need** a **vulnerability**.

AWE pages can have two states:

- **PAGE_READWRITE**: Non-executable, user-mode, RW
- **PAGE_NOACCESS**: How can we **emulate** “no access”?!

# PAGE_NOACCESS on AWE Pages

Normally, **emulating PAGE_NOACCESS** means turning the **Valid** bit off in the **PTE**, and using some **Software PTE** bits, combined with the **VAD**, for the memory manager to understand this is a "real" allocation, but marked as **inaccessible**.

- But **AWE** pages are "**invisible**" or "**unmanaged**" by the memory manager, and it **isn't easy** to replicate this logic without major changes

So a simpler trick is used instead…

- The kernel maps the page **supervisor**
- This enforces "**no access**" as far as **user-mode** is concerned
- But it also means one of your user page is now **ring 0**!

Even better, the internal array that determines which Win32 permissions (**PAGE_READWRITE** vs **PAGE_EXECUTE_READWRITE**) should flip the **NX** bit **on**, do not include **PAGE_NOACCESS**.

- Kernel will flip the **NX** bit **off**. You get **supervisor, executable** memory.

# QUESTIONS?

THANK YOU!