Exploiting CVE-2018-8611

Windows Kernel Transaction Manager (KTM) Race Condition

Cedric Halbronn - OffensiveCon 2020

About

- Exploit Development Group (EDG), NCC Group
- Write exploits to help consultants do their job
- Focus on patched vulnerabilities

Cedric Halbronn

- Presenting
- @saidelike, cedric.halbronn@nccgroup.com

Aaron Adams

- Co-researcher, unable to attend
- @fidgetingbits, aaron.adams@nccgroup.com

This talk

- Discuss an interesting race condition affecting Microsoft Kernel Transaction Manager (KTM)
  - Found used in the wild by Kaspersky
  - Boris Larin (@Oct0xor) and Igor Soumenkov (@2igosha)
- Exploited by us early 2019
  - Never got to see the original exploit or details
- Minimal details from Kaspersky at the time
  - Race condition in KTM
  - Exploitable from inside browser sandbox
  - Works on Windows 10
  - A few hints for triggering the race
- Presented by Aaron at POC2019 in November

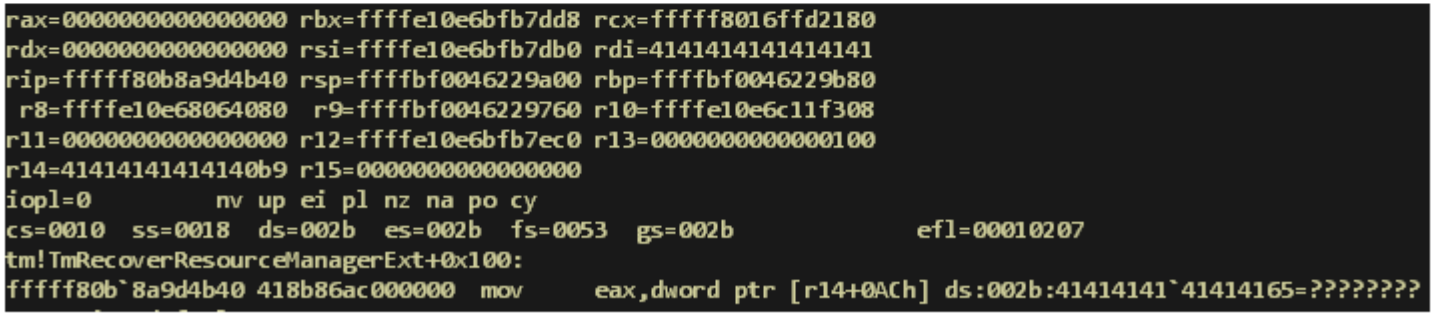
Notable KTM-related security findings

- 2010 - CVE-2010-1889 - Tavis Ormandy - invalid free
- 2015 - MS15-038 - James Forshaw - type confusion
- 2017 - CVE-2017-8481 - j00ru - stack memory disclosure
- 2018 - CVE-2018-8611 - Oct0xor/2igosha - Kaspersky blog
- 2019 - Proton Bot malware uses KTM
  - Used transacted versions of common functions to evade API inspection

Oct0xor/2igosha - Kaspersky blog

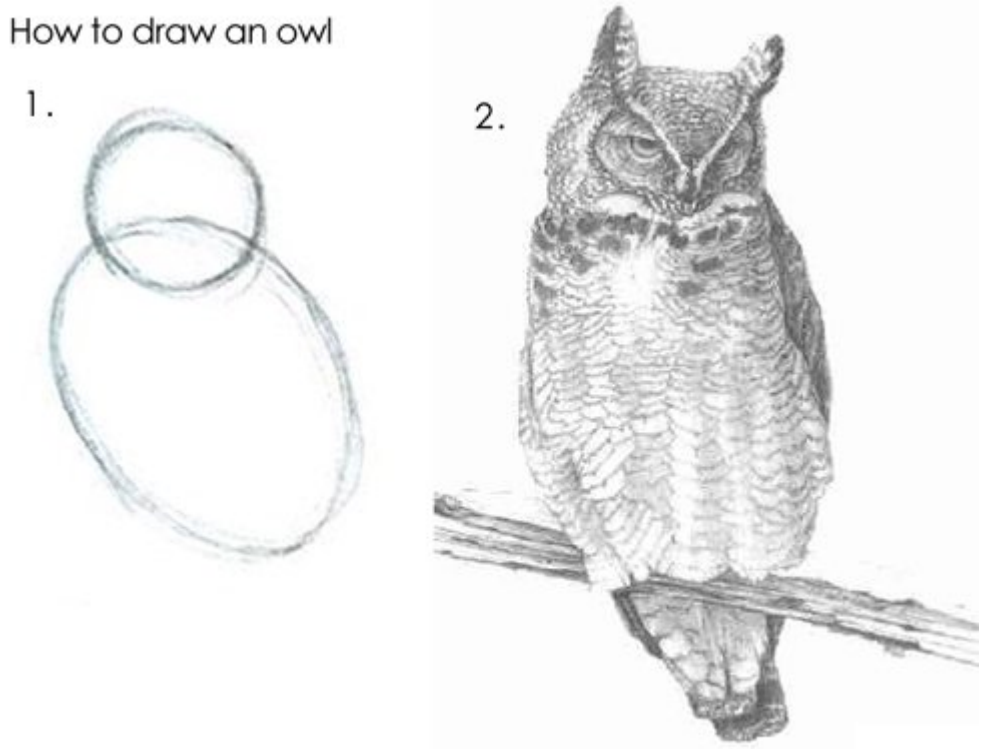
To abuse this vulnerability exploit first creates a named pipe and opens it for read and write. Then it creates a pair of new transaction manager objects, resource manager objects, transaction objects and creates a big number of enlistment objects for what we will call "Transaction #2". Enlistment is a special object that is used for association between a transaction and a resource manager. When the transaction state changes associated resource manager is notified by the KTM. After that it creates one more enlistment object only now it does so for "Transaction #1" and commits all the changes made during this transaction.

After all the initial preparations have been made exploit proceeds to the second part of vulnerability trigger. It creates multiple threads and binds them to a single CPU core. One of created threads calls NtQueryInformationResourceManager in a loop, while second thread tries to execute NtRecoverResourceManager once. But the vulnerability itself is triggered in the third thread. This thread uses a trick of execution NtQueryInformationThread to obtain information on the latest executed syscall for the second thread. Successful execution of NtRecoverResourceManager will mean that race condition has occurred and further execution of WriteFile on previously created named pipe will lead to memory corruption.



Proof of concept: execution of WriteFile with buffer set to 0x41

Me when reading write-up



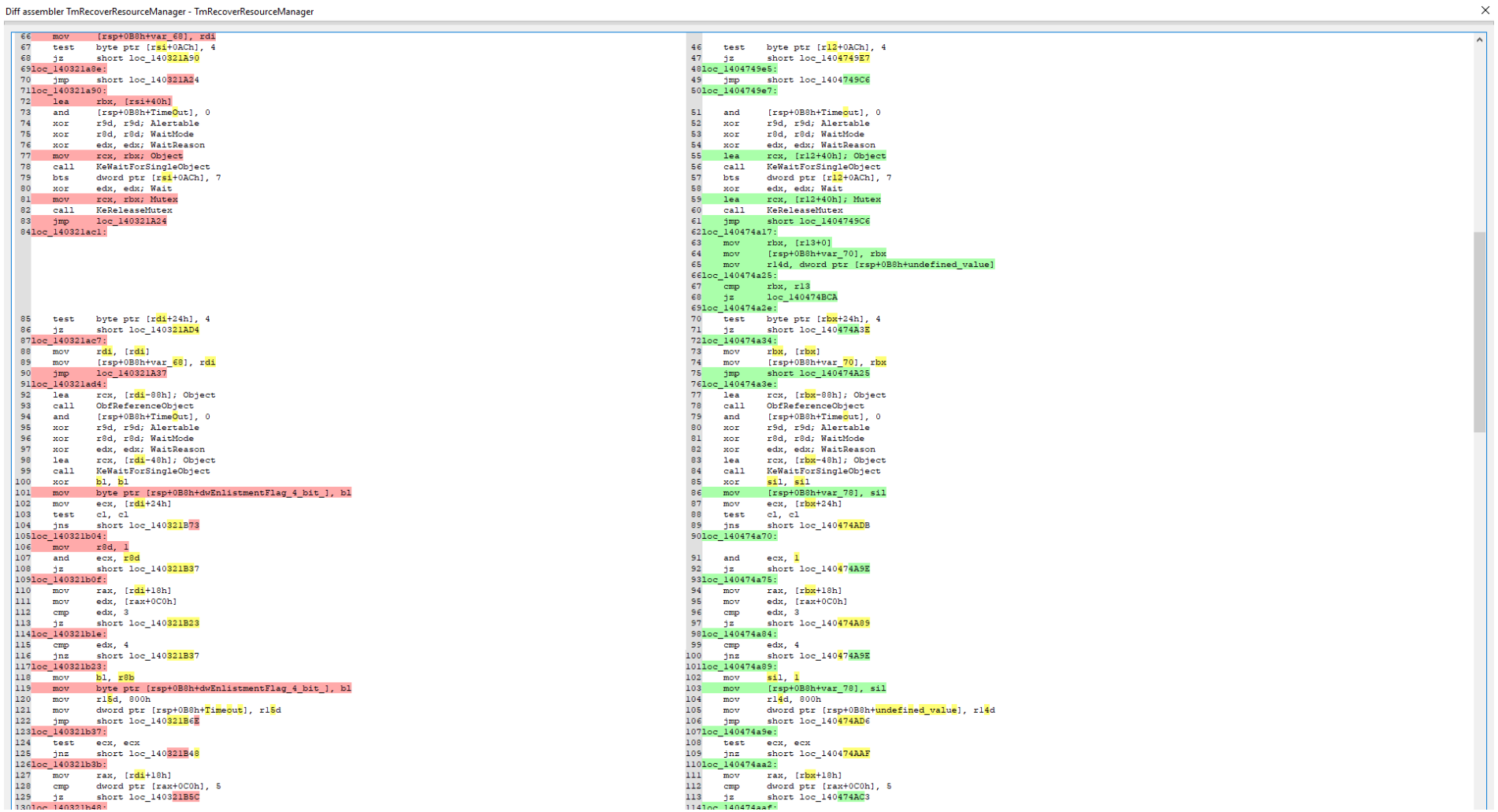
Tooling

- Virtualization: VMWare Workstation
- Binary analysis: IDA Pro, Hex-Rays Decompiler
- Binary diffing: Diaphora
- Collaboration: IDArling
- Debugging: WinDbg (ring0), virtualkd, x64dbg (ring3)
- Additional plugins/tools: ret-sync, HexRaysPyTools
- Structure analysis: Vergilius Project, ReactOS source
- Syscall numbers: windows-syscalls by j00ru
- Slides: Remarkjs

Diffing - functions

Line	Address	Name	Address 2	Name 2	Ratio	BBlocks 1	BBlocks 2	Description
00009	1403dabd0	PfpRepurposeNameLoggingTrace	1403da8a0	PfpRepurposeNameLoggingTrace	0.890	1	1	Perfect match, same name
00019	140561540	PnpWaitForDevicesToStart	140562540	PnpWaitForDevicesToStart	0.880	1	1	Perfect match, same name
00014	140432b40	LpcRequestWaitReplyPort	140432850	LpcRequestWaitReplyPort	0.860	1	1	Perfect match, same name
00003	14033da80	TmCommitComplete	14033d760	TmCommitComplete	0.860	1	1	Perfect match, same name
00002	14033da14	TmPrepareComplete	14033d6f4	TmPrepareComplete	0.860	1	1	Perfect match, same name
00004	14033fe14	TmReadOnlyEnlistment	14033fa84	TmReadOnlyEnlistment	0.770	1	1	Perfect match, same name
00026	140574680	TmpEnlistmentInitialization	140575680	TmpEnlistmentInitialization	0.680	1	1	Perfect match, same name
00025	1405745b0	TmpTransactionManagerInitialization	1405755b0	TmpTransactionManagerInitialization	0.670	1	1	Perfect match, same name
00012	1403ead50	TmpFindTransactionManager	1403eaa20	TmpFindTransactionManager	0.670	1	1	Perfect match, same name
00000	14030b5c4	ObInsertObject	14030b5d4	ObInsertObject	0.670	1	1	Perfect match, same name
00013	14042ebf0	TmRollbackComplete	14042e900	TmRollbackComplete	0.610	1	1	Perfect match, same name
00001	140321998	TmRecoverResourceManager	140474940	TmRecoverResourceManager	0.610	38	39	Perfect match, same name
00028	14050bd20	VerifierExEnterPriorityRegionAndAcquir...	14050cd20	VerifierExEnterCriticalRegionAndAcquir...	0.500	1	1	Nodes, edges, complexity and mnemonics with small diffe

Diffing - assembly




```
66    mov     [rsp+0B8h+var_68], rdi
67    test    byte ptr [rsi+0ACh], 4
68    jz      short loc_140321A90
69loc_140321a8e:
70    jmp     short loc_140321AA4
71loc_140321a90:
72    lea     rbx, [rsi+40h]
73    and     [rsp+0B8h+TimeOut], 0
74    xor     r9d, r9d; Alertable
75    xor     r8d, r8d; WaitMode
76    xor     edx, edx; WaitReason
77    mov     rcx, rbx; Object
78    call    KeWaitForSingleObject
79    bts     dword ptr [rsi+0ACh], 7
80    xor     edx, edx; Wait
81    mov     rcx, rbx; Mutex
82    call    KeReleaseMutex
83    jmp     loc_140321A24
84loc_140321ac1:
85    test    byte ptr [rdi+24h], 4
86    jz      short loc_140321AD4
87loc_140321ac7:
88    mov     rdi, [rdi]
89    mov     [rsp+0B8h+var_68], rdi
90    jmp     loc_140321A37
91loc_140321ad4:
92    lea     rcx, [rdi-88h]; Object
93    call    ObfReferenceObject
94    and     [rsp+0B8h+TimeOut], 0
95    xor     r9d, r9d; Alertable
96    xor     r8d, r8d; WaitMode
97    xor     edx, edx; WaitReason
98    lea     rcx, [rdi-40h]; Object
99    call    KeWaitForSingleObject
100   xor     bl, bl
101   mov     byte ptr [rsp+0B8h+dwEnlistmentFlag_4_bit_], bl
102   mov     ecx, [rdi+24h]
103   test    cl, cl
104   jns     short loc_140321B73
105loc_140321b04:
106   mov     r8d, 1
107   and     ecx, r8d
108   jz      short loc_140321B37
109loc_140321b0f:
110   mov     rax, [rdi+18h]
111   mov     edx, [rax+0C0h]
112   cmp     edx, 3
113   jz      short loc_140321B23
114loc_140321b1e:
115   cmp     edx, 4
116   jnz     short loc_140321B37
117loc_140321b23:
118   mov     bl, r8b
119   mov     byte ptr [rsp+0B8h+dwEnlistmentFlag_4_bit_], bl
120   mov     r15d, 800h
121   mov     dword ptr [rsp+0B8h+Timeout], r15d
122   jmp     short loc_140321B6E
123loc_140321b37:
124   test    ecx, ecx
125   jnz     short loc_140321B48
126loc_140321b3b:
127   mov     rax, [rdi+18h]
128   cmp     dword ptr [rax+0C0h], 5
129   jz      short loc_140321B5C
130loc_140321b48:
```

```
46    test    byte ptr [r12+0ACh], 4
47    jz      short loc_1404749E7
48loc_1404749e5:
49    jmp     short loc_1404749C6
50loc_1404749e7:
51    and     [rsp+0B8h+Timeout], 0
52    xor     r9d, r9d; Alertable
53    xor     r8d, r8d; WaitMode
54    xor     edx, edx; WaitReason
55    lea     rcx, [r12+40h]; Object
56    call    KeWaitForSingleObject
57    bts     dword ptr [r12+0ACh], 7
58    xor     edx, edx; Wait
59    lea     rcx, [r12+40h]; Mutex
60    call    KeReleaseMutex
61    jmp     short loc_1404749C6
62loc_140474a17:
63    mov     rbx, [r13+0]
64    mov     [rsp+0B8h+var_70], rbx
65    mov     r14d, dword ptr [rsp+0B8h+undefined_value]
66loc_140474a25:
67    cmp     rbx, r13
68    jz      loc_140474BCA
69loc_140474a2e:
70    test    byte ptr [rbx+24h], 4
71    jz      short loc_140474A3E
72loc_140474a34:
73    mov     rbx, [rbx]
74    mov     [rsp+0B8h+var_70], rbx
75    jmp     short loc_140474A25
76loc_140474a3e:
77    lea     rcx, [rbx-88h]; Object
78    call    ObfReferenceObject
79    and     [rsp+0B8h+Timeout], 0
80    xor     r9d, r9d; Alertable
81    xor     r8d, r8d; WaitMode
82    xor     edx, edx; WaitReason
83    lea     rcx, [rbx-40h]; Object
84    call    KeWaitForSingleObject
85    xor     sil, sil
86    mov     [rsp+0B8h+var_78], sil
87    mov     ecx, [rbx+24h]
88    test    cl, cl
89    jns     short loc_140474ADB
90loc_140474a70:
91    and     ecx, 1
92    jz      short loc_140474A9E
93loc_140474a75:
94    mov     rax, [rbx+18h]
95    mov     edx, [rax+0C0h]
96    cmp     edx, 3
97    jz      short loc_140474A89
98loc_140474a84:
99    cmp     edx, 4
100   jnz     short loc_140474A9E
101loc_140474a89:
102   mov     sil, 1
103   mov     [rsp+0B8h+var_78], sil
104   mov     r14d, 800h
105   mov     dword ptr [rsp+0B8h+undefined_value], r14d
106   jmp     short loc_140474AD6
107loc_140474a9e:
108   test    ecx, ecx
109   jnz     short loc_140474AAF
110loc_140474aa2:
111   mov     rax, [rbx+18h]
112   cmp     dword ptr [rax+0C0h], 5
113   jz      short loc_140474AC3
114loc_140474aaf:
```

Diffing - Hex-Rays pre-cleanup

```
91              v17);
92        v15 = v18;
93        if ( *(_BYTE *)(v9 + 172) & 4 )
94          v15 = 1;
95        v18 = v15;
96        ObfDereferenceObject(v7 - 17);
97        KeWaitForSingleObject((char *)v1 + 40, Executive, 0, 0, 0i64);
98        if ( *((_DWORD *)v1 + 7) != 2 )
99          goto LABEL_34;
100       v2 = v18;
101     }
102     else
103     {
104       ObfDereferenceObject(v7 - 17);
105     }
106     if ( v2 )
107     {
108       v7 = (_QWORD *)*((_QWORD *)v1 + 34);
109       v2 = 0;
110       v18 = 0;
111     }
112     else
113     {
114LABEL_12:
```

```
88              v16);
89        ObfDereferenceObject(v6 - 17);
90        KeWaitForSingleObject((char *)v1 + 40, Executive, 0, 0, 0i64);
91        if ( *((_DWORD *)v1 + 7) != 2 )
92          goto LABEL_32;
93        v14 = *((_QWORD *)v1 + 45);
94        if ( !v14 || *(_DWORD *)(v14 + 64) != 3 )
95          goto LABEL_31;
96        v6 = (_QWORD *)*((_QWORD *)v1 + 34);
97      }
98      else
99      {
100       ObfDereferenceObject(v6 - 17);
101LABEL_12:
```

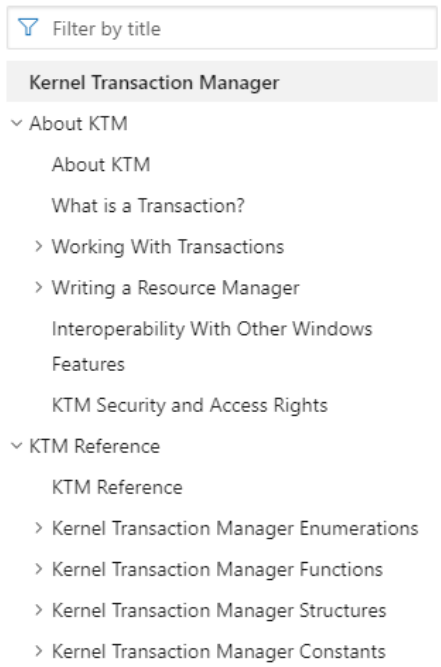
Agenda

- What is KTM?
- Patch analysis
- Triggering the bug
- Finding a write primitive
- Building a read primitive
- Privilege escalation
- Recent bonus info

Windows Kernel Transaction Manager (KTM)

KTM - What is it?

