

# Exploiting Hardcore Pool Corruptions in Microsoft Windows Kernel

Nikita Tarakanov  
Anonymous Developer

# Who the heck is Nikita Tarakanov?

- Independent Russian Security Researcher
- Vulnerability Assassin
- Crazy Wild Russian
- Aligner of stars
- Nice dude 😊

# Agenda

- Introduction/Kernel Pool Basics
- Previous research
- TBD
- Conclusion
- Q&A

# Introduction

- Many modern popular applications have sandbox
- Sandboxes have low attack surface
- Attacking kernel from the sandbox is convenient
- Untrusted -> r0 -> full compromise RULEZZZ (Nils (@nils) and Jon (@securitea) vs Google Chrome at pwn2own 2013)

# Introduction

- Most of vulnerabilities in MS kernel are memory corruptions
- Most of them are Pool Corruptions
- MS enhances security of Pool Allocator
- Windows 7 – “Safe” unlinking
- Windows 8 – almost every technique is dead

# Kernel Pool research MUST READ

- Following slides are basics of Kernel Pool mechanisms
- Read slides of Tarjei Mandt aka @kernelpool which is the most comprehensive work on Kernel Pool Internals
- Newest research by Zhenhua 'Eric' Liu at BH EU 2013 about advanced Pool Manipulation techniques on win8

# Kernel Pool Basics

- Kernel pools are divided into types: Non-Paged, Paged, Session, etc.
- Each kernel pool is defined by a pool descriptor (**POOL\_DESCRIPTOR** structure)
- The initial descriptors for paged and non-paged pools are defined in the **nt!PoolVector** array

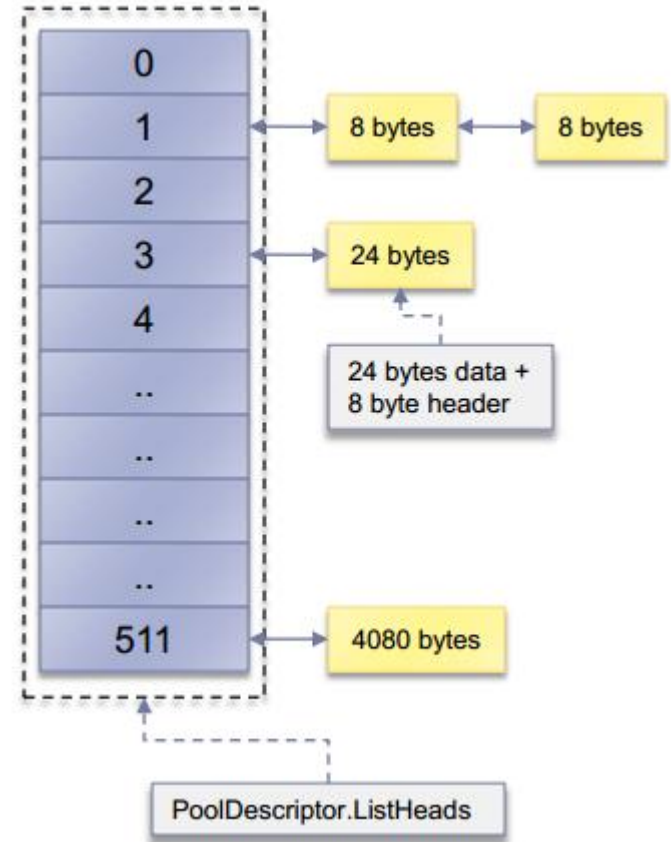
# Kernel Pool Descriptor (Win 8 x86)

- dt nt!\_POOL\_DESCRIPTOR
- +0x000 PoolType : \_POOL\_TYPE
- +0x004 PagedLock : \_FAST\_MUTEX
- +0x004 NonPagedLock : Uint4B
- +0x040 RunningAllocs : Int4B
- +0x044 RunningDeAllocs : Int4B
- +0x048 TotalBigPages : Int4B
- +0x04c ThreadsProcessingDeferrals : Int4B
- +0x050 TotalBytes : Uint4B
- +0x080 PoolIndex : Uint4B
- +0x0c0 TotalPages : Int4B
- **+0x100 PendingFrees : \_SINGLE\_LIST\_ENTRY**
- +0x104 PendingFreeDepth : Int4B
- **+0x140 ListHeads : [512] \_LIST\_ENTRY**



# ListHeads

- Each pool descriptor has a ListHeads array of 512 doubly linked lists of free chunks of the same size
- Free chunks are indexed into the ListHeads array by block size
- Each pool chunk is preceded by an 8-byte **pool header**
- 



# Pool Header (x86)

- `kd> dt nt!_POOL_HEADER`
- `+0x000 PreviousSize` : Pos 0, 9 Bits
- `+0x000 PoolIndex` : Pos 9, 7 Bits
- `+0x002 BlockSize` : Pos 0, 9 Bits
- `+0x002 PoolType` : Pos 9, 7 Bits
- `+0x004 PoolTag` : Uint4B
- `PreviousSize`: BlockSize of the preceding chunk
- `PoolIndex`: Index into the associated pool descriptor array
- `BlockSize`:  $(\text{NumberOfBytes} + 0xF) \gg 3$
- `PoolType`: Free=0, Allocated=(PoolType|2) `PoolTag`: 4 printable characters identifying the code responsible for the allocation