- MSDN documentation
  - KTM Portal



Kernel Transaction Manager

05/31/2018 • 2 minutes to read •

Purpose

The Kernel Transaction Manager (KTM) enables the development of applications that use transactions. The transaction engine itself is within the kernel, but transactions can be developed for kernel- or user-mode transactions, and within a single host or among distributed hosts.

The KTM is used to implement Transactional NTFS (TxF) and Transactional Registry (TxR). TxF allows transacted file system operations within the NTFS file system. TxR allows transacted registry operations. KTM enables client applications to coordinate file system and registry operations with a transaction.

To develop an application that coordinates transactions with resources other than TxF or TxR, you must first develop a Win32 transaction-aware service, also called a resource manager.

Managed and COM+ applications should use their native transaction managers.

Where applicable

KTM can be used with applications and resource managers hosted on Windows Vista or Windows Server 2008.

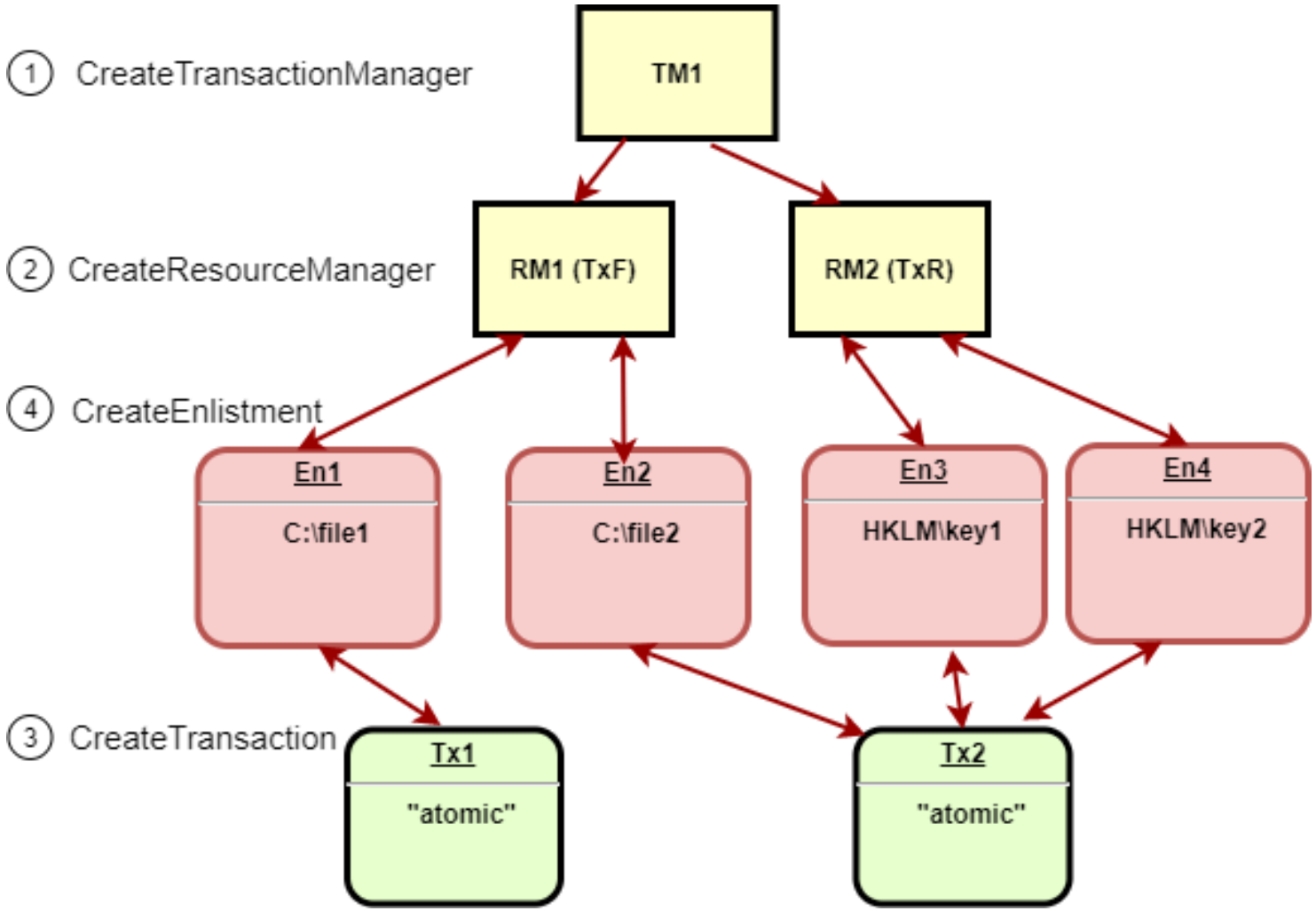
KTM - What is it?

- Kernel service added in Windows Vista (~2006)
  - Windows 7 and earlier: `ntoskrnl.exe`
  - Windows 8 and later: `tm.sys`
- Provide "ACID" functionality: atomic, consistent, isolated, and durable
- KTM service used by two major Windows components
  - Transactional Registry
  - Transactional NTFS
- A few dozen APIs/system calls exposed to userland

Important objects

- KTM service has **4** fundamental kernel objects
  - All referenced counted objects created by `ObCreateObject()`
- **Transaction Manager (TM)**
  - Manages a log of transactions associated with one or more resource managers
- **Resource Manager (RM)**
  - Manages enlistments related to a specific managed resource doing work for a Transaction
- **Transaction (Tx)**
  - Tracks a series of sub actions making up a single atomic operation
- **Enlistment (En)**
  - Some code responsible for doing work related to a Transaction

Important objects



Transaction Manager (TM)

- Created using `CreateTransactionManager()`
  - Usually first to exist

```
HANDLE CreateTransactionManager(
  IN LPSECURITY_ATTRIBUTES lpTransactionAttributes,
  LPWSTR                   LogFileName,
  IN ULONG                 CreateOptions,
  IN ULONG                 CommitStrength
);
```

- Allocates a `_KTM` structure on the non-paged pool
  - `TmTm` pool tag
- A resource manager must be associated with some TM
- Optional log for transactions
  - A **volatile** TM is one that uses no log file
  - Set `TRANSACTION_MANAGER_VOLATILE` flag in `CreateOptions` parameter
  - Logs have limited size - problematic for exploitation

_KTM

- Most fields omitted

```
//0x3c0 bytes (sizeof)
struct _KTM
{
    ULONG cookie;                                                           //0x0
    struct _KMUTANT Mutex;                                                  //0x8
    enum KTM_STATE State;                                                   //0x40
    [...]
    ULONG Flags;                                                            //0x80
    [...]
    struct _KRESOURCEMANAGER* TmRm;                                         //0x2a8
    [...]
};
```

Resource Manager (RM)

- Created using `CreateResourceManager()`

```
HANDLE CreateResourceManager(
  IN LPSECURITY_ATTRIBUTES lpResourceManagerAttributes,
  IN LPGUID                ResourceManagerId,
  IN DWORD                 CreateOptions,
  IN HANDLE                TmHandle,
  LPWSTR                   Description
);
```

- Must be passed a TM handle
- Optional `Description` parameter
- Allocates a `_KRESOURCEMANAGER` structure on the non-paged pool
  - `TmRm` pool tag

_KRESOURCEMANAGER

```
//0x250 bytes (sizeof)
struct _KRESOURCEMANAGER
{
    struct _KEVENT NotificationAvailable;                                   //0x0
    ULONG cookie;                                                           //0x18
    enum _KRESOURCEMANAGER_STATE State;                                     //0x1c
    ULONG Flags;                                                            //0x20
    struct _KMUTANT Mutex;                                                  //0x28
    [...]
    struct _KQUEUE NotificationQueue;                                       //0x98
    struct _KMUTANT NotificationMutex;                                      //0xd8
    struct _LIST_ENTRY EnlistmentHead;                                      //0x110
    ULONG EnlistmentCount;                                                  //0x120
    LONG (*NotificationRoutine)(struct _KENLISTMENT* arg1, VOID* arg2, VOID* arg3,
                                ULONG arg4, union _LARGE_INTEGER* arg5, ULONG arg6, VOID* arg7); //0x128
    [...]
    struct _KTM* Tm;                                                        //0x168
    struct _UNICODE_STRING Description;                                     //0x170
    [...]
};
```

_KRESOURCEMANAGER fields

- `Tm` - Pointer to the associated transaction manager
- `Description` - Unicode description of resource manager
- `Mutex` - Locks RM. Other code cannot
  - Parse the resource manager's enlistments list
  - Read `Description`
  - etc.
- `EnlistmentHead` - List of associated enlistments with resource manager
- `NotificationQueue` - Notification events
  - Queried from ring3 to read enlistment state change events

Transaction (Tx)

- Created using `CreateTransaction()` function

```
HANDLE CreateTransaction(
  IN LPSECURITY_ATTRIBUTES lpTransactionAttributes,
  IN LPGUID                UOW,
  IN DWORD                 CreateOptions,
  IN DWORD                 IsolationLevel,
  IN DWORD                 IsolationFlags,
  IN DWORD                 Timeout,
  LPWSTR                   Description
);
```

- Creates a `_KTRANSACTION` structure on the non-paged pool using
  - `TmTx` pool tag
- Represents whole piece of work to be done
- Resource managers enlist in this transaction to complete the work

_KTRANSACTION

```
//0x2d8 bytes (sizeof)
struct _KTRANSACTION
{
    struct _KEVENT OutcomeEvent;                                    //0x0
    ULONG cookie;                                                   //0x18
    struct _KMUTANT Mutex;                                          //0x20
    [...]
    struct _GUID UOW;                                               //0xb0
    enum _KTRANSACTION_STATE State;                                 //0xc0
    ULONG Flags;                                                    //0xc4
    struct _LIST_ENTRY EnlistmentHead;                              //0xc8
    ULONG EnlistmentCount;                                          //0xd8
    [...]
    union _LARGE_INTEGER Timeout;                                   //0x128
    struct _UNICODE_STRING Description;                             //0x130
    [...]
    struct _KTM* Tm;                                                //0x200
    [...]
};
```

Enlistments (En)

- Created using `CreateEnlistment()`

```
hEn = CreateEnlistment(
    NULL,   // lpEnlistmentAttributes
    hRM,    // ResourceManagerHandle - Existing resource manager handle
    hTx,    // TransactionHandle - Existing transaction handle
    0x39ffff0f, // NotificationMask - Special value to receive all possible notifications
    0,        // CreateOptions
    NULL    // EnlistmentKey
);
```

- Allocates a `_KENLISTMENT` structure on the non-paged pool
  - `TmEn` pool tag
- Each has an assigned `GUID`
- Must be associated with both a resource manager and a transaction
- Typically a transaction will have multiple enlistments

_KENLISTMENT

```
//0x1e0 bytes (sizeof)
struct _KENLISTMENT
{
    ULONG cookie;                                                           //0x0
    struct _KTMOBJECT_NAMESPACE_LINK NamespaceLink;                         //0x8
    struct _GUID EnlistmentId;                                              //0x30
    struct _KMUTANT Mutex;                                                  //0x40
    struct _LIST_ENTRY NextSameTx;                                          //0x78
    struct _LIST_ENTRY NextSameRm;                                          //0x88
    struct _KRESOURCEMANAGER* ResourceManager;                              //0x98
    struct _KTRANSACTION* Transaction;                                      //0xa0
    enum _KENLISTMENT_STATE State;                                          //0xa8
    ULONG Flags;                                                            //0xac
    ULONG NotificationMask;                                                 //0xb0
    [...]
};
```

_KENLISTMENT fields of interest

- `Transaction` - The transaction that the enlistment is actually doing work for
- `Flags` - Indicates the type and state of the enlistment
- `Mutex` - Locks the enlistment and prevents other code from manipulating it
- `State` - The current state of the enlistment in relation to the transaction
- `NotificationMask` - Which notifications should be queued to the resource manager related to this enlistment
- `NextSameRm` - A linked list of enlistments associated with the same resource manager
  - This is the list entry whose head is `_KRESOURCEMANAGER.EnlistmentHead`

_KENLISTMENT flags

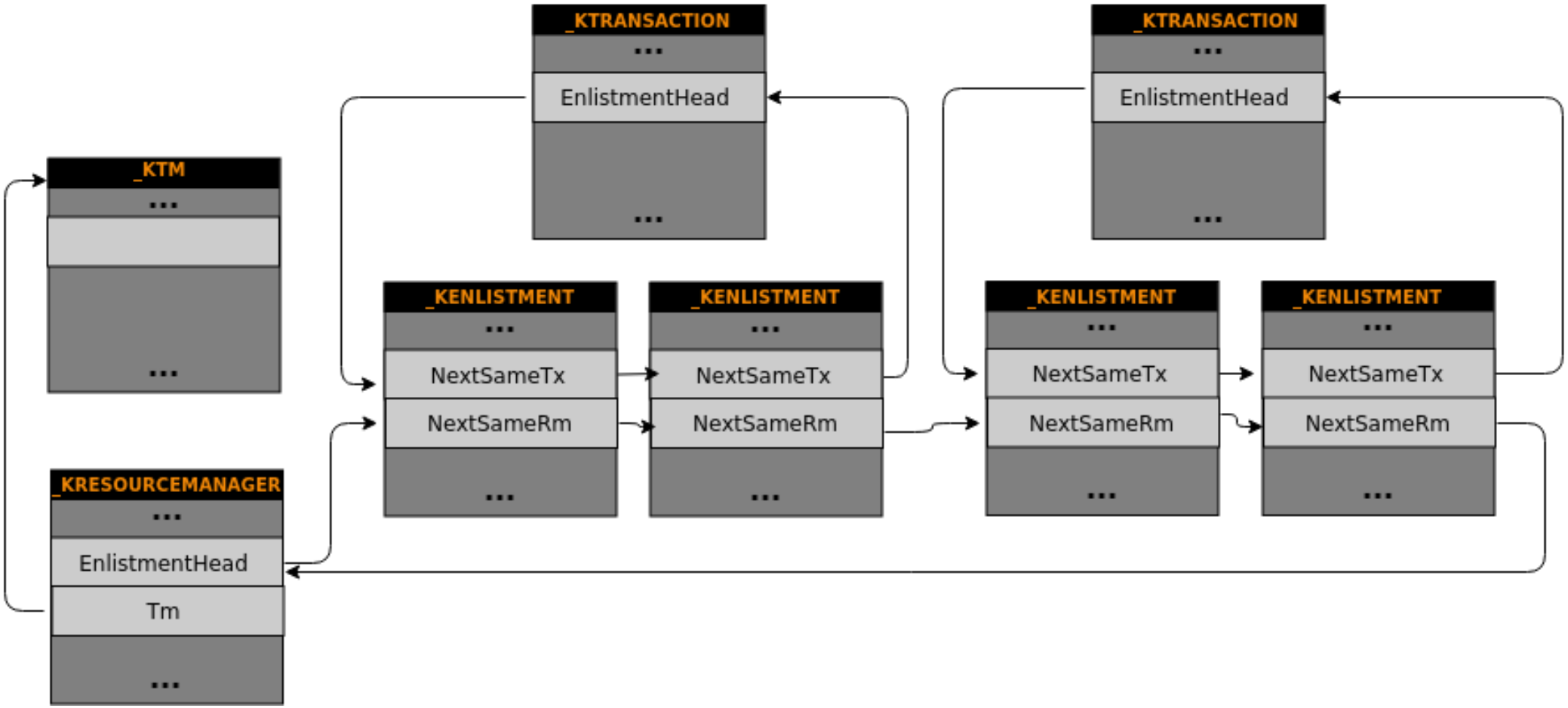
- The `Flags` field uses undocumented flags

```
enum KENLISTMENT_FLAGS {
    KENLISTMENT_SUPERIOR           = 0x01,
    KENLISTMENT_RECOVERABLE        = 0x02,
    KENLISTMENT_FINALIZED          = 0x04,
    KENLISTMENT_FINAL_NOTIFICATION = 0x08,
    KENLISTMENT_OUTCOME_REQUIRED   = 0x10,
    KENLISTMENT_HAS_SUPERIOR_SUB   = 0x20,
    KENLISTMENT_IS_NOTIFIABLE      = 0x80,
    KENLISTMENT_DELETED            = 0x80000000
};
```

How to finalize and free an enlistment?

- Enlistments are a reference counted object
- Call some code path that triggers `TmpFinalizeEnlistment()` to lower ref count
  - A `Prepared` enlistment upon moving to `Committed` state will be finalized
  - Use `CommitComplete()` function on enlistment handle
- Then `CloseHandle()` to remove our final userland reference
- Either frees immediately, or upon any other KTM kernel code doing final dereference

Structure relationship overview



Transaction and Enlistment States

- Transaction not complete until all enlistments have committed
- Transaction cannot be committed until all of enlistments transition through a series of synchronized states
- A transaction with only one enlistment is the exception
- Typical state transitions

`PrePreparing` -> `PrePrepared` -> `Preparing` -> `Prepared` -> `Committed`

_KENLISTMENT_STATE

Documented

```
enum _KENLISTMENT_STATE
{
    //...
    KEnlistmentPreparing = 257,
    KEnlistmentPrepared = 258,
    KEnlistmentCommitted = 260,
    //...
    KEnlistmentPreparing = 257,
    //...
    KEnlistmentPrePreparing = 266,
    //...
    KEnlistmentPrePrepared = 273,
};
```

Notifications

- Dictated by enlistment `NotificationMask` option at creation
- Each RM has a set of associated Tx notifications that occur on milestone events, such as an En switching from one state to another
- Notifications can be read using `GetNotificationResourceManager()`
- The events are queued/retrieved using FIFO

```
BOOL GetNotificationResourceManager(
IN HANDLE                       ResourceManagerHandle,
OUT PTRANSACTION_NOTIFICATION TransactionNotification,
IN ULONG                        NotificationLength,
IN DWORD                        dwMilliseconds,
OUT PULONG                      ReturnLength
);
```

- `TRANSACTION_NOTIFICATION` struct contains a `TRANSACTION_NOTIFICATION_RECOVERY_ARGUMENT`
  - Tells us which En a notification is associated with

Recovery

- If a Tx fails or is interrupted for whatever reason, it can be possible to recover
- Recovery in part possible by calling `RecoverResourceManager()`

```
BOOL RecoverResourceManager(
IN HANDLE ResourceManagerHandle
);
```

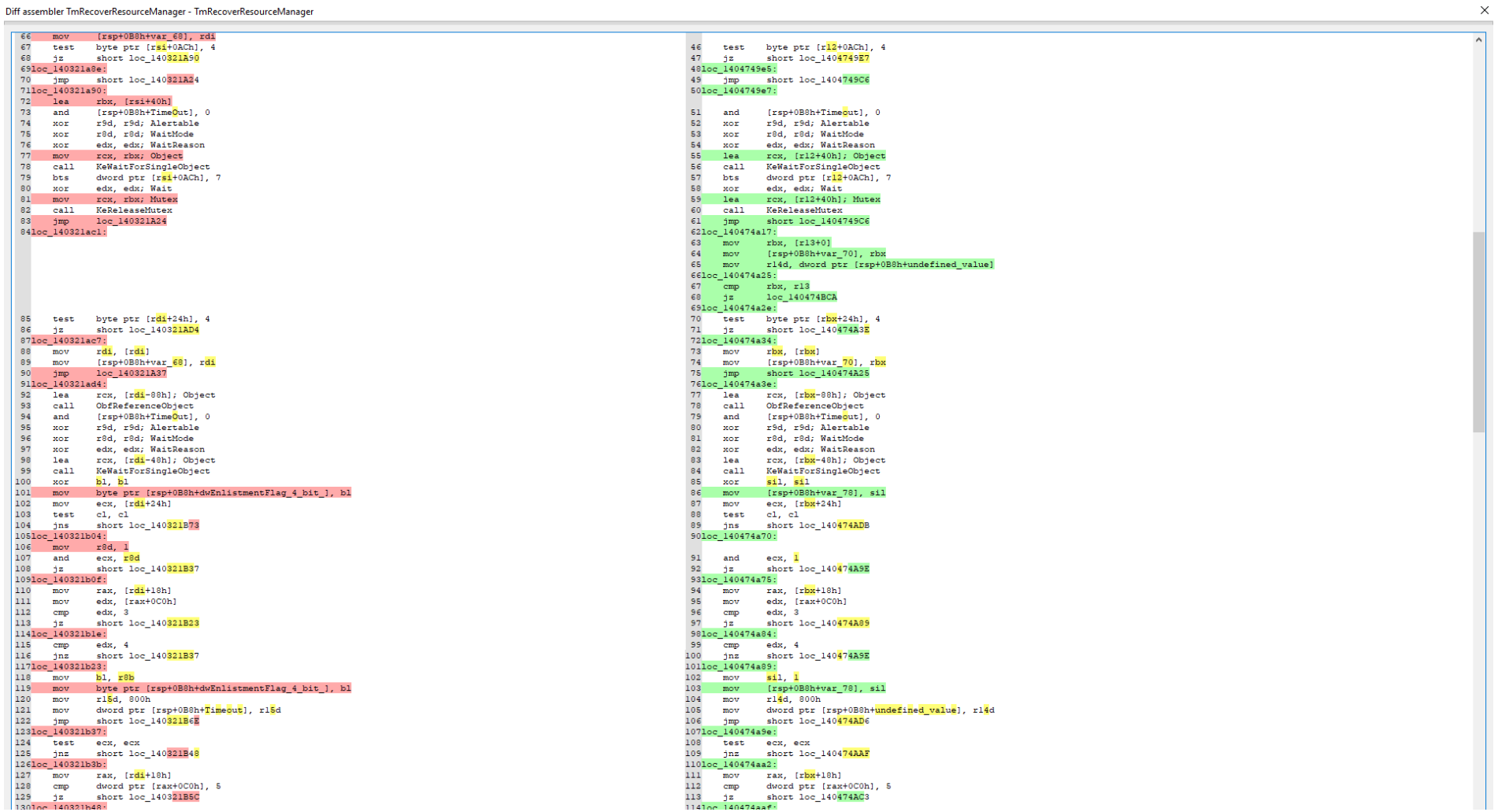
- During this recovery phase, each enlistment associated with transactions in specific states will receive a notification
- Allows the enlisted workers to synchronize on what they were doing for the transaction

Understanding CVE-2018-8611

Diffing - functions

Line	Address	Name	Address 2	Name 2	Ratio	BBlocks 1	BBlocks 2	Description
00009	1403dabd0	PfpRepurposeNameLoggingTrace	1403da8a0	PfpRepurposeNameLoggingTrace	0.890	1	1	Perfect match, same name
00019	140561540	PnpWaitForDevicesToStart	140562540	PnpWaitForDevicesToStart	0.880	1	1	Perfect match, same name
00014	140432b40	LpcRequestWaitReplyPort	140432850	LpcRequestWaitReplyPort	0.860	1	1	Perfect match, same name
00003	14033da80	TmCommitComplete	14033d760	TmCommitComplete	0.860	1	1	Perfect match, same name
00002	14033da14	TmPrepareComplete	14033d6f4	TmPrepareComplete	0.860	1	1	Perfect match, same name
00004	14033fe14	TmReadOnlyEnlistment	14033fa84	TmReadOnlyEnlistment	0.770	1	1	Perfect match, same name
00026	140574680	TmpEnlistmentInitialization	140575680	TmpEnlistmentInitialization	0.680	1	1	Perfect match, same name
00025	1405745b0	TmpTransactionManagerInitialization	1405755b0	TmpTransactionManagerInitialization	0.670	1	1	Perfect match, same name
00012	1403ead50	TmpFindTransactionManager	1403eaa20	TmpFindTransactionManager	0.670	1	1	Perfect match, same name
00000	14030b5c4	ObInsertObject	14030b5d4	ObInsertObject	0.670	1	1	Perfect match, same name
00013	14042ebf0	TmRollbackComplete	14042e900	TmRollbackComplete	0.610	1	1	Perfect match, same name
00001	140321998	TmRecoverResourceManager	140474940	TmRecoverResourceManager	0.610	38	39	Perfect match, same name
00028	14050bd20	VerifierExEnterPriorityRegionAndAcquir...	14050cd20	VerifierExEnterCriticalRegionAndAcquir...	0.500	1	1	Nodes, edges, complexity and mnemonics with small diffe

Diffing - assembly



Diffing - Hex-Rays pre-cleanup

```
91              v17);
92        v15 = v18;
93        if ( *(_BYTE *)(v9 + 172) & 4 )
94          v15 = 1;
95        v18 = v15;
96        ObfDereferenceObject(v7 - 17);
97        KeWaitForSingleObject((char *)v1 + 40, Executive, 0, 0, 0i64);
98        if ( *((_DWORD *)v1 + 7) != 2 )
99          goto LABEL_34;
100       v2 = v18;
101     }
102     else
103     {
104       ObfDereferenceObject(v7 - 17);
105     }
106     if ( v2 )
107     {
108       v7 = (_QWORD *)*((_QWORD *)v1 + 34);
109       v2 = 0;
110       v18 = 0;
111     }
112     else
113     {
114LABEL_12:
```

```
88              v16);
89        ObfDereferenceObject(v6 - 17);
90        KeWaitForSingleObject((char *)v1 + 40, Executive, 0, 0, 0i64);
91        if ( *((_DWORD *)v1 + 7) != 2 )
92          goto LABEL_32;
93        v14 = *((_QWORD *)v1 + 45);
94        if ( !v14 || *(_DWORD *)(v14 + 64) != 3 )
95          goto LABEL_31;
96        v6 = (_QWORD *)*((_QWORD *)v1 + 34);
97      }
98      else
99      {
100       ObfDereferenceObject(v6 - 17);
101LABEL_12:
```

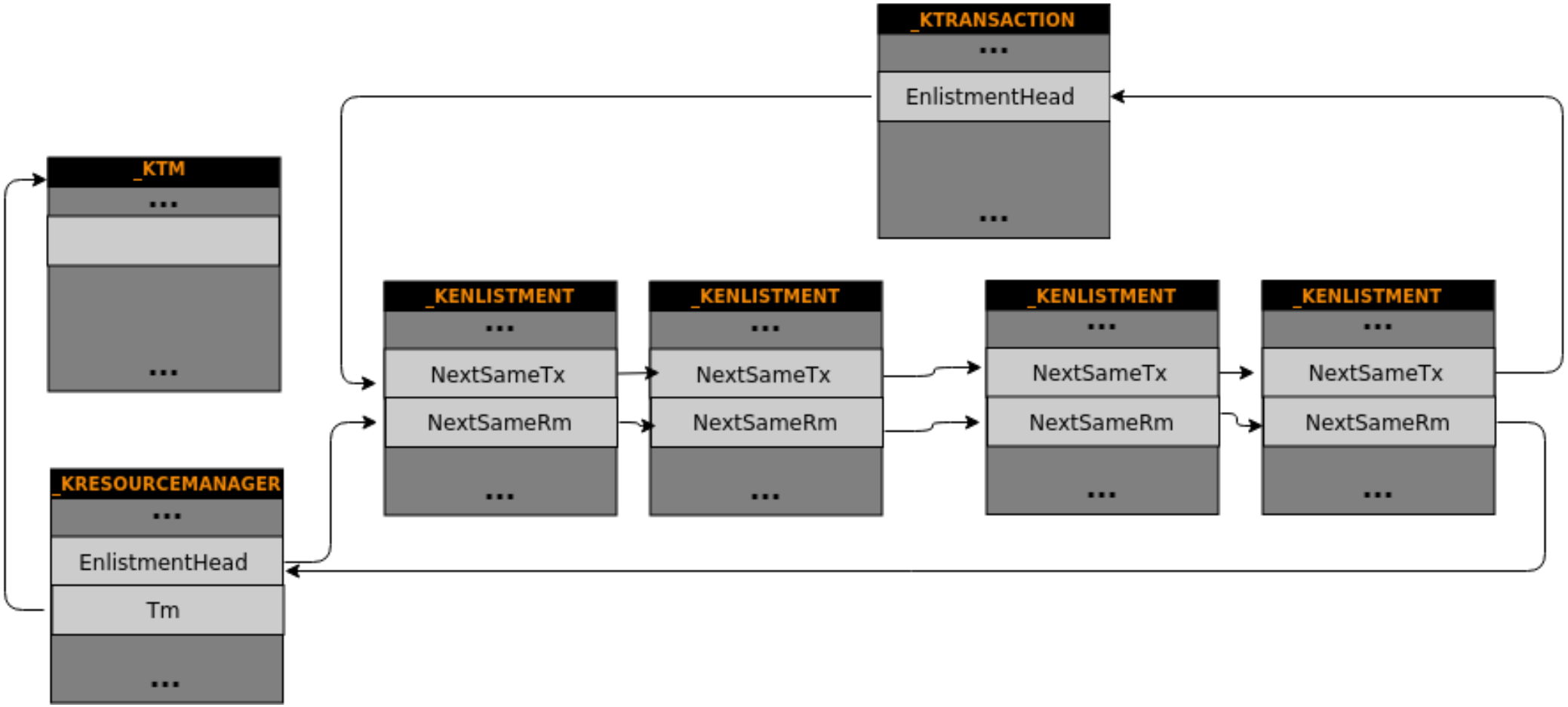
Diffing - Hex-Rays post-cleanup

```
83                  0x20u,
84                  &cur_enlistment_guid);
85          if ( ADJ(pEnlistment_shifted)->Flags & KENLISTMENT_FINALIZED )
86            bEnlistmentIsFinalized = 1;
87          ObfDereferenceObject(ADJ(pEnlistment_shifted));
88          KeWaitForSingleObject(&pResMgr->Mutex, Executive, 0, 0, 0i64);
89          if ( pResMgr->State != KResourceManagerOnline )
90            goto b_release_mutex;
91        }
92        else
93        {
94          ObfDereferenceObject(ADJ(pEnlistment_shifted));
95        }
96        if ( bEnlistmentIsFinalized )
97        {
98          pEnlistment_shifted = EnlistmentHead_addr->Flink;
99          bEnlistmentIsFinalized = 0;
00          bEnlistmentIsFinalized = 0;
01        }
02        else
03        {
04          pEnlistment_shifted = ADJ(pEnlistment_shifted)->NextSameRm.Flink;
05        }
06      }
```

```
83                  0x20u,
84                  &cur_enlistment_guid);
85          ObfDereferenceObject(ADJ(pEnlistment_shifted));
86          KeWaitForSingleObject(&pResMgr->Mutex, Executive, 0, 0, 0i64);
87          if ( pResMgr->State != KResourceManagerOnline )
88            goto b_release_mutex;
89          Tm_ = pResMgr->Tm;
90          if ( !Tm_ || Tm_->State != KKtmOnline )
91          {
92            ret = STATUS_TRANSACTIONMANAGER_NOT_ONLINE;
93            goto b_release_mutex;
94          }
95          pEnlistment_shifted = EnlistmentHead_addr->Flink;
96        }
97        else
98        {
99          ObfDereferenceObject(ADJ(pEnlistment_shifted));
100         pEnlistment_shifted = ADJ(pEnlistment_shifted)->NextSameRm.Flink;
101       }
102     }
```

```
pEnlistment_shifted = EnlistmentHead_addr->Flink;
while ( pEnlistment_shifted != EnlistmentHead_addr ) {
  pEnlistment = ADJ(pEnlistment_shifted)
  if ( pEnlistment->Flags & KENLISTMENT_FINALIZED ) {
    pEnlistment_shifted = pEnlistment->NextSameRm.Flink;
  } else {
    ObfReferenceObject(pEnlistment));
    KeWaitForSingleObject(&pEnlistment->Mutex, Executive, 0, 0, 0i64);
    bSendNotification = 0;
    if ( (pEnlistment->Flags & KENLISTMENT_IS_NOTIFIABLE) != 0 ) {
      // ...
      isSuperior = pEnlistment->Flags & KENLISTMENT_SUPERIOR;
      state = pEnlistment->Transaction->State;
      if ( ... ) {
        // ...
      } else if ((!isSuperior && state == KTransactionCommitted)
              || state == KTransactionInDoubt
              || state == KTransactionPrepared ) {
        bSendNotification = 1;
        NotificationMask = TRANSACTION_NOTIFY_RECOVER;
      }
      pEnlistment->Flags &= ~KENLISTMENT_IS_NOTIFIABLE;
    }
    // ...
    KeReleaseMutex(&pEnlistment->Mutex, 0);

    if ( bSendNotification ) {
      KeReleaseMutex(&pResMgr->Mutex, 0);
      ret = TmpSetNotificationResourceManager( ... );

      if ( pEnlistment->Flags & KENLISTMENT_FINALIZED ) {
        bEnlistmentIsFinalized = 1;
      }

      ObfDereferenceObject(pEnlistment);
      KeWaitForSingleObject(&pResMgr->Mutex, Executive, 0, 0, 0i64);
      //...
    } else {
      ObfDereferenceObject(pEnlistment);
    }

    if ( bEnlistmentIsFinalized ) {
      pEnlistment_shifted = EnlistmentHead_addr->Flink;
      bEnlistmentIsFinalized = 0;
    } else {
      pEnlistment_shifted = pEnlistment->NextSameRm.Flink;
    }
  }
}
```

Vulnerable TmRecoverResourceManager() loop


_KTRANSACTION
...
EnlistmentHead
...
_KTM
...
...
_KENLISTMENT
...
NextSameTx
NextSameRm
...
_KENLISTMENT
...
NextSameTx
NextSameRm
...
_KENLISTMENT
...
NextSameTx
NextSameRm
...
_KENLISTMENT
...
NextSameTx
NextSameRm
...
_KRESOURCEMANAGER
...
EnlistmentHead
Tm
...

```
pEnlistment_shifted = EnlistmentHead_addr->Flink;
while ( pEnlistment_shifted != EnlistmentHead_addr ) {
  pEnlistment = ADJ(pEnlistment_shifted)
  if ( pEnlistment->Flags & KENLISTMENT_FINALIZED ) {
    pEnlistment_shifted = pEnlistment->NextSameRm.Flink;
  } else {
    ObfReferenceObject(pEnlistment));
    KeWaitForSingleObject(&pEnlistment->Mutex, Executive, 0, 0, 0i64);
    bSendNotification = 0;
    if ( (pEnlistment->Flags & KENLISTMENT_IS_NOTIFIABLE) != 0 ) {
      // ...
      isSuperior = pEnlistment->Flags & KENLISTMENT_SUPERIOR;
      state = pEnlistment->Transaction->State;
      if ( ... ) {
        // ...
      } else if ((!isSuperior && state == KTransactionCommitted)
            || state == KTransactionInDoubt
            || state == KTransactionPrepared ) {
        bSendNotification = 1;
        NotificationMask = TRANSACTION_NOTIFY_RECOVER;
      }
      pEnlistment->Flags &= ~KENLISTMENT_IS_NOTIFIABLE;
    }
    // ...
    KeReleaseMutex(&pEnlistment->Mutex, 0);

    if ( bSendNotification ) {
      KeReleaseMutex(&pResMgr->Mutex, 0);
      ret = TmpSetNotificationResourceManager( ... );

      if ( pEnlistment->Flags & KENLISTMENT_FINALIZED ) {
        bEnlistmentIsFinalized = 1;
      }

      ObfDereferenceObject(pEnlistment);
      KeWaitForSingleObject(&pResMgr->Mutex, Executive, 0, 0, 0i64);
      //...
    } else {
      ObfDereferenceObject(pEnlistment);
    }

    if ( bEnlistmentIsFinalized ) {
      pEnlistment_shifted = EnlistmentHead_addr->Flink;
      bEnlistmentIsFinalized = 0;
    } else {
      pEnlistment_shifted = pEnlistment->NextSameRm.Flink;
    }
  }
}
```

```
pEnlistment_shifted = EnlistmentHead_addr->Flink;
while ( pEnlistment_shifted != EnlistmentHead_addr ) {
```