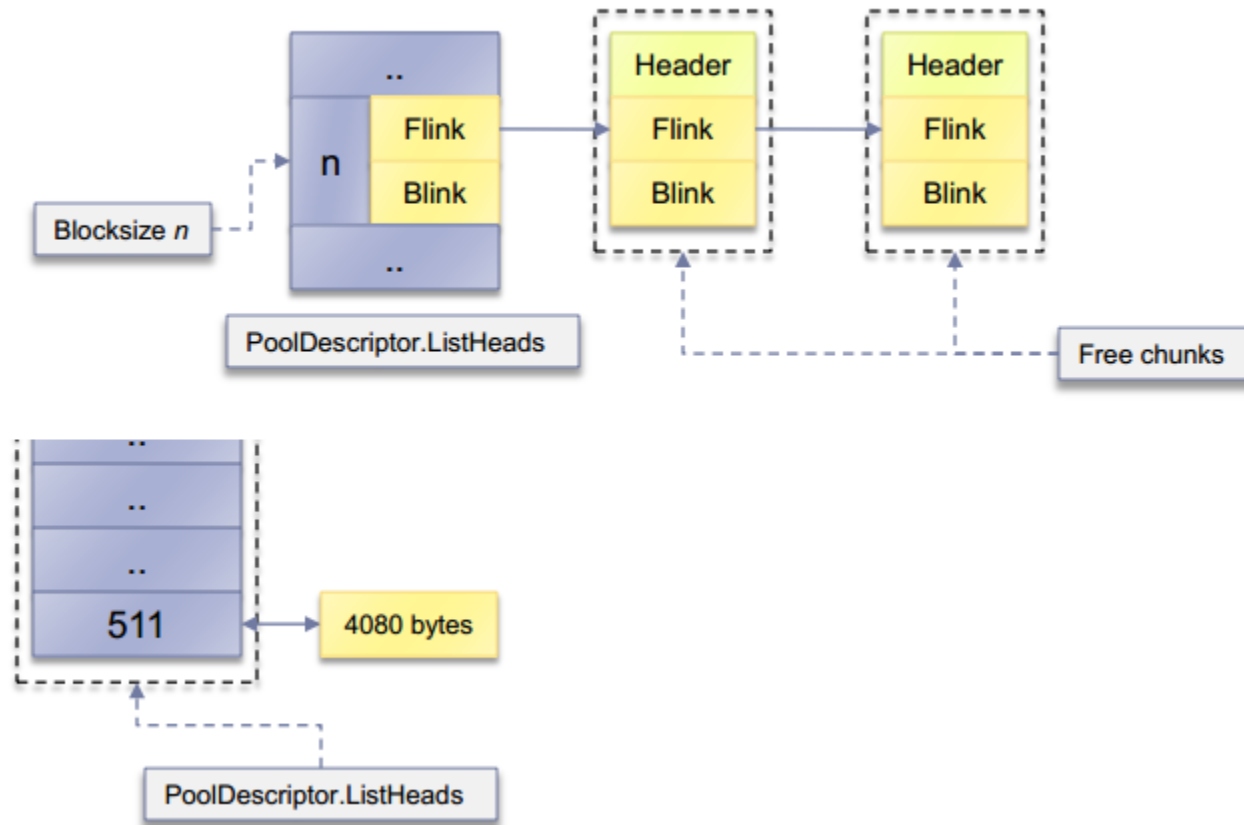
# Pool Header (x64)

- `kd> dt nt!_POOL_HEADER`
- `+0x000 PreviousSize` : Pos 0, 8 Bits
- `+0x000 PoolIndex` : Pos 8, 8 Bits
- `+0x000 BlockSize` : Pos 16, 8 Bits
- `+0x000 PoolType` : Pos 24, 8 Bits
- `+0x004 PoolTag` : Uint4B
- `+0x008 ProcessBilled` : Ptr64 `_EPROCESS`
- `BlockSize`:  $(\text{NumberOfBytes} + 0x1F) \gg 4$  (256 ListHeads entries due to 16 byte block size)
- `ProcessBilled`: Pointer to process object charged for the pool allocation (used in quota management)

# Free Pool Chunks

- If a pool chunk is freed to a pool descriptor ListHeads list, the header is followed by a **LIST\_ENTRY** structure
- Pointed to by the ListHeads doubly-linked list
- kd> dt nt!\_LIST\_ENTRY
- +0x000 Flink : Ptr32 \_LIST\_ENTRY
- +0x004 Blink : Ptr32 \_LIST\_ENTRY