Current enlistment points to _KRESOURCEMANAGER head to exit loop

```
pEnlistment_shifted = EnlistmentHead_addr->Flink;
while ( pEnlistment_shifted != EnlistmentHead_addr ) {
  pEnlistment = ADJ(pEnlistment_shifted)
  if ( pEnlistment->Flags & KENLISTMENT_FINALIZED ) {
    pEnlistment_shifted = pEnlistment->NextSameRm.Flink;
  } else {
    ObfReferenceObject(pEnlistment));
    KeWaitForSingleObject(&pEnlistment->Mutex, Executive, 0, 0, 0i64);
    bSendNotification = 0;
    if ( (pEnlistment->Flags & KENLISTMENT_IS_NOTIFIABLE) != 0 ) {
      // ...
      isSuperior = pEnlistment->Flags & KENLISTMENT_SUPERIOR;
      state = pEnlistment->Transaction->State;
      if ( ... ) {
        // ...
      } else if ((!isSuperior && state == KTransactionCommitted)
            || state == KTransactionInDoubt
            || state == KTransactionPrepared ) {
        bSendNotification = 1;
        NotificationMask = TRANSACTION_NOTIFY_RECOVER;
      }
      pEnlistment->Flags &= ~KENLISTMENT_IS_NOTIFIABLE;
    }
    // ...
    KeReleaseMutex(&pEnlistment->Mutex, 0);

    if ( bSendNotification ) {
      KeReleaseMutex(&pResMgr->Mutex, 0);
      ret = TmpSetNotificationResourceManager( ... );

      if ( pEnlistment->Flags & KENLISTMENT_FINALIZED ) {
        bEnlistmentIsFinalized = 1;
      }

      ObfDereferenceObject(pEnlistment);
      KeWaitForSingleObject(&pResMgr->Mutex, Executive, 0, 0, 0i64);
      //...
    } else {
      ObfDereferenceObject(pEnlistment);
    }

    if ( bEnlistmentIsFinalized ) {
      pEnlistment_shifted = EnlistmentHead_addr->Flink;
      bEnlistmentIsFinalized = 0;
    } else {
      pEnlistment_shifted = pEnlistment->NextSameRm.Flink;
    }
  }
}
```

```
if ( pEnlistment->Flags & KENLISTMENT_FINALIZED ) {
  pEnlistment_shifted = pEnlistment->NextSameRm.Flink;
} else {
```

Won't parse already finalized enlistments

```
pEnlistment_shifted = EnlistmentHead_addr->Flink;
while ( pEnlistment_shifted != EnlistmentHead_addr ) {
  pEnlistment = ADJ(pEnlistment_shifted)
  if ( pEnlistment->Flags & KENLISTMENT_FINALIZED ) {
    pEnlistment_shifted = pEnlistment->NextSameRm.Flink;
  } else {
    ObfReferenceObject(pEnlistment));
    KeWaitForSingleObject(&pEnlistment->Mutex, Executive, 0, 0, 0i64);
    bSendNotification = 0;
    if ( (pEnlistment->Flags & KENLISTMENT_IS_NOTIFIABLE) != 0 ) {
      // ...
      isSuperior = pEnlistment->Flags & KENLISTMENT_SUPERIOR;
      state = pEnlistment->Transaction->State;
      if ( ... ) {
        // ...
      } else if ((!isSuperior && state == KTransactionCommitted)
            || state == KTransactionInDoubt
            || state == KTransactionPrepared ) {
        bSendNotification = 1;
        NotificationMask = TRANSACTION_NOTIFY_RECOVER;
      }
      pEnlistment->Flags &= ~KENLISTMENT_IS_NOTIFIABLE;
    }
    // ...
    KeReleaseMutex(&pEnlistment->Mutex, 0);

    if ( bSendNotification ) {
      KeReleaseMutex(&pResMgr->Mutex, 0);
      ret = TmpSetNotificationResourceManager( ... );

      if ( pEnlistment->Flags & KENLISTMENT_FINALIZED ) {
        bEnlistmentIsFinalized = 1;
      }

      ObfDereferenceObject(pEnlistment);
      KeWaitForSingleObject(&pResMgr->Mutex, Executive, 0, 0, 0i64);
      //...
    } else {
      ObfDereferenceObject(pEnlistment);
    }

    if ( bEnlistmentIsFinalized ) {
      pEnlistment_shifted = EnlistmentHead_addr->Flink;
      bEnlistmentIsFinalized = 0;
    } else {
      pEnlistment_shifted = pEnlistment->NextSameRm.Flink;
    }
  }
}
```

```
ObfReferenceObject(pEnlistment));
KeWaitForSingleObject(&pEnlistment->Mutex, Executive, 0, 0, 0i64);
bSendNotification = 0;
```

Bump the enlistment ref count and lock the current enlistment

Ref count bump prevents deletion upon finalization while sending notification

```
pEnlistment_shifted = EnlistmentHead_addr->Flink;
while ( pEnlistment_shifted != EnlistmentHead_addr ) {
  pEnlistment = ADJ(pEnlistment_shifted)
  if ( pEnlistment->Flags & KENLISTMENT_FINALIZED ) {
    pEnlistment_shifted = pEnlistment->NextSameRm.Flink;
  } else {
    ObfReferenceObject(pEnlistment));
    KeWaitForSingleObject(&pEnlistment->Mutex, Executive, 0, 0, 0i64);
    bSendNotification = 0;
    if ( (pEnlistment->Flags & KENLISTMENT_IS_NOTIFIABLE) != 0 ) {
      // ...
      isSuperior = pEnlistment->Flags & KENLISTMENT_SUPERIOR;
      state = pEnlistment->Transaction->State;
      if ( ... ) {
        // ...
      } else if ((!isSuperior && state == KTransactionCommitted)
            || state == KTransactionInDoubt
            || state == KTransactionPrepared ) {
        bSendNotification = 1;
        NotificationMask = TRANSACTION_NOTIFY_RECOVER;
      }
      pEnlistment->Flags &= ~KENLISTMENT_IS_NOTIFIABLE;
    }
    // ...
    KeReleaseMutex(&pEnlistment->Mutex, 0);

    if ( bSendNotification ) {
      KeReleaseMutex(&pResMgr->Mutex, 0);
      ret = TmpSetNotificationResourceManager( ... );

      if ( pEnlistment->Flags & KENLISTMENT_FINALIZED ) {
        bEnlistmentIsFinalized = 1;
      }

      ObfDereferenceObject(pEnlistment);
      KeWaitForSingleObject(&pResMgr->Mutex, Executive, 0, 0, 0i64);
      //...
    } else {
      ObfDereferenceObject(pEnlistment);
    }

    if ( bEnlistmentIsFinalized ) {
      pEnlistment_shifted = EnlistmentHead_addr->Flink;
      bEnlistmentIsFinalized = 0;
    } else {
      pEnlistment_shifted = pEnlistment->NextSameRm.Flink;
    }
  }
}
```

```
if ( (pEnlistment->Flags & KENLISTMENT_IS_NOTIFIABLE) != 0 ) {
  // ...
  isSuperior = pEnlistment->Flags & KENLISTMENT_SUPERIOR;
  state = pEnlistment->Transactio
  if ( ... ) {
    // ...
  } else if ((!isSuperior && state == KTransactionCommitted)
        || state == KTransactionInDoubt
        || state == KTransactionPrepared ) {
    bSendNotification = 1;
    NotificationMask = TRANSACTION_NOTIFY_RECOVER;
  }
  pEnlistment->Flags &= ~KENLISTMENT_IS_NOTIFIABLE;
}
```

Each enlistment only gets notified once per loop iteration

```
pEnlistment_shifted = EnlistmentHead_addr->Flink;
while ( pEnlistment_shifted != EnlistmentHead_addr ) {
  pEnlistment = ADJ(pEnlistment_shifted)
  if ( pEnlistment->Flags & KENLISTMENT_FINALIZED ) {
    pEnlistment_shifted = pEnlistment->NextSameRm.Flink;
  } else {
    ObfReferenceObject(pEnlistment));
    KeWaitForSingleObject(&pEnlistment->Mutex, Executive, 0, 0, 0i64);
    bSendNotification = 0;
    if ( (pEnlistment->Flags & KENLISTMENT_IS_NOTIFIABLE) != 0 ) {
      // ...
      isSuperior = pEnlistment->Flags & KENLISTMENT_SUPERIOR;
      state = pEnlistment->Transaction->State;
      if ( ... ) {
        // ...
      } else if ((!isSuperior && state == KTransactionCommitted)
            || state == KTransactionInDoubt
            || state == KTransactionPrepared ) {
        bSendNotification = 1;
        NotificationMask = TRANSACTION_NOTIFY_RECOVER;
      }
      pEnlistment->Flags &= ~KENLISTMENT_IS_NOTIFIABLE;
    }
    // ...
    KeReleaseMutex(&pEnlistment->Mutex, 0);

    if ( bSendNotification ) {
      KeReleaseMutex(&pResMgr->Mutex, 0);
      ret = TmpSetNotificationResourceManager( ... );

      if ( pEnlistment->Flags & KENLISTMENT_FINALIZED ) {
        bEnlistmentIsFinalized = 1;
      }

      ObfDereferenceObject(pEnlistment);
      KeWaitForSingleObject(&pResMgr->Mutex, Executive, 0, 0, 0i64);
      //...
    } else {
      ObfDereferenceObject(pEnlistment);
    }

    if ( bEnlistmentIsFinalized ) {
      pEnlistment_shifted = EnlistmentHead_addr->Flink;
      bEnlistmentIsFinalized = 0;
    } else {
      pEnlistment_shifted = pEnlistment->NextSameRm.Flink;
    }
  }
}
```

```
if ( (pEnlistment->Flags & KENLISTMENT_IS_NOTIFIABLE) != 0 ) {
  // ...
  isSuperior = pEnlistment->Flags & KENLISTMENT_SUPERIOR;
  state = pEnlistment->Transaction->State;
  if ( ... ) {
    // ...
  } else if ((!isSuperior && state == KTransactionCommitted)
        || state == KTransactionInDoubt
        || state == KTransactionPrepared ) {
    bSendNotification = 1;
    NotificationMask = TRANSACTION_NOTIFY_RECOVER;
  }
  pEnlistment->Flags &= ~KENLISTMENT_IS_NOTIFIABLE;
}
```

Send an enlistment notification for specific transaction states

```
pEnlistment_shifted = EnlistmentHead_addr->Flink;
while ( pEnlistment_shifted != EnlistmentHead_addr ) {
  pEnlistment = ADJ(pEnlistment_shifted)
  if ( pEnlistment->Flags & KENLISTMENT_FINALIZED ) {
    pEnlistment_shifted = pEnlistment->NextSameRm.Flink;
  } else {
    ObfReferenceObject(pEnlistment));
    KeWaitForSingleObject(&pEnlistment->Mutex, Executive, 0, 0, 0i64);
    bSendNotification = 0;
    if ( (pEnlistment->Flags & KENLISTMENT_IS_NOTIFIABLE) != 0 ) {
      // ...
      isSuperior = pEnlistment->Flags & KENLISTMENT_SUPERIOR;
      state = pEnlistment->Transaction->State;
      if ( ... ) {
        // ...
      } else if ((!isSuperior && state == KTransactionCommitted)
              || state == KTransactionInDoubt
              || state == KTransactionPrepared ) {
        bSendNotification = 1;
        NotificationMask = TRANSACTION_NOTIFY_RECOVER;
      }
      pEnlistment->Flags &= ~KENLISTMENT_IS_NOTIFIABLE;
    }
    // ...
    KeReleaseMutex(&pEnlistment->Mutex, 0);

    if ( bSendNotification ) {
      KeReleaseMutex(&pResMgr->Mutex, 0);
      ret = TmpSetNotificationResourceManager( ... );

      if ( pEnlistment->Flags & KENLISTMENT_FINALIZED ) {
        bEnlistmentIsFinalized = 1;
      }

      ObfDereferenceObject(pEnlistment);
      KeWaitForSingleObject(&pResMgr->Mutex, Executive, 0, 0, 0i64);
      //...
    } else {
      ObfDereferenceObject(pEnlistment);
    }

    if ( bEnlistmentIsFinalized ) {
      pEnlistment_shifted = EnlistmentHead_addr->Flink;
      bEnlistmentIsFinalized = 0;
    } else {
      pEnlistment_shifted = pEnlistment->NextSameRm.Flink;
    }
  }
}
```

```
// ...
KeReleaseMutex(&pEnlistment->Mutex, 0);

if ( bSendNotification ) {
  KeReleaseMutex(&pResMgr->Mutex, 0);
  ret = TmpSetNotificationResourceManager( ... );
```

Unlock resource manager mutex! Finalizing enlistments is now possible, which can lead to deletion if refcount = 0

```
pEnlistment_shifted = EnlistmentHead_addr->Flink;
while ( pEnlistment_shifted != EnlistmentHead_addr ) {
  pEnlistment = ADJ(pEnlistment_shifted)
  if ( pEnlistment->Flags & KENLISTMENT_FINALIZED ) {
    pEnlistment_shifted = pEnlistment->NextSameRm.Flink;
  } else {
    ObfReferenceObject(pEnlistment));
    KeWaitForSingleObject(&pEnlistment->Mutex, Executive, 0, 0, 0i64);
    bSendNotification = 0;
    if ( (pEnlistment->Flags & KENLISTMENT_IS_NOTIFIABLE) != 0 ) {
      // ...
      isSuperior = pEnlistment->Flags & KENLISTMENT_SUPERIOR;
      state = pEnlistment->Transaction->State;
      if ( ... ) {
        // ...
      } else if ((!isSuperior && state == KTransactionCommitted)
              || state == KTransactionInDoubt
              || state == KTransactionPrepared ) {
        bSendNotification = 1;
        NotificationMask = TRANSACTION_NOTIFY_RECOVER;
      }
      pEnlistment->Flags &= ~KENLISTMENT_IS_NOTIFIABLE;
    }
    // ...
    KeReleaseMutex(&pEnlistment->Mutex, 0);

    if ( bSendNotification ) {
      KeReleaseMutex(&pResMgr->Mutex, 0);
      ret = TmpSetNotificationResourceManager( ... );

      if ( pEnlistment->Flags & KENLISTMENT_FINALIZED ) {
        bEnlistmentIsFinalized = 1;
      }

      ObfDereferenceObject(pEnlistment);
      KeWaitForSingleObject(&pResMgr->Mutex, Executive, 0, 0, 0i64);
      //...
    } else {
      ObfDereferenceObject(pEnlistment);
    }

    if ( bEnlistmentIsFinalized ) {
      pEnlistment_shifted = EnlistmentHead_addr->Flink;
      bEnlistmentIsFinalized = 0;
    } else {
      pEnlistment_shifted = pEnlistment->NextSameRm.Flink;
    }
  }
}
```

```
if ( pEnlistment->Flags & KENLISTMENT_FINALIZED ) {
  bEnlistmentIsFinalized = 1;
}
```

Attempt to prevent a use-after-free

Will not use finalized enlistment here if boolean is set

```
pEnlistment_shifted = EnlistmentHead_addr->Flink;
while ( pEnlistment_shifted != EnlistmentHead_addr ) {
  pEnlistment = ADJ(pEnlistment_shifted)
  if ( pEnlistment->Flags & KENLISTMENT_FINALIZED ) {
    pEnlistment_shifted = pEnlistment->NextSameRm.Flink;
  } else {
    ObfReferenceObject(pEnlistment));
    KeWaitForSingleObject(&pEnlistment->Mutex, Executive, 0, 0, 0i64);
    bSendNotification = 0;
    if ( (pEnlistment->Flags & KENLISTMENT_IS_NOTIFIABLE) != 0 ) {
      // ...
      isSuperior = pEnlistment->Flags & KENLISTMENT_SUPERIOR;
      state = pEnlistment->Transaction->State;
      if ( ... ) {
        // ...
      } else if ((!isSuperior && state == KTransactionCommitted)
              || state == KTransactionInDoubt
              || state == KTransactionPrepared ) {
        bSendNotification = 1;
        NotificationMask = TRANSACTION_NOTIFY_RECOVER;
      }
      pEnlistment->Flags &= ~KENLISTMENT_IS_NOTIFIABLE;
    }
    // ...
    KeReleaseMutex(&pEnlistment->Mutex, 0);

    if ( bSendNotification ) {
      KeReleaseMutex(&pResMgr->Mutex, 0);
      ret = TmpSetNotificationResourceManager( ... );

      if ( pEnlistment->Flags & KENLISTMENT_FINALIZED ) {
        bEnlistmentIsFinalized = 1;
      }

      ObfDereferenceObject(pEnlistment);
      KeWaitForSingleObject(&pResMgr->Mutex, Executive, 0, 0, 0i64);
      //...
    } else {
      ObfDereferenceObject(pEnlistment);
    }

    if ( bEnlistmentIsFinalized ) {
      pEnlistment_shifted = EnlistmentHead_addr->Flink;
      bEnlistmentIsFinalized = 0;
    } else {
      pEnlistment_shifted = pEnlistment->NextSameRm.Flink;
    }
  }
}
```

```
ObfDereferenceObject(pEnlistment);
KeWaitForSingleObject(&pResMgr->Mutex, Executive, 0, 0, 0i64);
//...
```

Lower ref count. If enlistment is finalized before relocking mutex, pEnlistment points to freed memory

Prone to race condition abuse. Can congest this mutex from userland.