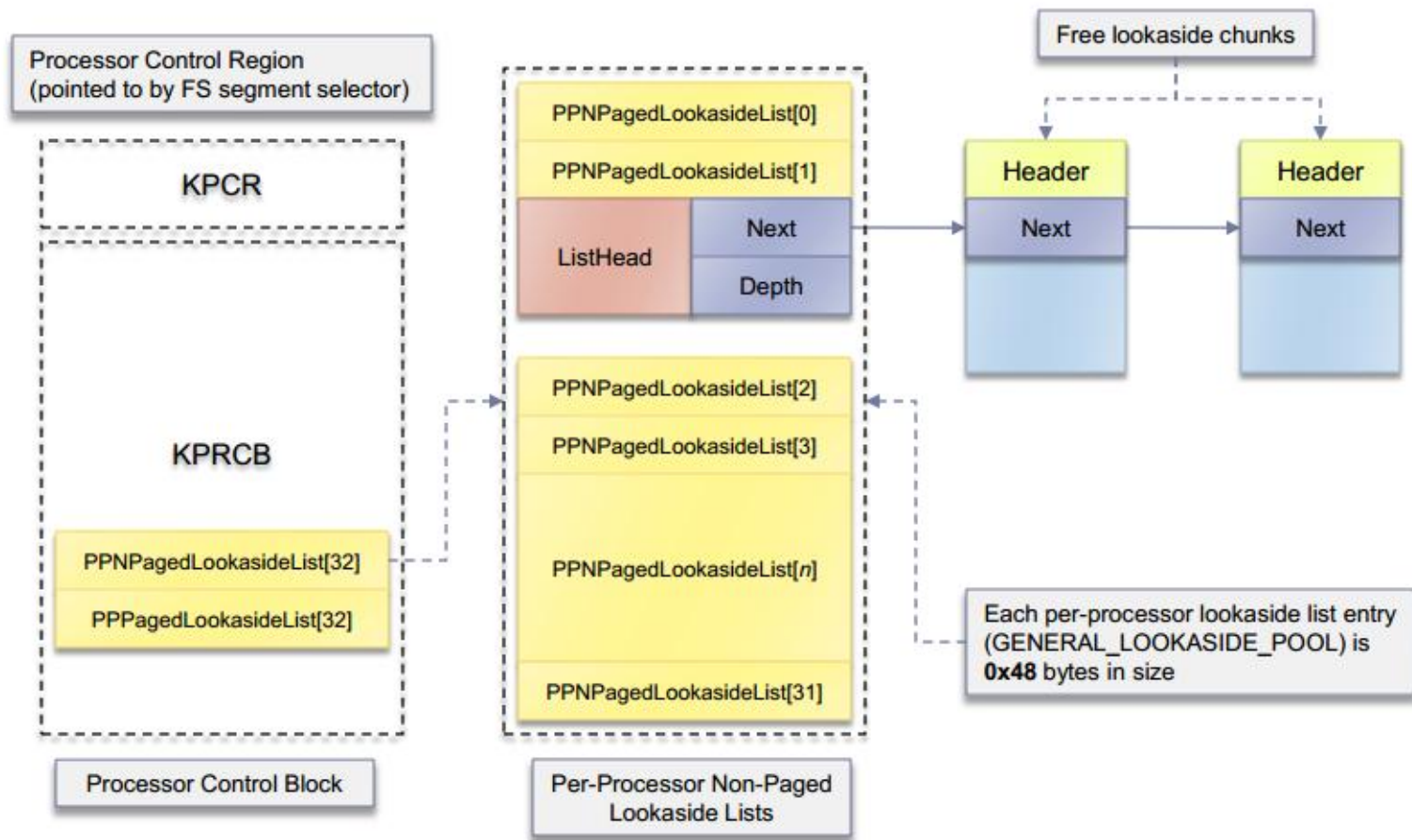
# Free Pool Chunks



# Lookaside Lists

- Kernel uses lookaside lists for faster allocation/deallocation of small pool chunks
- Separate per-processor lookaside lists for pagable and non-pagable allocations
- Defined in the Processor Control Block (KPRCB)
- Maximum BlockSize being 0x20 (256 bytes)

# Lookaside Lists



# Large Pool Allocations

- Allocations greater than 0xff0 (4080) bytes
- Handled by the function `nt!ExpAllocateBigPool`
- Each node (e.g. processor) has 4 singly-linked lookaside lists for big pool allocations
- 1 paged for allocations of a single page
- 3 non-paged for allocations of page count 1, 2, and 3