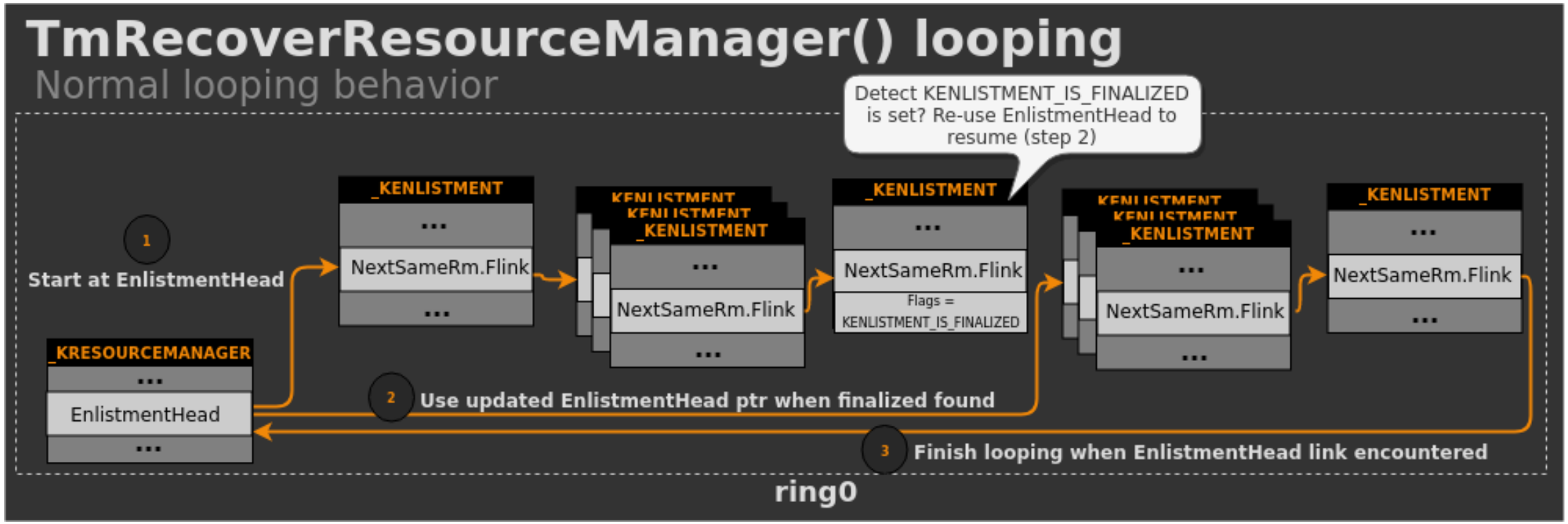
```
pEnlistment_shifted = EnlistmentHead_addr->Flink;
while ( pEnlistment_shifted != EnlistmentHead_addr ) {
  pEnlistment = ADJ(pEnlistment_shifted)
  if ( pEnlistment->Flags & KENLISTMENT_FINALIZED ) {
    pEnlistment_shifted = pEnlistment->NextSameRm.Flink;
  } else {
    ObfReferenceObject(pEnlistment));
    KeWaitForSingleObject(&pEnlistment->Mutex, Executive, 0, 0, 0i64);
    bSendNotification = 0;
    if ( (pEnlistment->Flags & KENLISTMENT_IS_NOTIFIABLE) != 0 ) {
      // ...
      isSuperior = pEnlistment->Flags & KENLISTMENT_SUPERIOR;
      state = pEnlistment->Transaction->State;
      if ( ... ) {
        // ...
      } else if ((!isSuperior && state == KTransactionCommitted)
              || state == KTransactionInDoubt
              || state == KTransactionPrepared ) {
        bSendNotification = 1;
        NotificationMask = TRANSACTION_NOTIFY_RECOVER;
      }
      pEnlistment->Flags &= ~KENLISTMENT_IS_NOTIFIABLE;
    }
    // ...
    KeReleaseMutex(&pEnlistment->Mutex, 0);

    if ( bSendNotification ) {
      KeReleaseMutex(&pResMgr->Mutex, 0);
      ret = TmpSetNotificationResourceManager( ... );

      if ( pEnlistment->Flags & KENLISTMENT_FINALIZED ) {
        bEnlistmentIsFinalized = 1;
      }

      ObfDereferenceObject(pEnlistment);
      KeWaitForSingleObject(&pResMgr->Mutex, Executive, 0, 0, 0i64);
      //...
    } else {
      ObfDereferenceObject(pEnlistment);
    }

    if ( bEnlistmentIsFinalized ) {
      pEnlistment_shifted = EnlistmentHead_addr->Flink;
      bEnlistmentIsFinalized = 0;
    } else {
      pEnlistment_shifted = pEnlistment->NextSameRm.Flink;
    }
  }
}
```

Safe use of resource managers head pointer if race lost

```
if ( bEnlistmentIsFinalized ) {
  pEnlistment_shifted = EnlistmentHead_addr->Flink;
  bEnlistmentIsFinalized = 0;
} else {
  pEnlistment_shifted = pEnlistment->NextSameRm.Flink;
}
```

Used after free if race condition is won

What does `TmRecoverResourceManager()` normally do?



Vulnerability analysis key points

- A recovering `_KRESOURCEMANAGER` is unlocked in order to queue a notification
- Code retains pointer to associated `_KENLISTMENT`, but no lock
- Sends notifications about said `_KENLISTMENT`
- Attempts to tell if `_KENLISTMENT` is finalized, but in a racable location
- Drops the reference count by 1, which allows it to become freed if already finalized
- Relocks `_KRESOURCEMANAGER`
- Tests for a boolean that wasn't set if race condition occurs
- Uses retained `_KENLISTMENT` pointer
- `_KENLISTMENT` could now be freed

Triggering CVE-2018-8611

Faking a race win

- Use WinDbg to force race window open
- Patch `KeWaitForSingleObject()` so we guarantee `pEnlistment` is freed
  - Patch is just an infinite loop

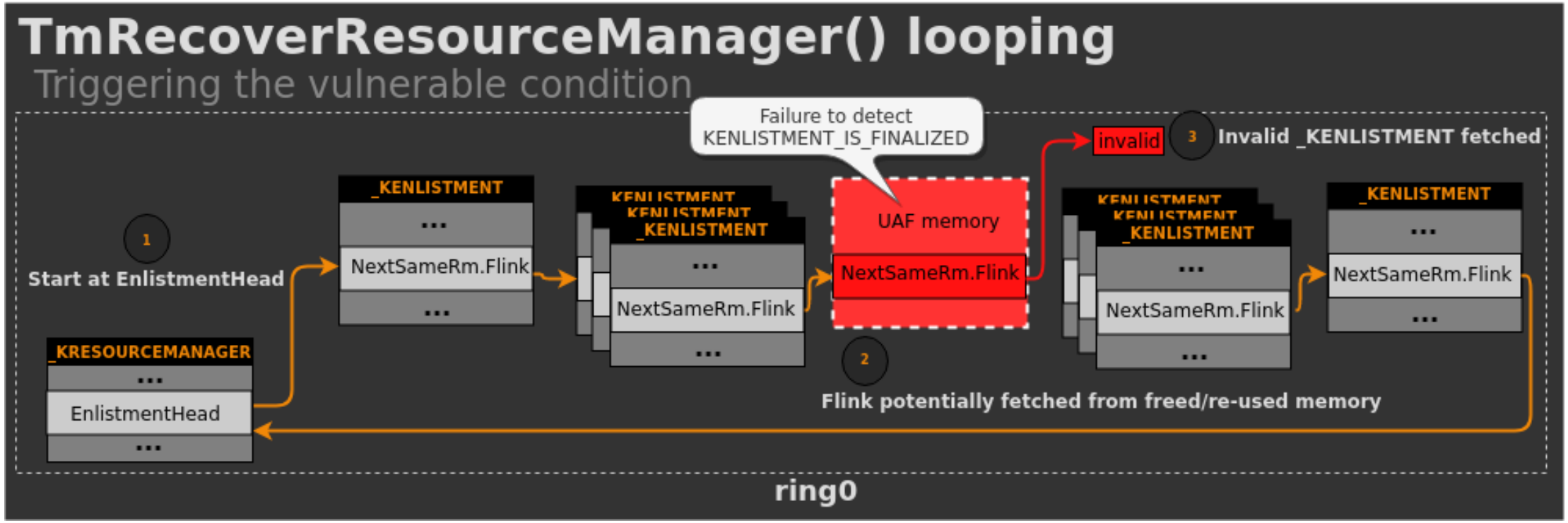
```
        //...
          ObfDereferenceObject(pEnlistment);
          KeWaitForSingleObject(&pResMgr->Mutex, Executive, 0, 0, 0i64);
          //...
        } else {
          ObfDereferenceObject(pEnlistment);
        }

        if ( bEnlistmentIsFinalized ) {
          pEnlistment_shifted = EnlistmentHead_addr->Flink;
          bEnlistmentIsFinalized = 0;
        } else {
          pEnlistment_shifted = pEnlistment->NextSameRm.Flink;
        }
```

- After freeing all `_KENLISTMENTS` test if `pEnlistment->NextSameRm` references freed memory

Which _KENLISTMENT to free?

- If we spam a lot of _KENLISTMENT and try to repeatably race...
  - How do we know which one to free?
  - Can't just free them all every time, as we want to maximize attempts
- GetNotificationResourceManager() tells us what a enlistment has been touched by the loop!
- Vulnerable function unlocks the RM specifically to send a notification
  - Correlate the notification to the enlistment, and free it
- Remove infinite loop after we triggered free from userland
- If UAF triggers, it confirms our understanding of the bug
- Run with Driver Verifier to easily confirm

Exploitable loop state

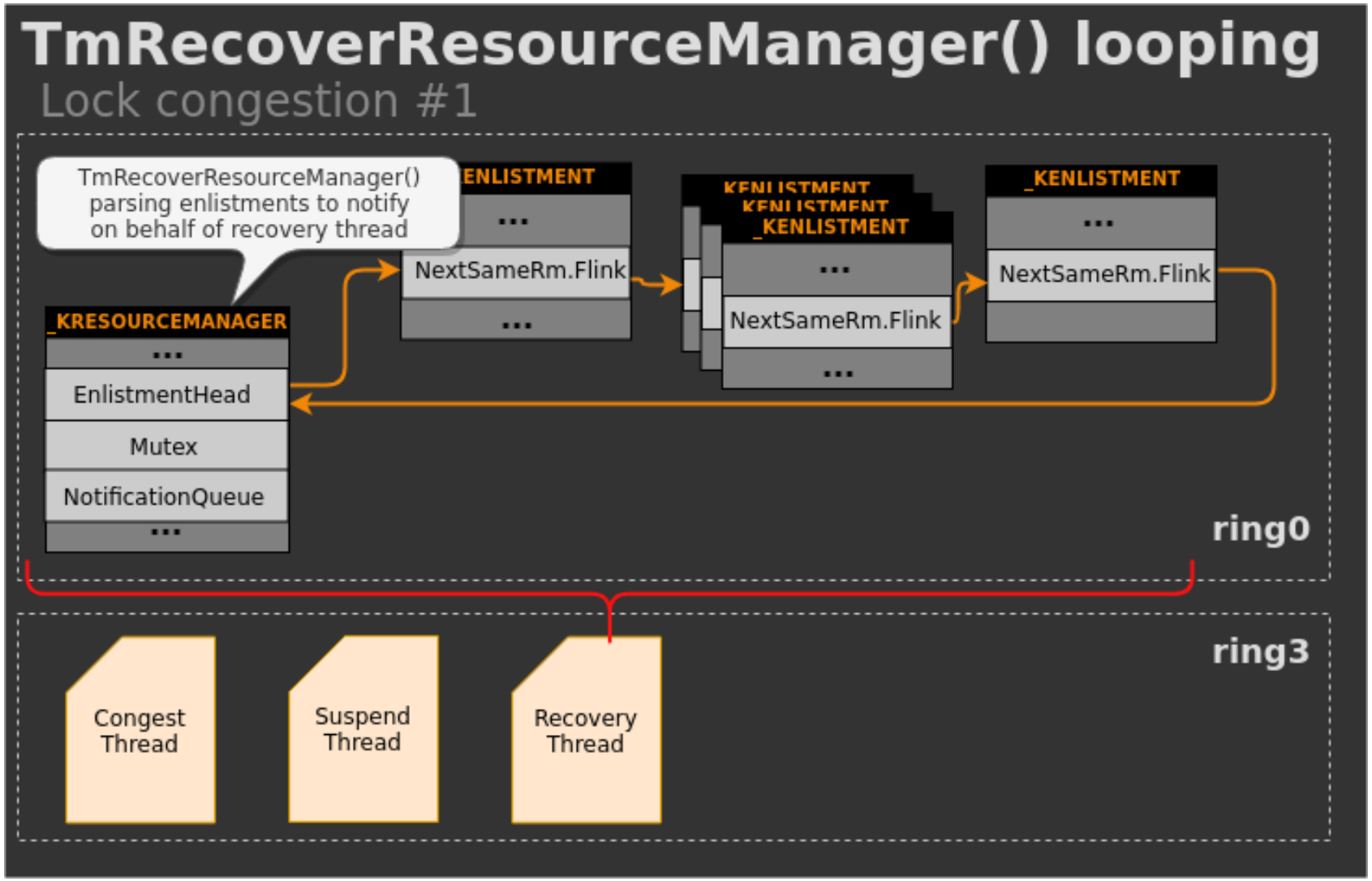


Actually winning the race

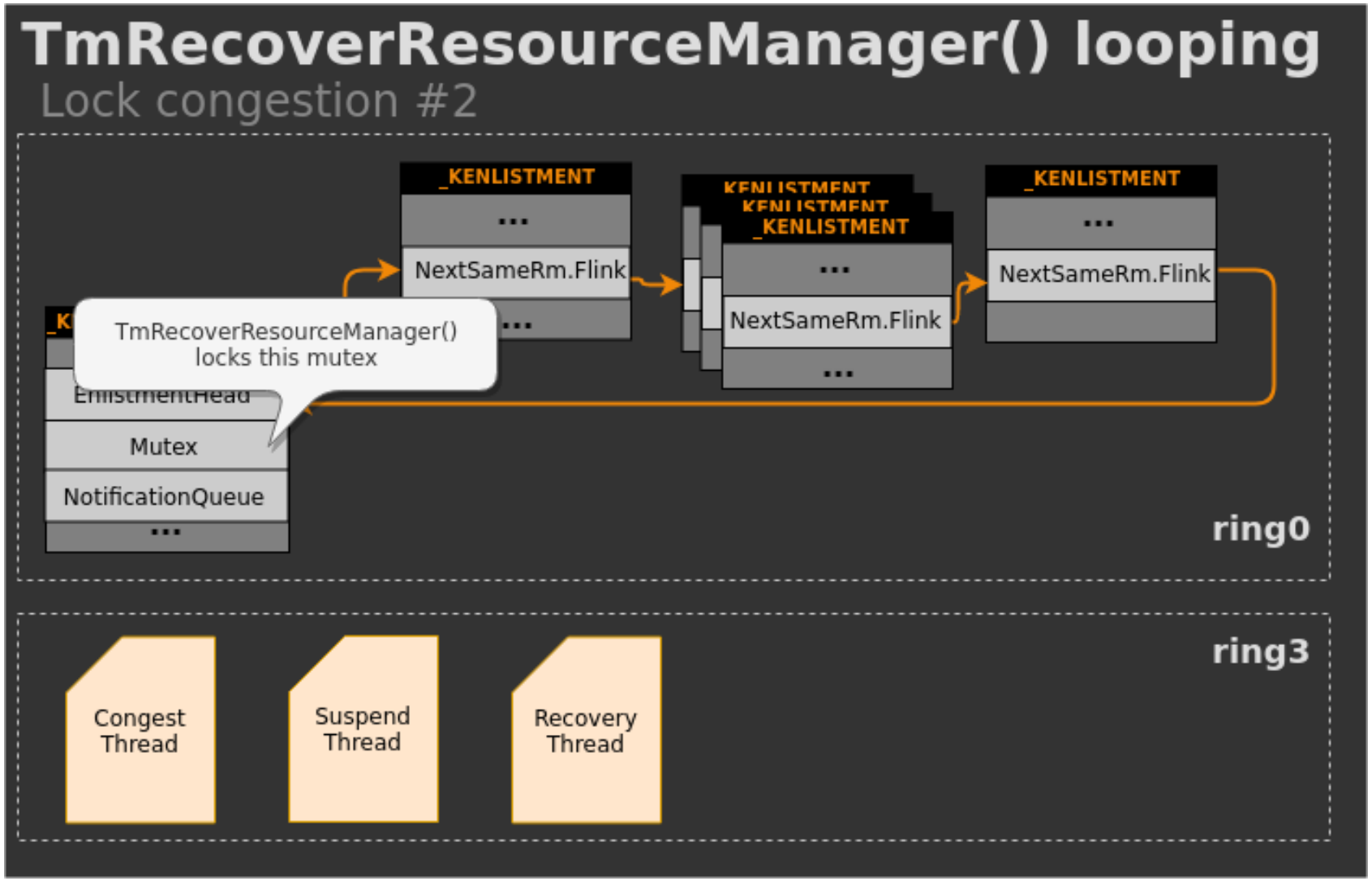
- How do we win this race without patching `KeWaitForSingleObject()`?
  - Was hinted in the Kaspersky blog (though still not obvious to us for quite some time)
  - Suspend the thread stuck in the `TmRecoverResourceManager()` causing it to effectively block until woken up
- A thread will become blocked on some natural blocking point
  - Like waiting to lock the congested RM mutex
- Congest RM lock to increase likelihood of thread suspending where we want
  - Have a higher priority thread constantly triggering syscall that locks RM
  - Ex: Query the RM description (`NtQueryInformationResourceManager`)

- Result
  - If thread suspended at right place, we have all the time in the world to free the enlistment
  - If not, no UAF happens, and we keep trying