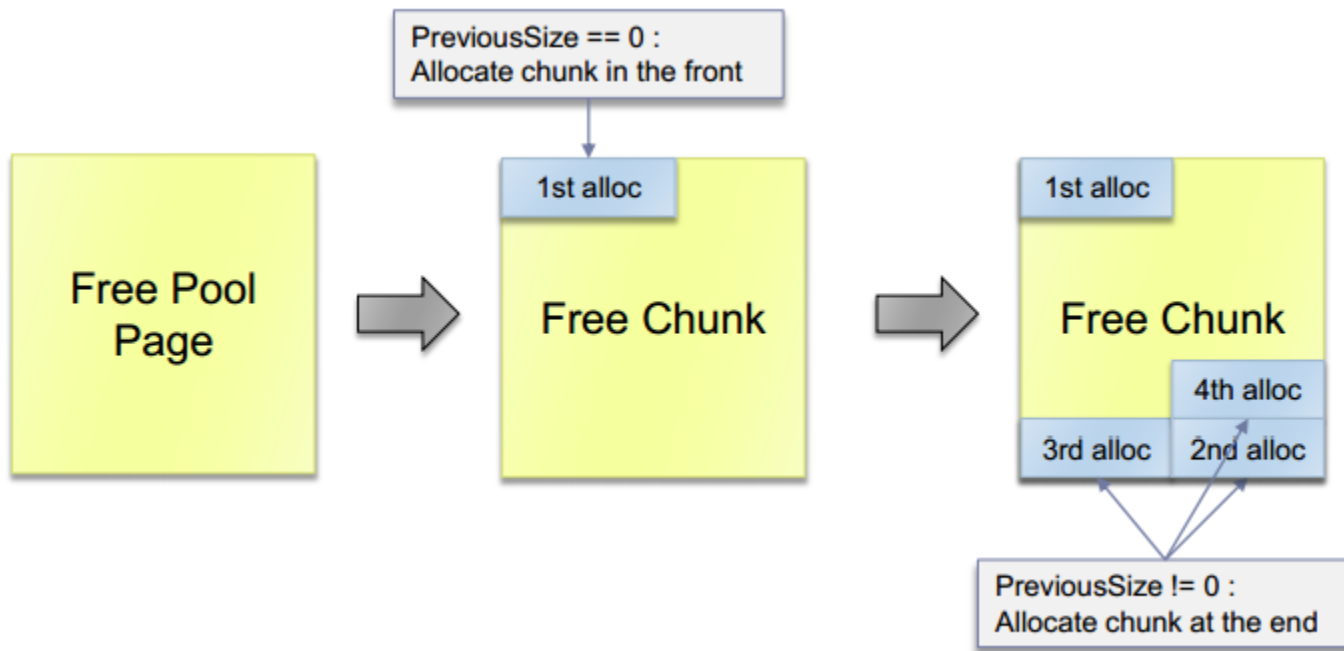
# Large Pool Allocations

- If lookaside lists cannot be used, an allocation bitmap is used to obtain the requested pool pages
- The bitmap is searched for the first index that holds the requested number of unused pages
- Bitmaps are defined for every major pool type with its own dedicated memory
- The array of bits is located at the beginning of the pool memory range

# Allocation Algorithm

- The kernel exports several allocation functions for kernel modules and drivers to use
- All exported kernel pool allocation routines are essentially wrappers for **ExAllocatePoolWithTag**
- The allocation algorithm returns a free chunk by
- checking with the following (in order)
  - Lookaside list(s)
  - ListHeads list(s)
  - Pool page allocator

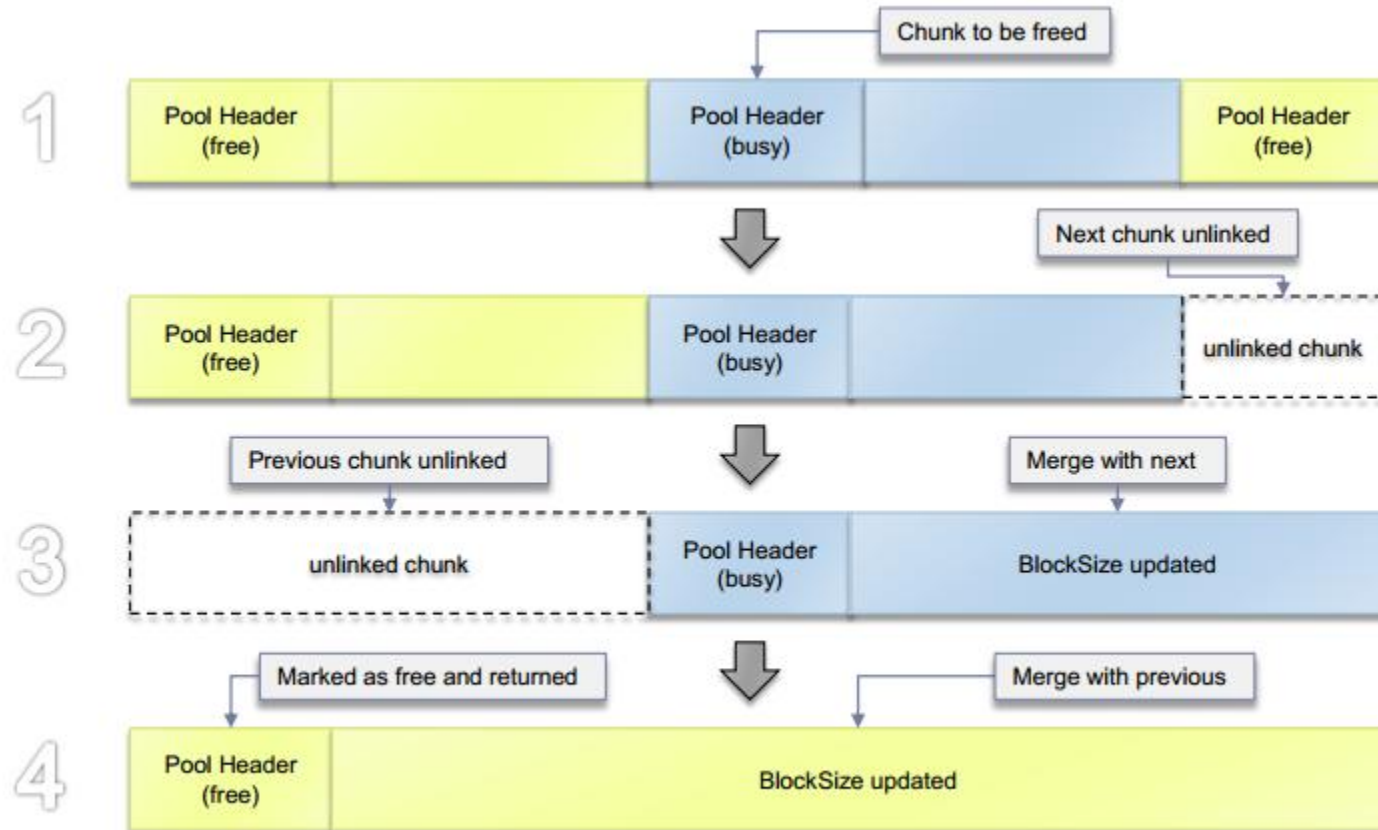
# Splitting on allocation / Order of chunk allocation on page



# Free Algorithm

- The Free Algorithm inspects the pool header of the chunk to be freed and frees it to the appropriate list (**ExFreePoolWithTag** function)
- Adjacent free chunks may be **merged** with the freed chunk to reduce fragmentation

# Coalescence/Merging



# Previous research

- SoBelt X'con 2005
- Kostya Kortchinsky SyScan 2008
- Tarjei Mandt BH DC 2011
- Tarjei Mandt BH US 2012
- Zhenhua 'Eric' Liu BH EU 2013

# Previous research (Kortchinsky )

- write4 techniques:
  - Unlink attack
  - Merge with next
  - Merge with previous
  - Lisheads unlinks
  - MmNonPagedPoolFreeListHead Unlink

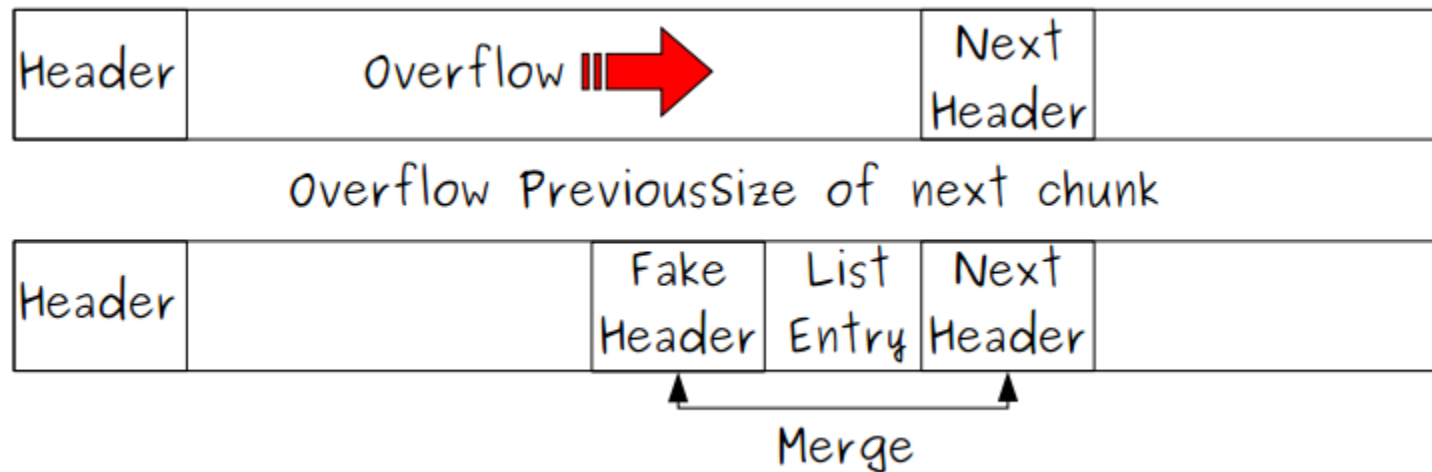
# Kortchinsky

- Removing an entry 'e' from a double linked list:
  - PLIST\_ENTRY b,f;
  - f=e->Flink;
  - b=e->Blink;
  - b->Flink=f;
  - f->Blink=b;
- This leads to a usual **write4** primitive:
  - **\*(where)=what**
  - **\*(what+4)=where**



# Kortchinsky

- Write4 example (happens when next is freed)



# Previous research (Mandt BH DC 2011)

- ListEntry Flink Overwrite
- Lookaside Pointer Overwrite
- PoolIndex Overwrite
- PendingFrees Pointer Overwrite
- Quota Process Pointer Overwrite

# Previous research (Mandt BH US 2012)

- MS eliminated Tarjei's techniques in win8
- Tarjei discovered more l33t stuff for win8:
  - BlockSize Attack
  - Split Chunk Attack