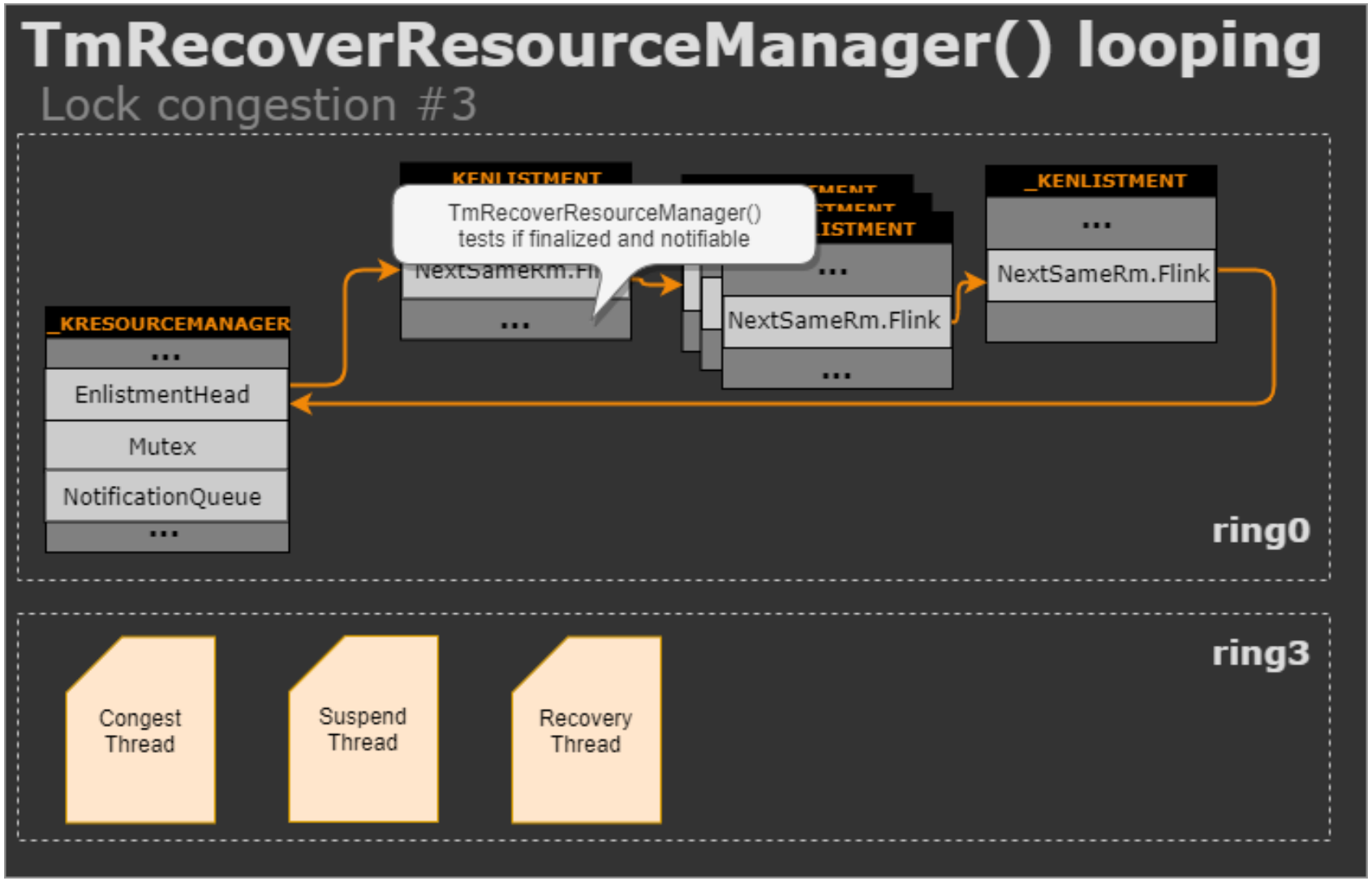
Lock congestion



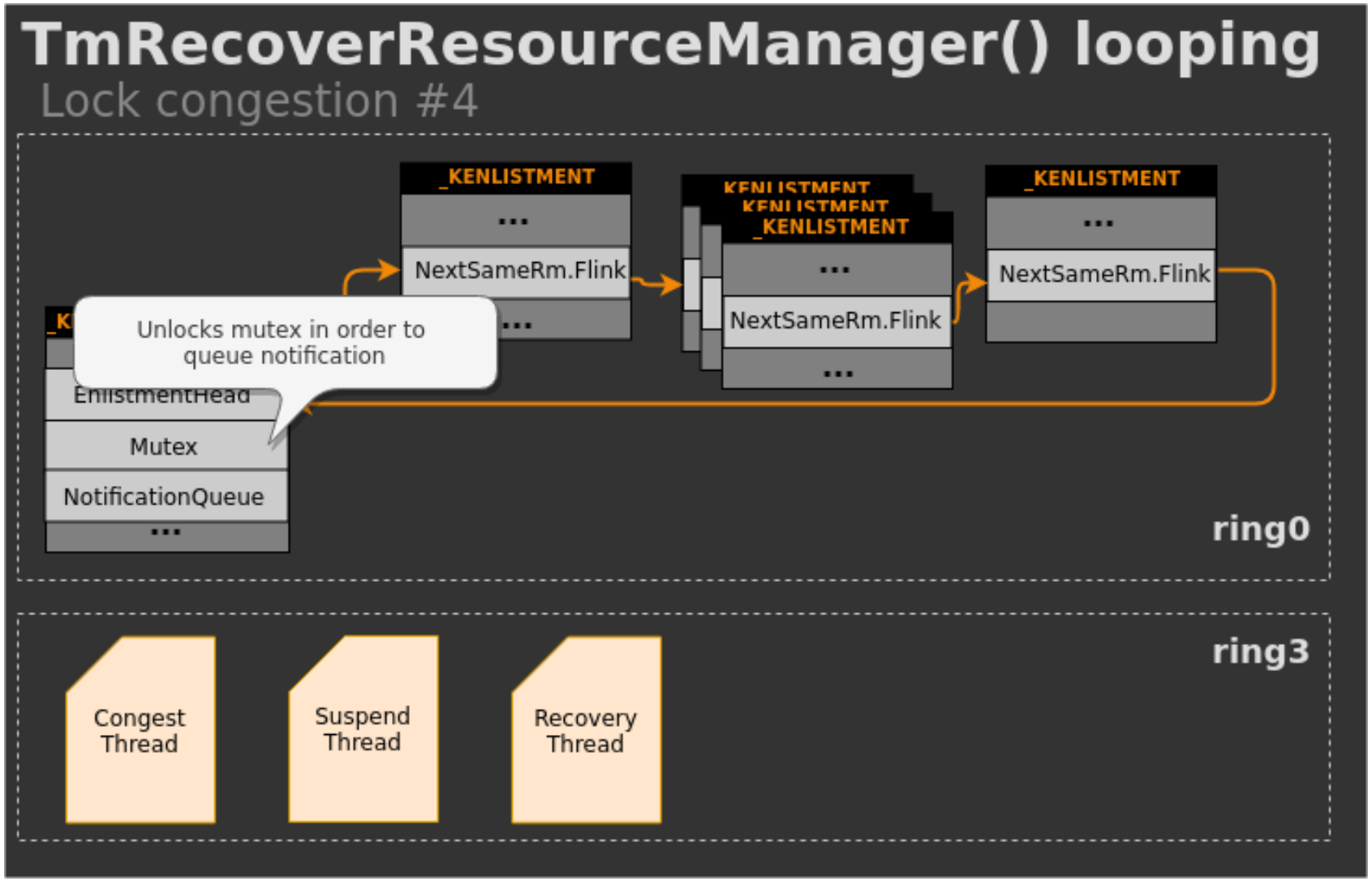
Lock congestion



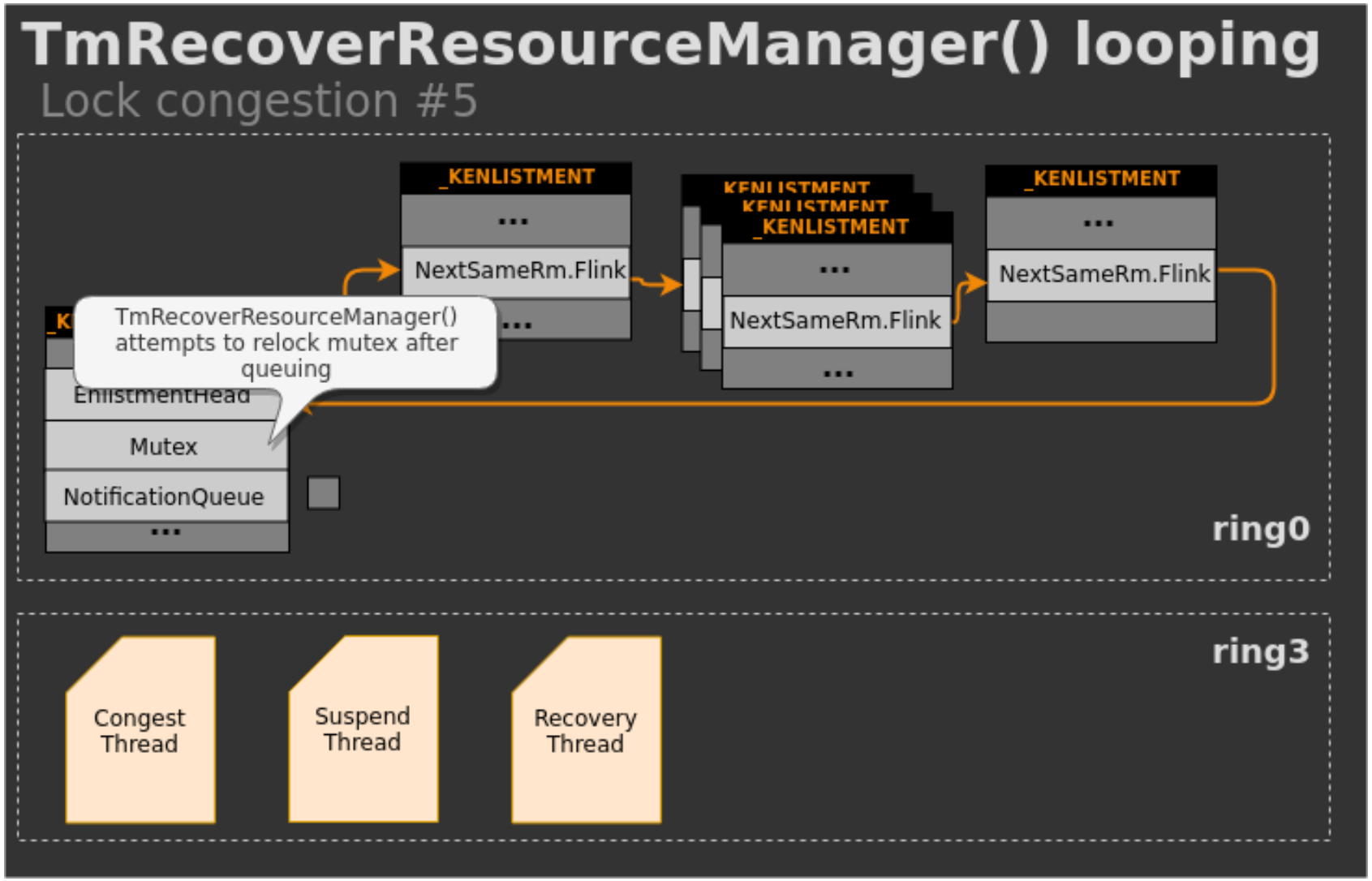
Lock congestion



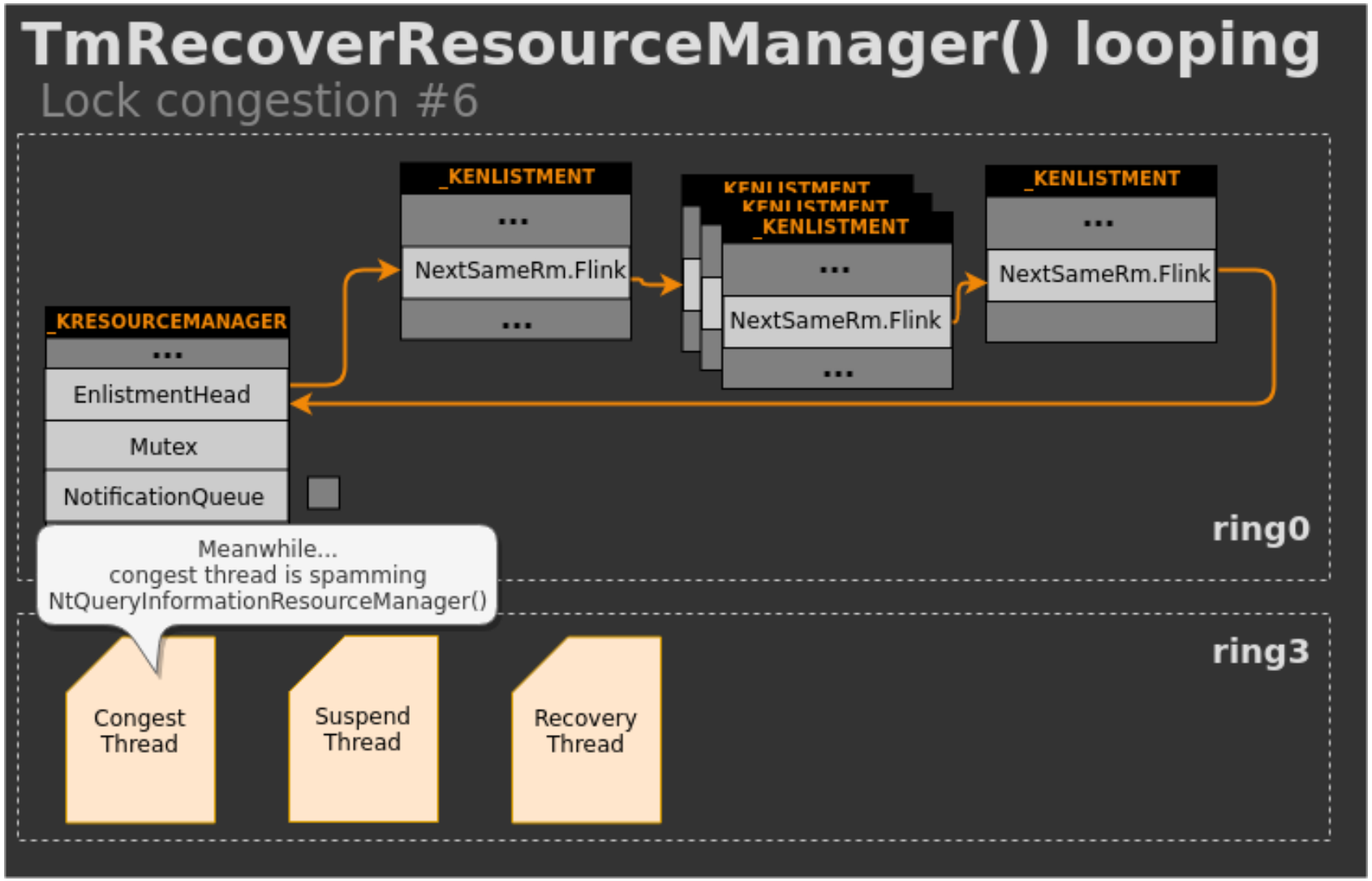
Lock congestion



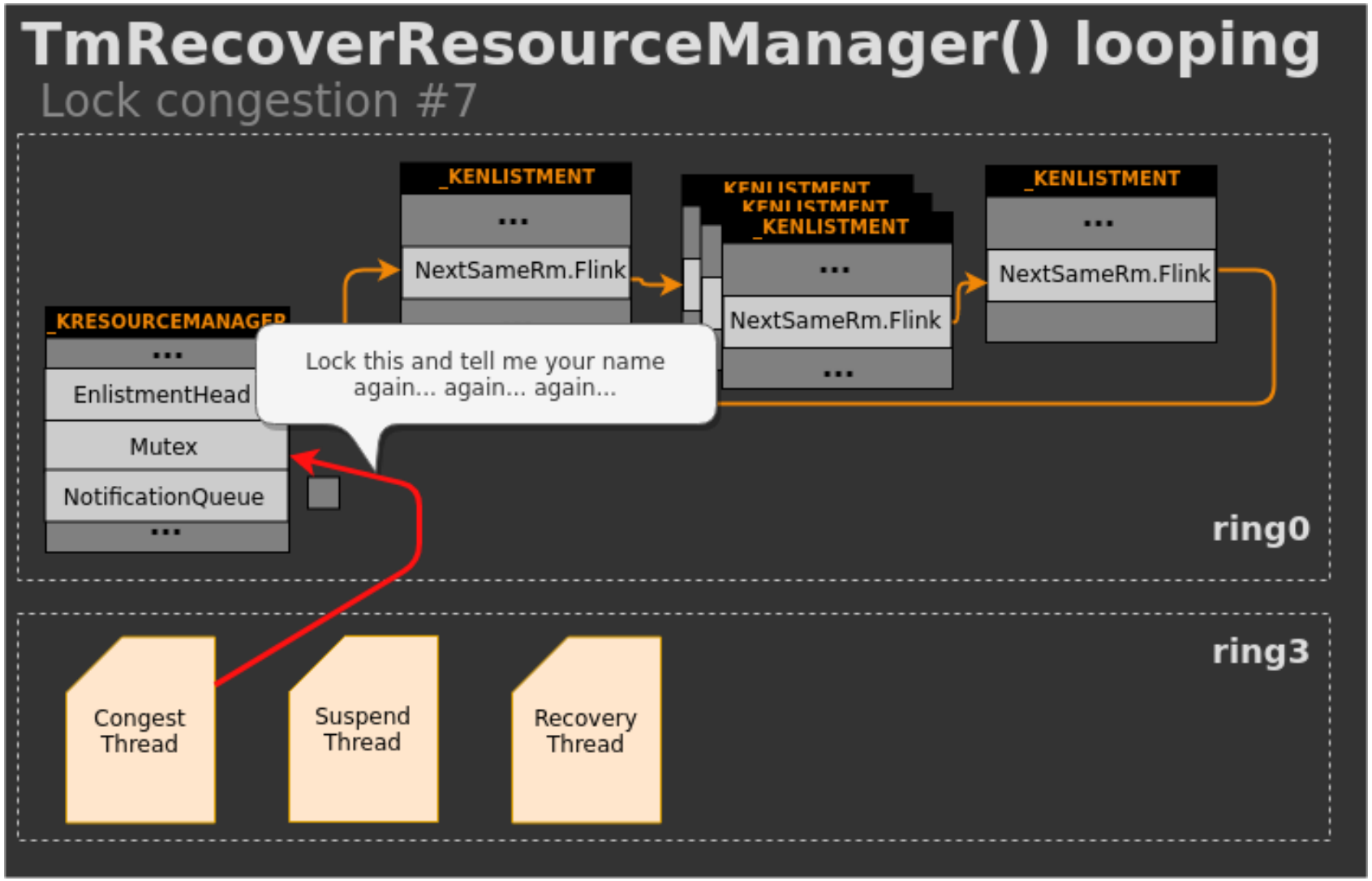
Lock congestion



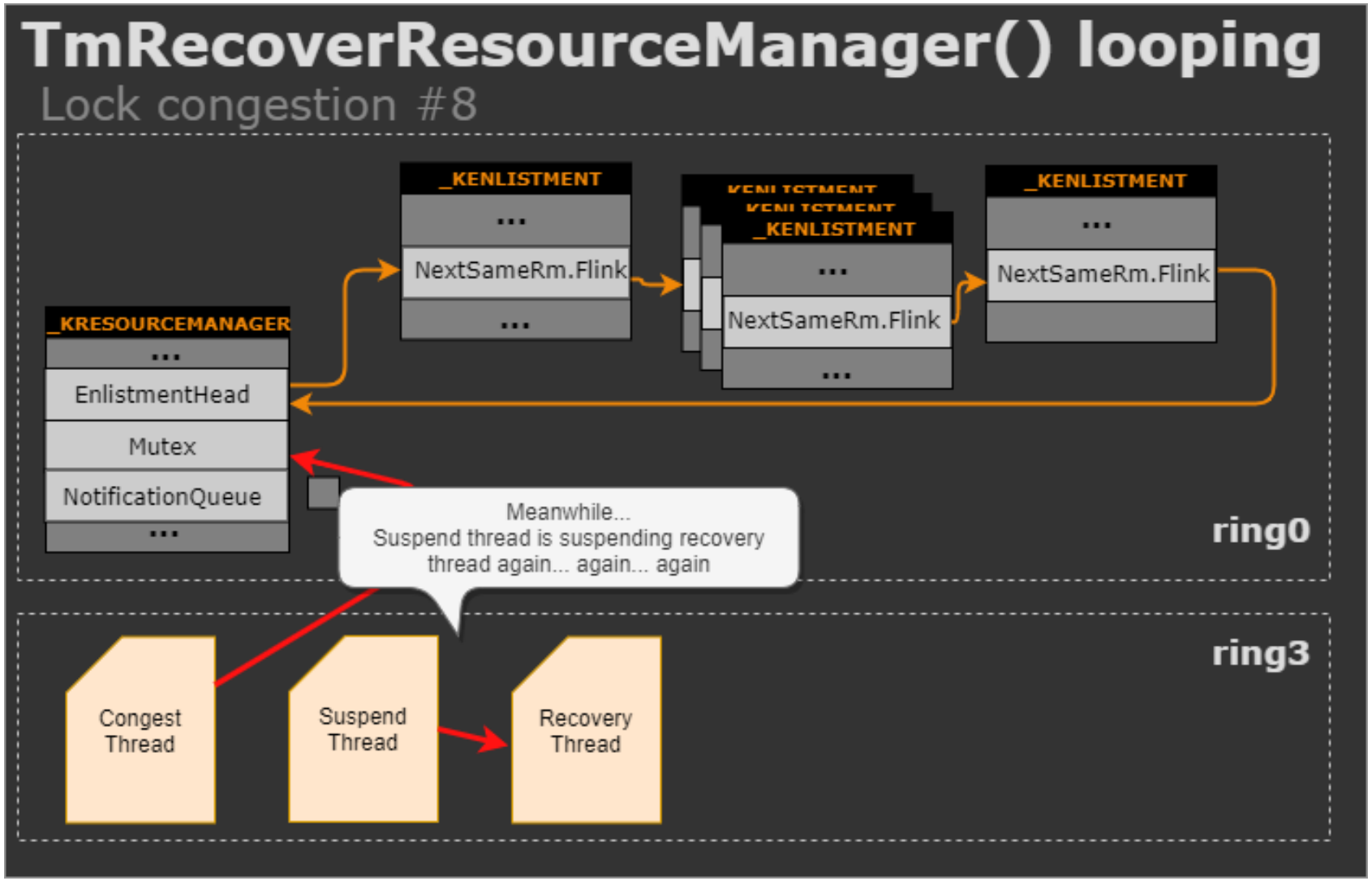
Lock congestion



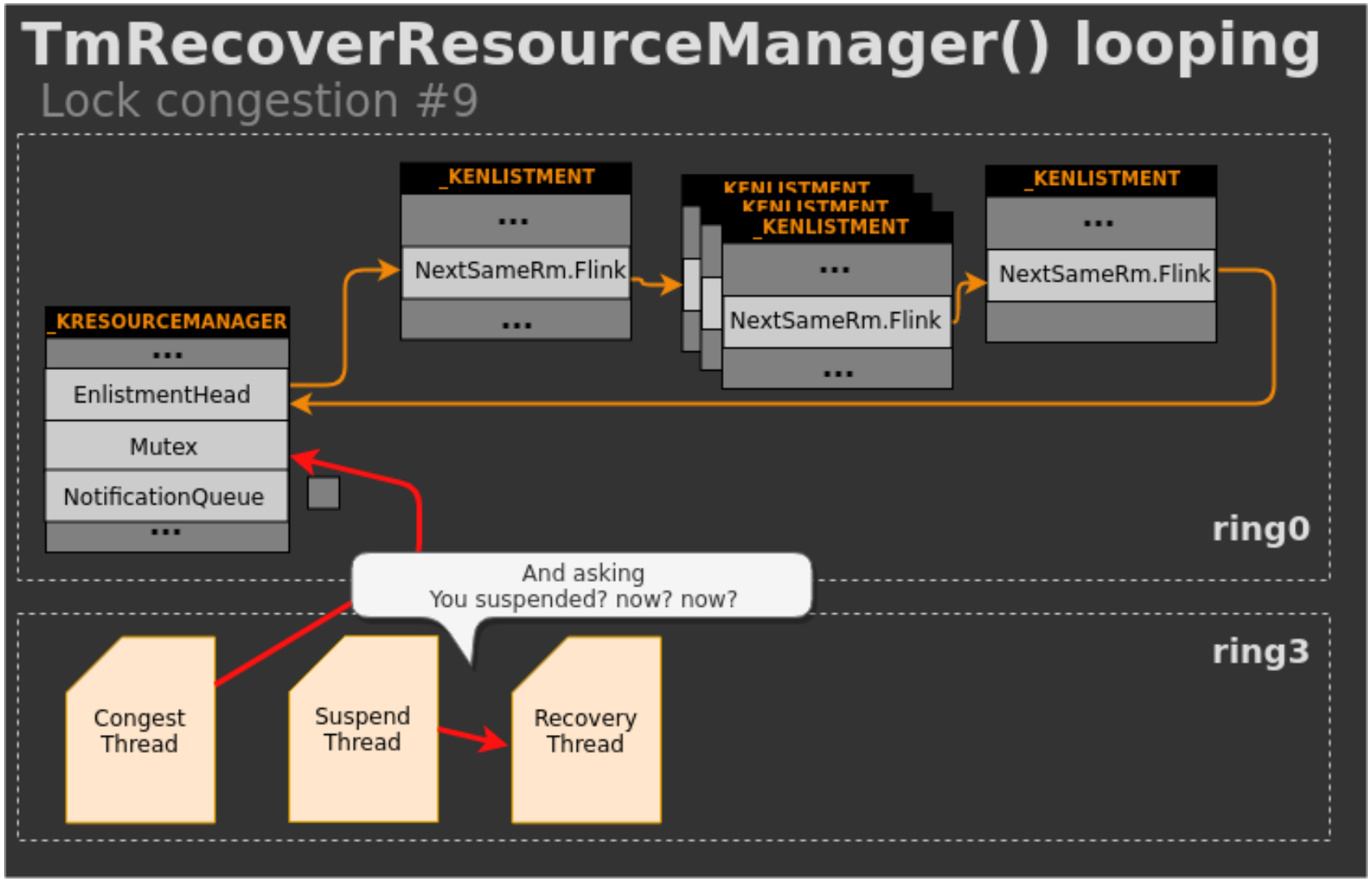