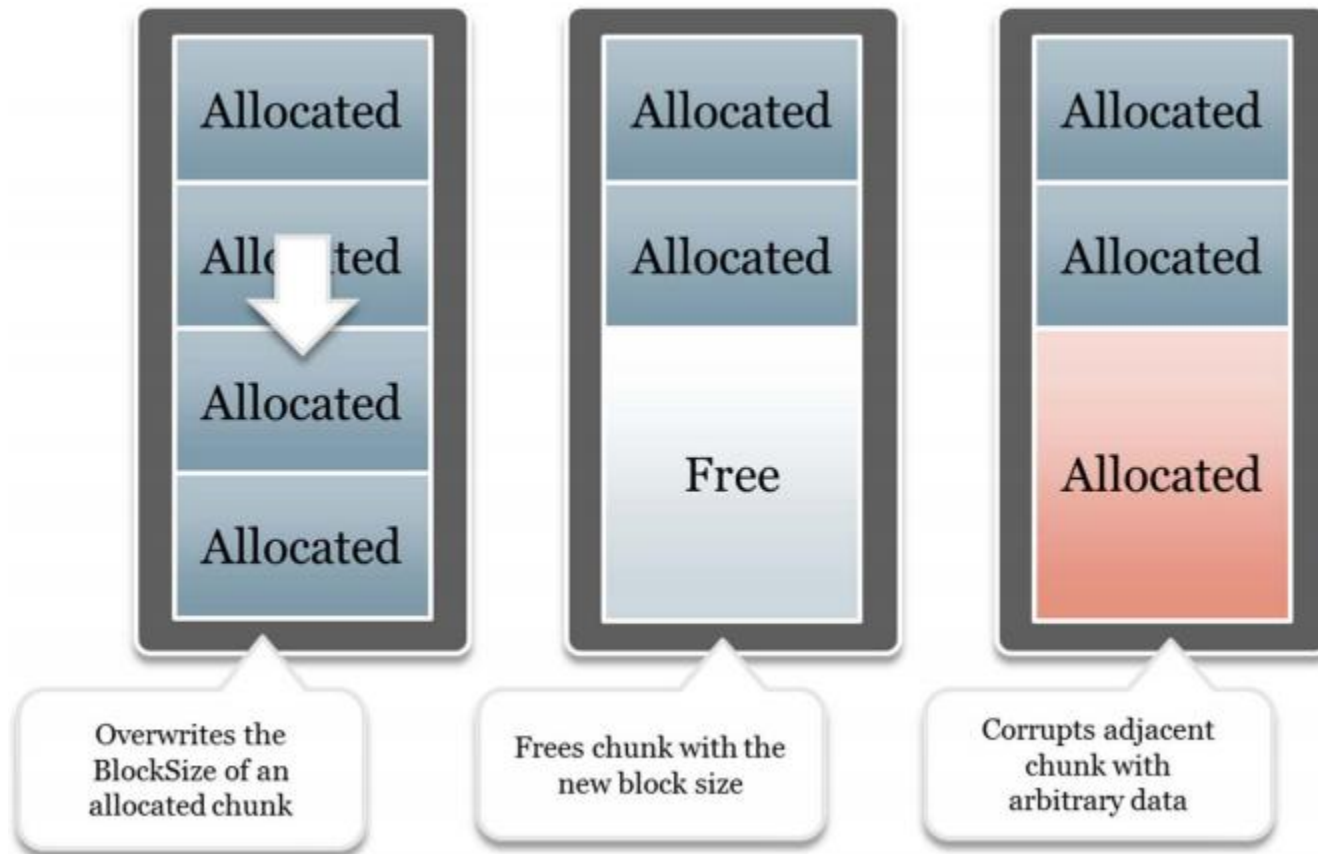
# BlockSize Attack

- When a chunk is freed, it is put in to a free list or lookaside based on its block size
- An attacker can overwrite the block size in order to put it into an arbitrary free list
- Setting the block size to cover the rest of the page **avoids** the **BlockSize/PreviousSize check on free** (no checks -> no BSOD)

# BlockSize Attack Steps

- Corrupt the block size of an in-use chunk (Set it to fill the rest of the page)
- Free the corrupted pool chunk
- Reallocate the freed memory using something controllable (like a unicode string)
- It leads to arbitrary pool corruption

# BlockSize Attack



# Previous Research (Summary)

- Attacks against Pool metadata/mechanisms
- Advanced Pool Manipulation (Feng Shui)
- Precise control over overflown data
- A lot of techniques/attacks are killed on win8 ☹️
- Some types of Pool Corruptions are hard/impossible to exploit ☹️

# The Problem

- All these techniques have prerequisites
- What if there is no chance to fulfill prerequisites?
- Separate Pool Corruptions:
  - Sweet – satisfy exploitable conditions
  - Hardcore – don't satisfy exploitable conditions



# The Problem: examples

- No chance to build correct pool header
  - Memset(mem, 0, count)
  - Memset(mem, CONST, count)
  - Memcpy(mem, uncontrolled\_mem, count)

# DKOHM

- Direct Kernel Object Header Manipulation!

# DKOHM

- Don't attack Pool Allocator mechanisms
- Attack Something Else
- Kernel Objects!
- Objects have header
- Also DKOM which is known in rootkit world

# Object Header (WRK)

- typedef struct \_OBJECT\_HEADER {
- LONG\_PTR PointerCount;
- union {
- LONG\_PTR HandleCount;
- PVOID NextToFree;
- };
- OBJECT\_TYPE Type;
- UCHAR NameInfoOffset;
- UCHAR HandleInfoOffset;
- UCHAR QuotaInfoOffset;
- UCHAR Flags;
- union {
- OBJECT\_CREATE\_INFORMATION ObjectCreateInfo;
- PVOID QuotaBlockCharged;
- };
- PSECURITY\_DESCRIPTOR SecurityDescriptor;
- QUAD Body;
- } OBJECT\_HEADER, \*POBJECT\_HEADER;

# Object Header (WRK)

- typedef struct \_OBJECT\_HEADER {
- [..]
- **POBJECT\_TYPE** Type;
- [..]
- union {
- POBJECT\_CREATE\_INFORMATION ObjectCreateInfo;
- PVOID QuotaBlockCharged;
- };
- PSECURITY\_DESCRIPTOR SecurityDescriptor;
- **QUAD** Body;
- } OBJECT\_HEADER, \*POBJECT\_HEADER;

# Object Header (Win8)

- kd> dt nt!\_OBJECT\_HEADER
- +0x000 PointerCount : Int4B
- +0x004 HandleCount : Int4B
- +0x004 NextToFree : Ptr32 Void
- +0x008 Lock : \_EX\_PUSH\_LOCK
- **+0x00c TypeIndex : UChar**
- +0x00d TraceFlags : UChar
- +0x00d DbgRefTrace : Pos 0, 1 Bit
- +0x00d DbgTracePermanent : Pos 1, 1 Bit
- +0x00e InfoMask : UChar
- +0x00f Flags : UChar
- +0x010 ObjectCreateInfo : Ptr32 \_OBJECT\_CREATE\_INFORMATION
- +0x010 QuotaBlockCharged : Ptr32 Void
- +0x014 SecurityDescriptor : Ptr32 Void
- +0x018 Body : \_QUAD

# Object Type (WRK)

- typedef struct \_OBJECT\_TYPE {
- ERESOURCE Mutex;
- LIST\_ENTRY TypeList;
- UNICODE\_STRING Name;
- PVOID DefaultObject;
- ULONG Index;
- ULONG TotalNumberOfObjects;
- ULONG TotalNumberOfHandles;
- ULONG HighWaterNumberOfObjects;
- ULONG HighWaterNumberOfHandles;
- **OBJECT\_TYPE\_INITIALIZER TypeInfo;**
- #ifdef POOL\_TAGGING
- ULONG Key;
- #endif //POOL\_TAGGING
- ERESOURCE ObjectLocks[ OBJECT\_LOCK\_COUNT ];
- } OBJECT\_TYPE, \*POBJECT\_TYPE;

# Object Type (win8)

- kd> dt nt!\_OBJECT\_TYPE
- +0x000 TypeList : \_LIST\_ENTRY
- +0x008 Name : \_UNICODE\_STRING
- +0x010 DefaultObject : Ptr32 Void
- +0x014 Index : UChar
- +0x018 TotalNumberOfObjects : Uint4B
- +0x01c TotalNumberOfHandles : Uint4B
- +0x020 HighWaterNumberOfObjects : Uint4B
- +0x024 HighWaterNumberOfHandles : Uint4B
- **+0x028 TypeInfo : \_OBJECT\_TYPE\_INITIALIZER**
- +0x080 TypeLock : \_EX\_PUSH\_LOCK
- +0x084 Key : Uint4B
- +0x088 CallbackList : \_LIST\_ENTRY



# Procedures (WRK)

- typedef struct \_OBJECT\_TYPE\_INITIALIZER {
- [..]
- **OB\_DUMP\_METHOD DumpProcedure;**
- **OB\_OPEN\_METHOD OpenProcedure;**
- **OB\_CLOSE\_METHOD CloseProcedure;**
- **OB\_DELETE\_METHOD DeleteProcedure;**
- **OB\_PARSE\_METHOD ParseProcedure;**
- **OB\_SECURITY\_METHOD SecurityProcedure;**
- **OB\_QUERYNAME\_METHOD QueryNameProcedure;**
- **OB\_OKAYTOCLOSE\_METHOD OkayToCloseProcedure;**
- } OBJECT\_TYPE\_INITIALIZER, \*POBJECT\_TYPE\_INITIALIZER;

# Procedures (win8)

- kd> dt nt!\_OBJECT\_TYPE\_INITIALIZER
- [..]
- **+0x030 DumpProcedure : Ptr32 void**
- **+0x034 OpenProcedure : Ptr32 long**
- **+0x038 CloseProcedure : Ptr32 void**
- **+0x03c DeleteProcedure : Ptr32 void**
- **+0x040 ParseProcedure : Ptr32 long**
- **+0x044 SecurityProcedure : Ptr32 long**
- **+0x048 QueryNameProcedure : Ptr32 long**
- **+0x04c OkayToCloseProcedure : Ptr32 unsigned char**

# Procedures (example)

- kd> dt nt!\_OBJECT\_TYPE\_INITIALIZER 849670c0
- +0x000 Length : 0x58
- +0x002 ObjectTypeFlags : 0x10 "
- +0x002 MaintainHandleCount : 0y1
- +0x024 PoolType : 200 ( NonPagedPoolNx )
- +0x02c DefaultNonPagedPoolCharge : 0x154
- **+0x030 DumpProcedure : (null)**
- **+0x034 OpenProcedure : 0x81b8f5df long nt!AlpcpOpenPort+0**
- **+0x038 CloseProcedure : 0x81add15f void nt!AlpcpClosePort+0**
- **+0x03c DeleteProcedure : 0x81adcdf3 void nt!AlpcpDeletePort+0**
- **+0x040 ParseProcedure : (null)**
- **+0x044 SecurityProcedure : 0x81b183c3 long nt!SeDefaultObjectMethod+0**

# Object Type Index Table (x86)

Memory	
Virtual:	nt!ObTypeIndexTable
81251dc0	00000000
81251dc4	bad0b0b0
81251dc8	84162308
81251dcc	841a7f70
81251dd0	8415ce30
81251dd4	8416d130
81251dd8	84160040
81251ddc	8419f378
81251de0	84171cc0
81251de4	84172520

# Object Type Index Table (x64)

Memory		
Virtual:	nt!ObTypeIndexTable	
fffff801`fda9ede0	0000000000000000	
fffff801`fda9ede8	00000000bad0b0b0	
fffff801`fda9edf0	fffffa800cc8d920	
fffff801`fda9edf8	fffffa800cca9c60	
fffff801`fda9ee00	fffffa800cca0d20	
fffff801`fda9ee08	fffffa800ccb3ea0	
fffff801`fda9ee10	fffffa800cc7d100	
fffff801`fda9ee18	fffffa800ccbfb20	
fffff801`fda9ee20	fffffa800ccbbee0	
fffff801`fda9ee28	fffffa800cc68f20	
fffff801`fda9ee30	fffffa800cc78ea0	
fffff801`fda9ee38	fffffa800cc6a080	
fffff801`fda9ee40	fffffa800cc81760	
fffff801`fda9ee48	fffffa800ccae550	
fffff801`fda9ee50	fffffa800cc87790	
fffff801`fda9ee58	fffffa800cc77080	
fffff801`fda9ee60	fffffa800cca5ea0	
fffff801`fda9ee68	fffffa800ccafc00	

# DKOHM Attack

- Smash object header
- Call magic syscall
- Magic syscall triggers dereference of smashed pointer
- It leads to hijack of control flow

# DKOHM Steps

- Spray Pool with Objects
- Fragment Pool (make holes at the **bottom** of page)
- Trigger Overflow/Corruption
- Call magic syscall
- EIP/RIP is under control, game over

# DKOHM


- There are some magic syscalls
- They trigger dereference of object type procedures
- But there is one unique magic syscall ;)