Lock congestion



Lock congestion



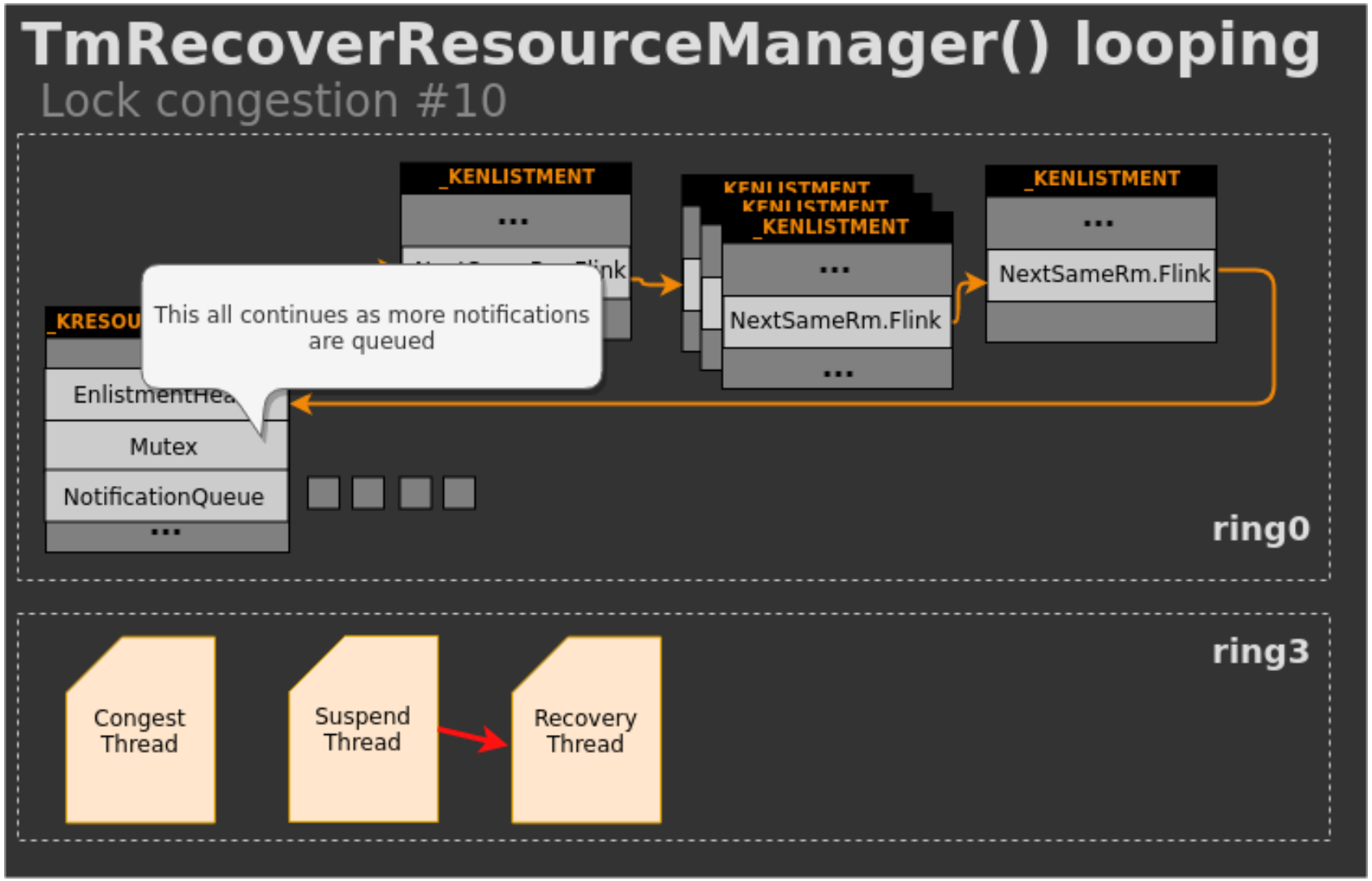
Lock congestion



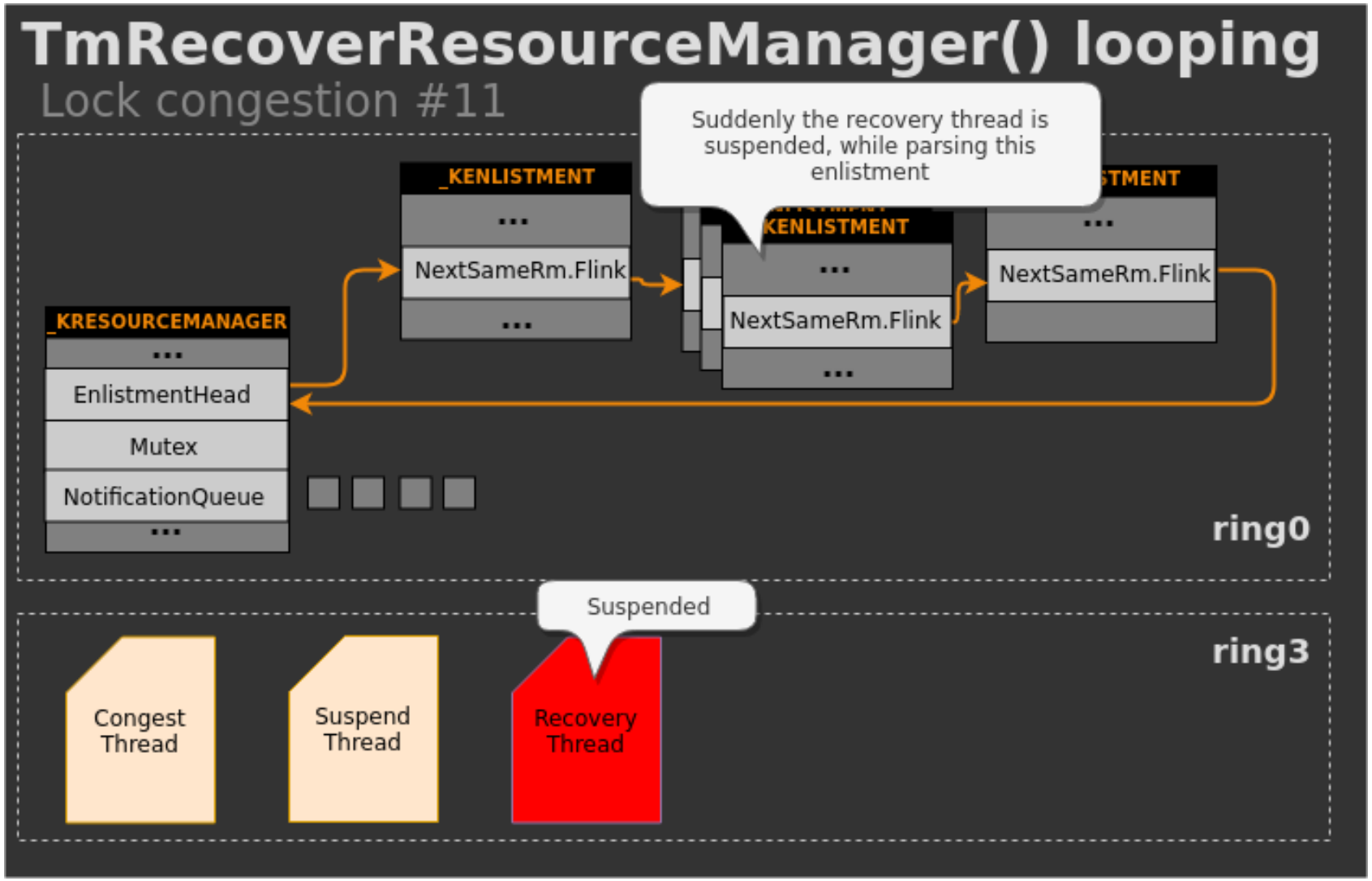
Thread suspension detection

- How can you tell if a thread is suspended?
  - Use `NtQueryThreadInformation()` to query thread
  - `ThreadInformationClass` of `ThreadLastSyscall`
  - Returns `STATUS_UNSUCCESSFUL` if thread is not suspended