# NtQuerySecurityObject

- Is Not so Secure! :D

```
call    _ObReferenceObjectByHandle@  
test    eax, eax  
js      short loc_637C7F
```



```
mov     edi, [ebp+Object]  
movzx   eax, byte ptr [edi-0Ch] ; eax is under control  
mov     ecx, _ObTypeIndexTable[eax*4] ; ecx is under control  
mov     edx, [ecx+6Ch]  
push    dword ptr [ebp+AccessMode]  
lea     eax, [ecx+34h]  
push    eax  
push    dword ptr [ecx+4Ch]  
lea     eax, [edi-4]  
push    eax  
lea     eax, [ebp+Length]  
push    eax  
push    [ebp+Address]  
lea     eax, [ebp+RequestedInformation]  
push    eax  
xor     esi, esi  
inc     esi  
push    esi  
push    edi  
call    edx ; jump to r0 shellcode/ROP
```

# DKOHM Attacks

- ObTypeIndexTable out of bounds access
- ObTypeIndexTable backdoor/magic entries  
(0x0, 0xBAD0B0B0)
- Object Type Confusion
- DKOM

# Object Type Index Table

- kd> dd nt!ObTypeIndexTable L40
- 81a3edc0 **00000000 bad0b0b0** 8499c040 849aa390
- 81a3edd0 84964f70 8499b4c0 84979500 84999618
- 81a3ede0 84974868 849783c8 8499bf70 84970b40
- 81a3edf0 849a8888 84979340 849aaf70 849a6a38
- 81a3ee00 8496df70 8495b040 8498cf70 84930a50
- 81a3ee10 8495af70 8497ff70 84985040 84999e78
- 81a3ee20 84997f70 8496c040 849646e0 84978f70
- 81a3ee30 8497aec0 84972608 849a0040 849a9750
- 81a3ee40 849586d8 84984f70 8499d578 849ab040
- 81a3ee50 84958938 84974a58 84967168 84967098
- 81a3ee60 8496ddd0 849a5140 8497ce40 849aa138
- 81a3ee70 84a6c058 84969c58 8497e720 85c62a28
- 81a3ee80 85c625f0 **00000000 00000000 00000000**

# ObTypeIndexTable out of bounds

- Uses non-existent object type
- Prerequisite: **only byte** of overflowed data must be in some range
- Triggers Null Pointer Dereference
- Does not work MS13-031 & win8 ☹️

# MS13-031 (LowVaAccess backport)

- Woke up today and ...



# ObTypeIndexTable out of bounds and MS13-031



# ObTypeIndexTable 0xBAD0B0B0 magic

```
pop     esi
push    esi                ; size_t
lea     eax, [ebp+var_1B0]
push    ebx                ; int
push    eax                ; void *
mov     dword_5E5DC4, 0BAD0B0B0h
call    _memset
```

```
.data:005E5DBF                db      ? ;
.data:005E5DC0                ; ULONG_PTR ObTypeIndexTable
.data:005E5DC0                _ObTypeIndexTable dd ?
.data:005E5DC0
.data:005E5DC4                dword_5E5DC4 dd ?
```

# ObTypeIndexTable 0xBAD0B0B0 magic

- Uses magic entry (CIA backdoor from 1994?)
- x86 – spray pool till 0xBAD0B000 Page is allocated
- Double Page Fault technique (Intel only)
- x64 0xBAD0B0B0 is extended by **zeroes!!!** Just alloc fake Object Type entry in r3
- **SMAP** will eliminate this technique ☹️ (x64)



# ObTypeIndexTable 0xBAD0B0B0 magic

- x64:
  - nt!NtQuerySecurityObject+0x89:
  - mov r10,qword ptr [rdx+98h]  
ds:002b:00000000`bad0b148 userland!!!
- x86:
  - nt!NtQuerySecurityObject+0x80:
  - mov edx,dword ptr [ecx+6Ch]  
ds:0023:bad0b11c (Paged Pool spray)

# Object Type Confusion

- kd> dt nt!\_OBJECT\_TYPE\_INITIALIZER 849a9778
  - +0x044 SecurityProcedure : 0x81b6b085 long  
**nt!IoGetSetSecurityObject**
- kd> dt nt!\_OBJECT\_TYPE\_INITIALIZER 84967190
  - +0x044 SecurityProcedure : 0x81b6b4c0 long  
**nt!CmpSecurityMethod**
- kd> dt nt!\_OBJECT\_TYPE\_INITIALIZER 849aa3b8
  - +0x044 SecurityProcedure : 0x81b183c3 long  
**nt!SeDefaultObjectMethod**

# Object Type Confusion / DKOM

- Change Type/Data of Kernel Object
- Redirect execution flow with fake object type/data
- Achieve write4 primitive or hijack of execution flow
- Prerequisite: precise control over overflowed data ☹️

# Conclusions

- 2013, but still generic techniques DO exist
- Windows kernel does not protect Object Manager / kernel object headers at all
- MS should implement cookie in object header
- SMAP(Windows Blue?) will eliminate some techniques ☹️ (0xBADB0B0 on x64)
- Anyway, we will be pwning Windows Kernel Pool Corruptions

# Q&A

- Dude! Where is 0day drop???
- @NTarakanov

# References

- SoBelt X'con 2005
- Kostya Kortchinsky SyScan 2008
- Tarjei Mandt BH DC 2011
- Tarjei Mandt BH US 2012
- Zhenhua 'Eric' Liu BH EU 2013
- Must read: j00ru's work on windows kernel objects