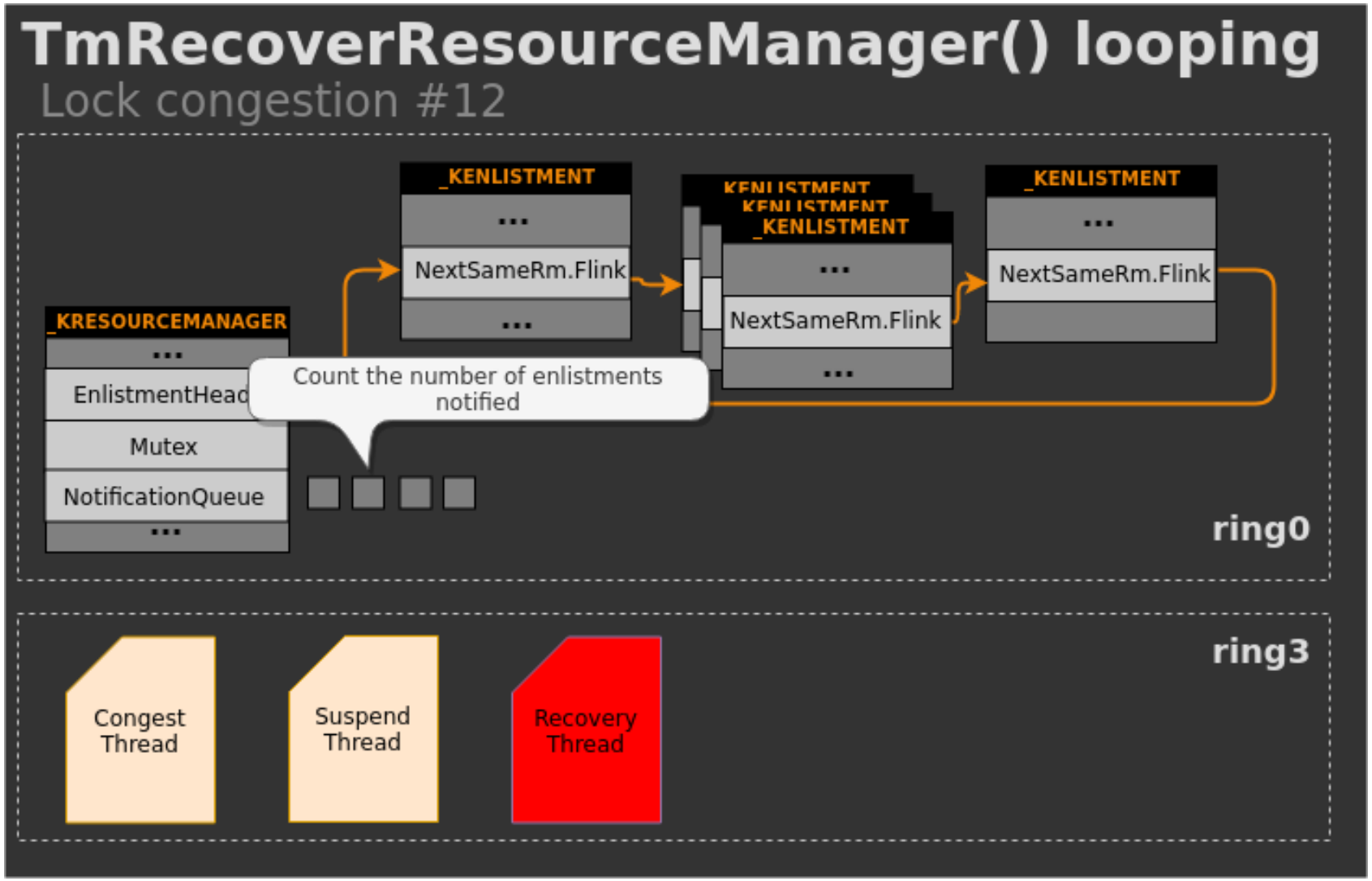
Lock congestion



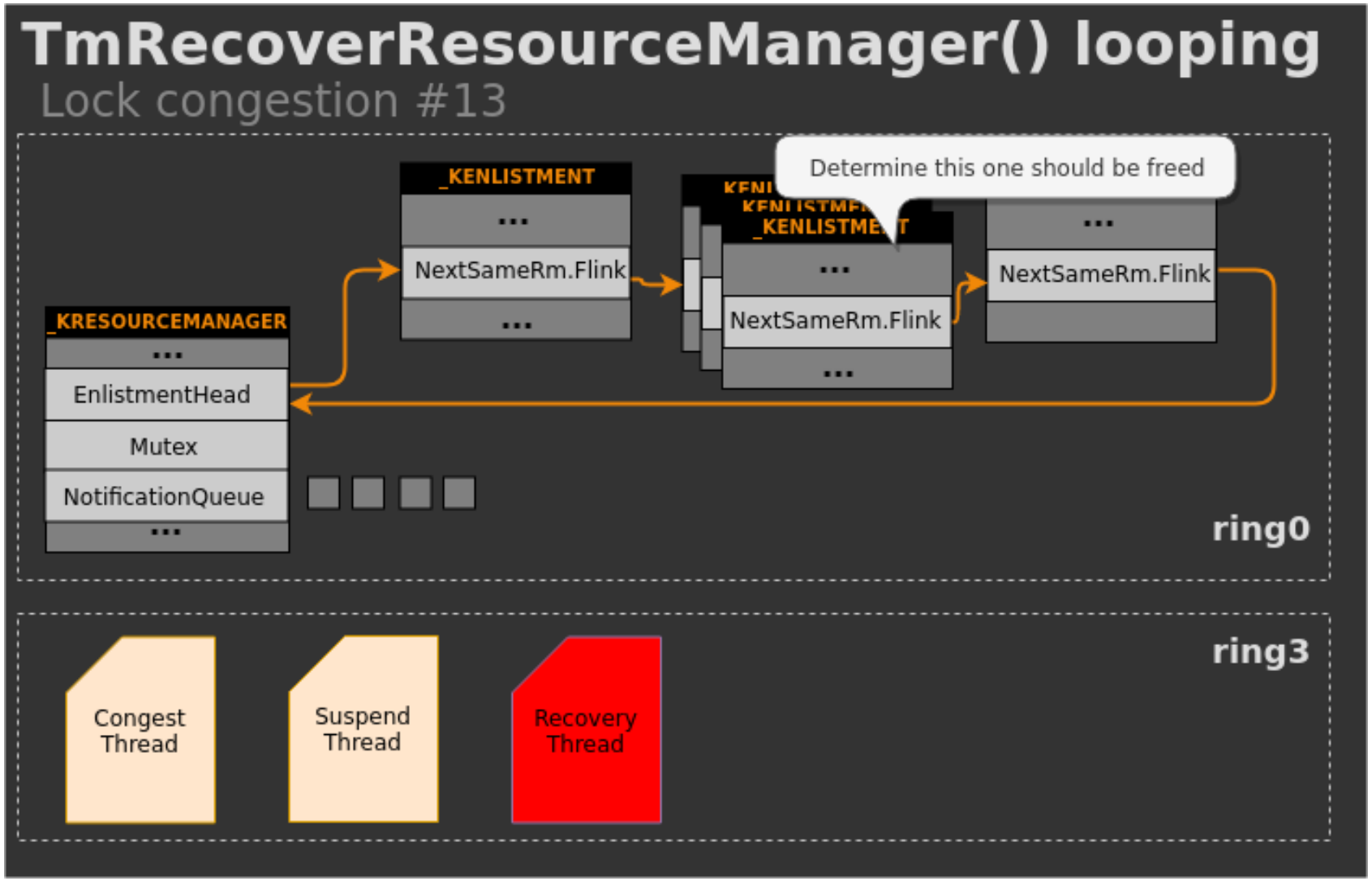
Lock congestion



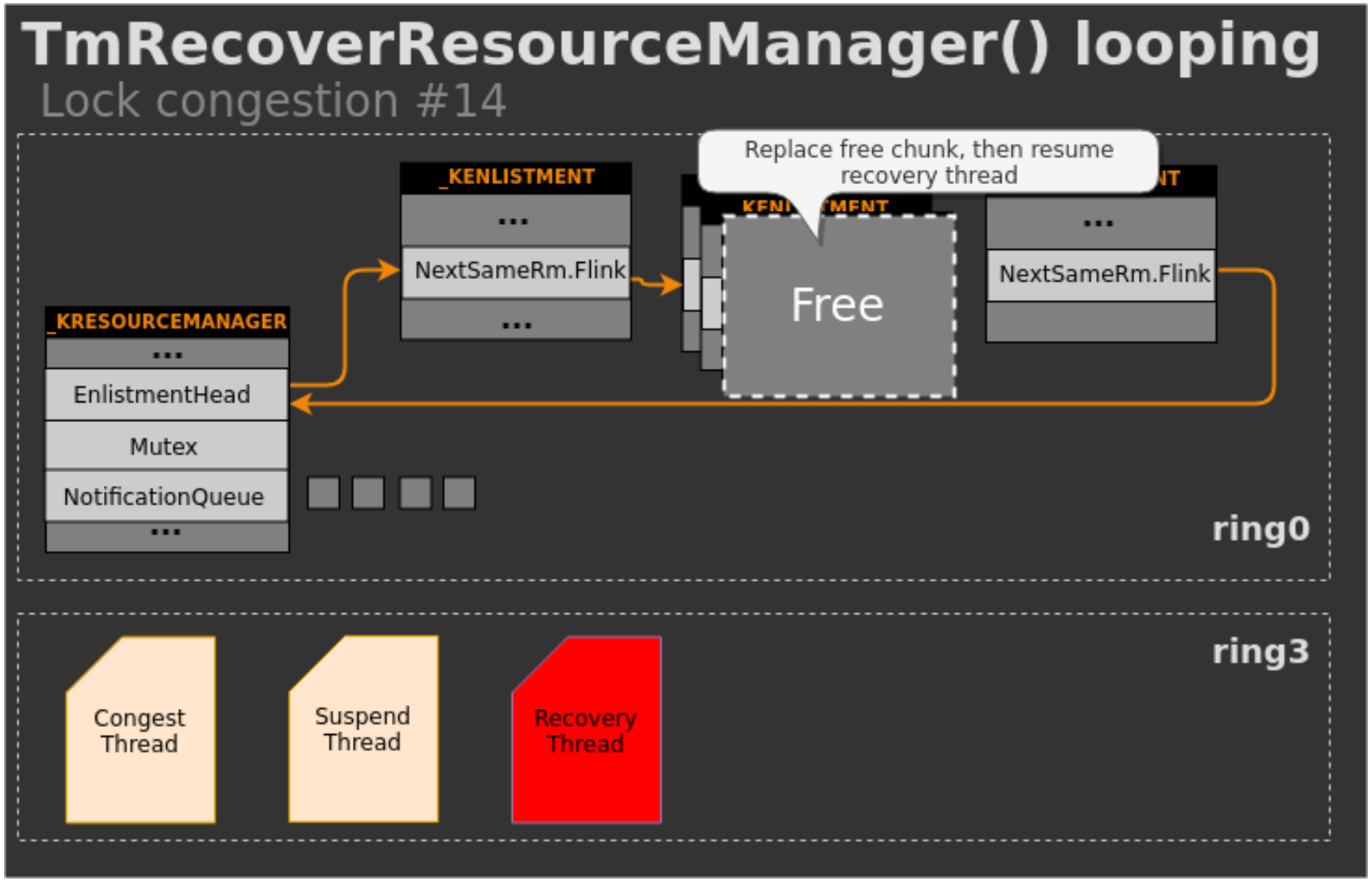
Lock congestion



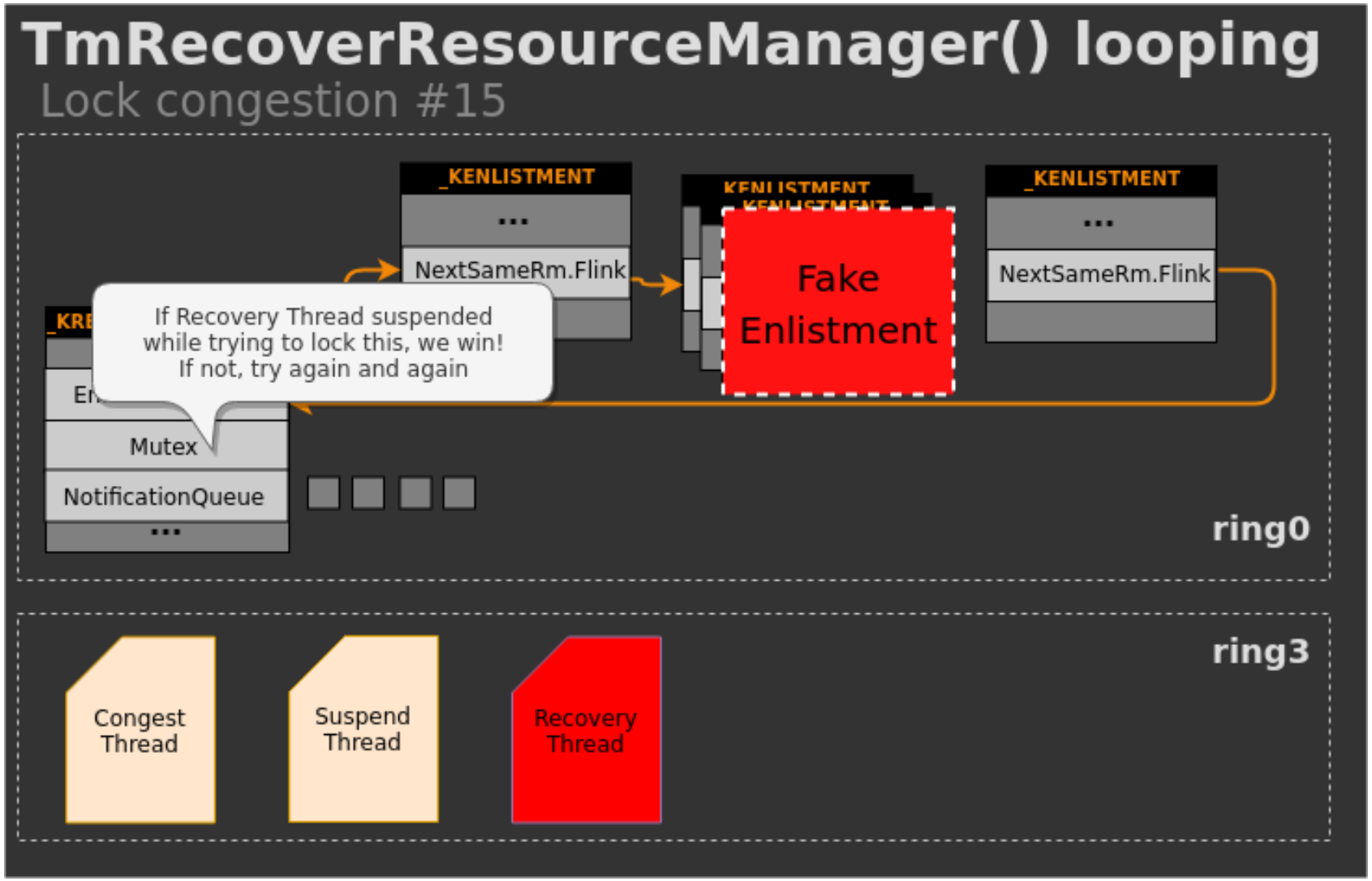
Lock congestion



Lock congestion



Lock congestion



_KENLISTMENT replacement

- We know everything is on the non-paged pool
- We know the size of the _KENLISTMENT
- Non-paged pool feng shui is the obvious approach

Non-Paged pool feng shui

- Widely known, not too widely shared?
  - Alex Ionescu in 2014
  - Andreas Fobian from Blue Frost Security in 2020

Non-Paged pool feng shui

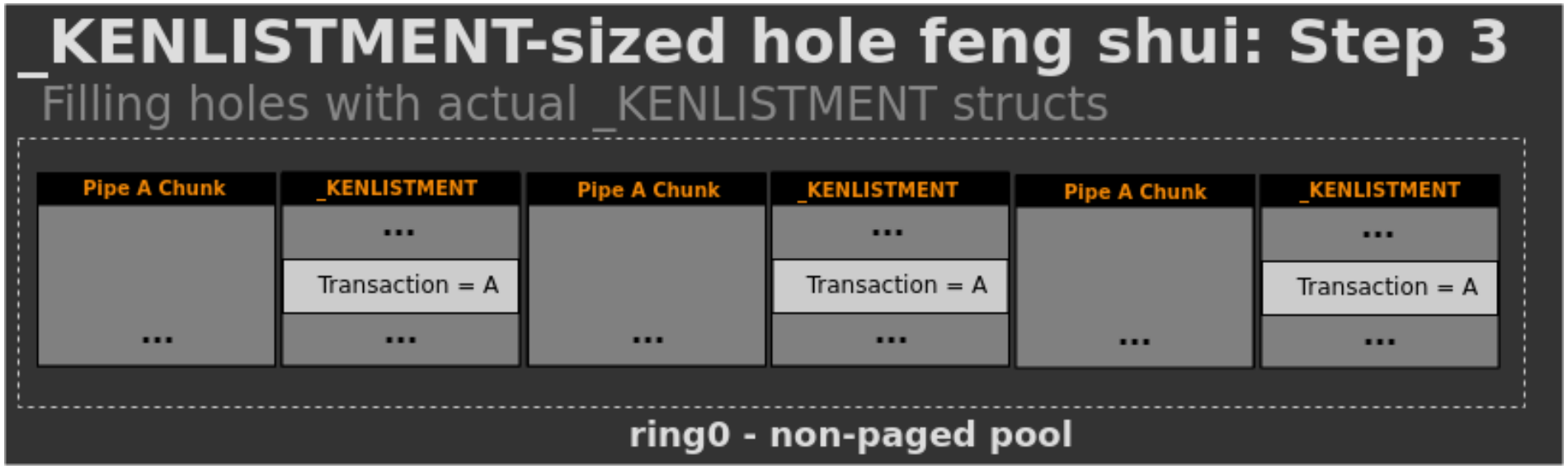
- Named Pipe writes allocate on non-paged pool
  - Code handled by `npfs.sys` (tracked by `NpFr` pool tag)
  - Persistent until other end of pipe reads data
  - Chunk free occurs when data is read
  - Size of chunk is fully controlled
  - All data of chunk aside from `DATA_ENTRY` is fully controlled
    - Data prefixed with an undocumented `DATA_ENTRY` structure (changed among Windows versions)
    - ReactOS is best starting point (reversing/hexdump for relevant changes)

Feng shui layout #1

- As usual, want to avoid coalescing causing big holes
- Writes on alternate named pipes

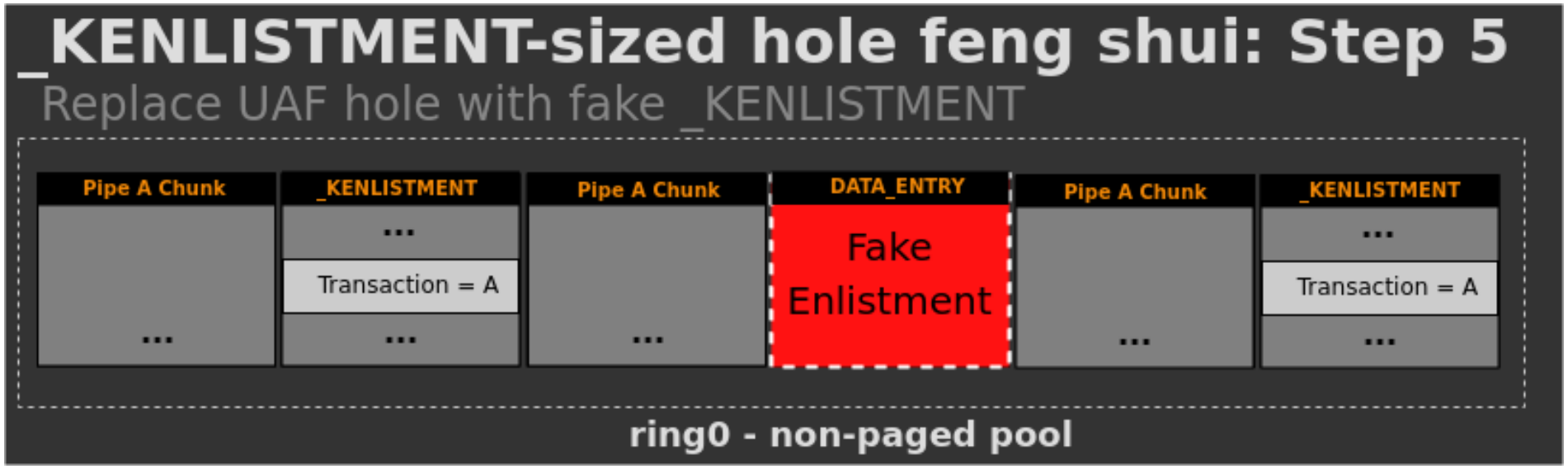
Feng shui layout #2

Feng shui layout #3

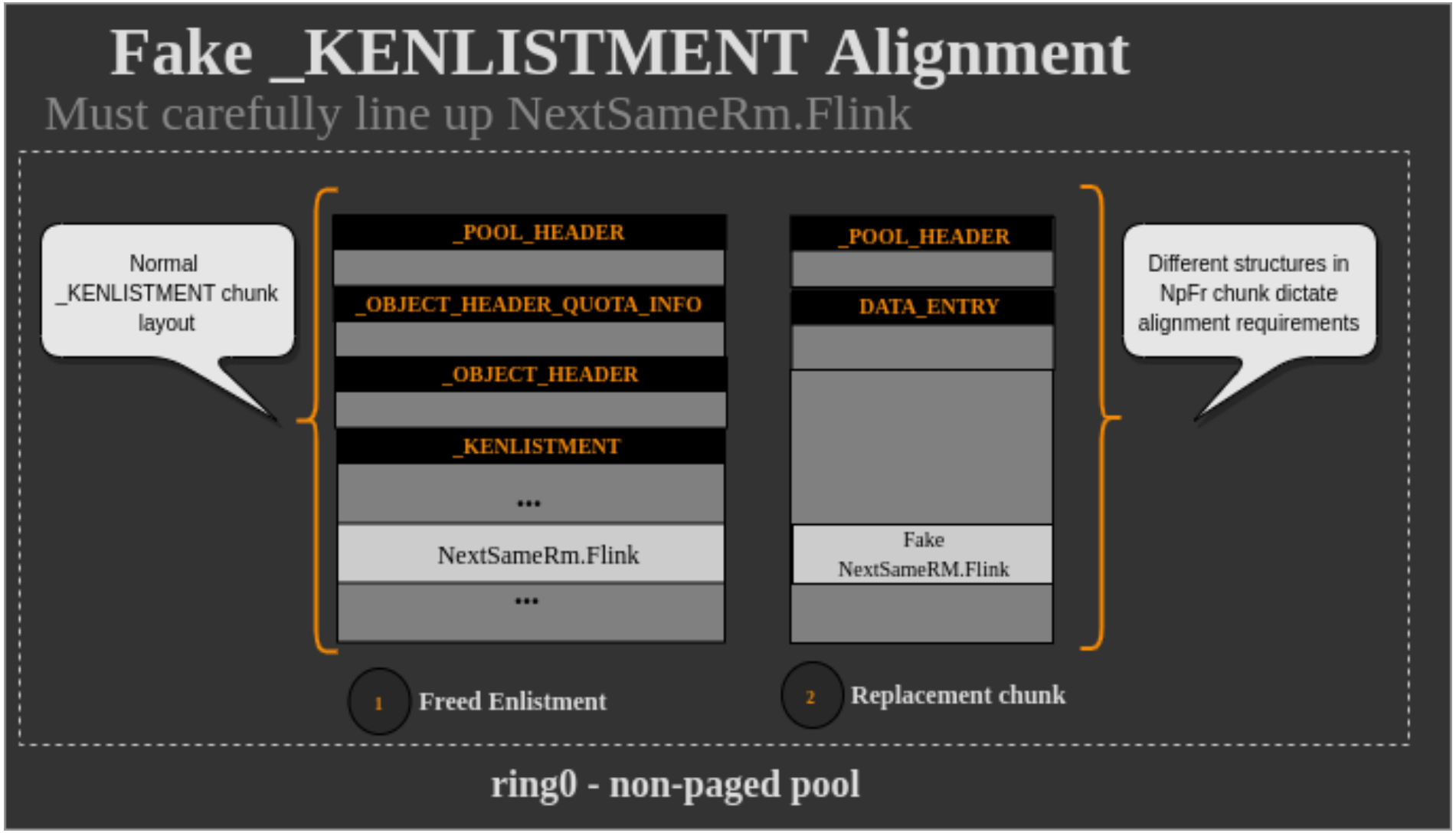


Feng shui layout #4

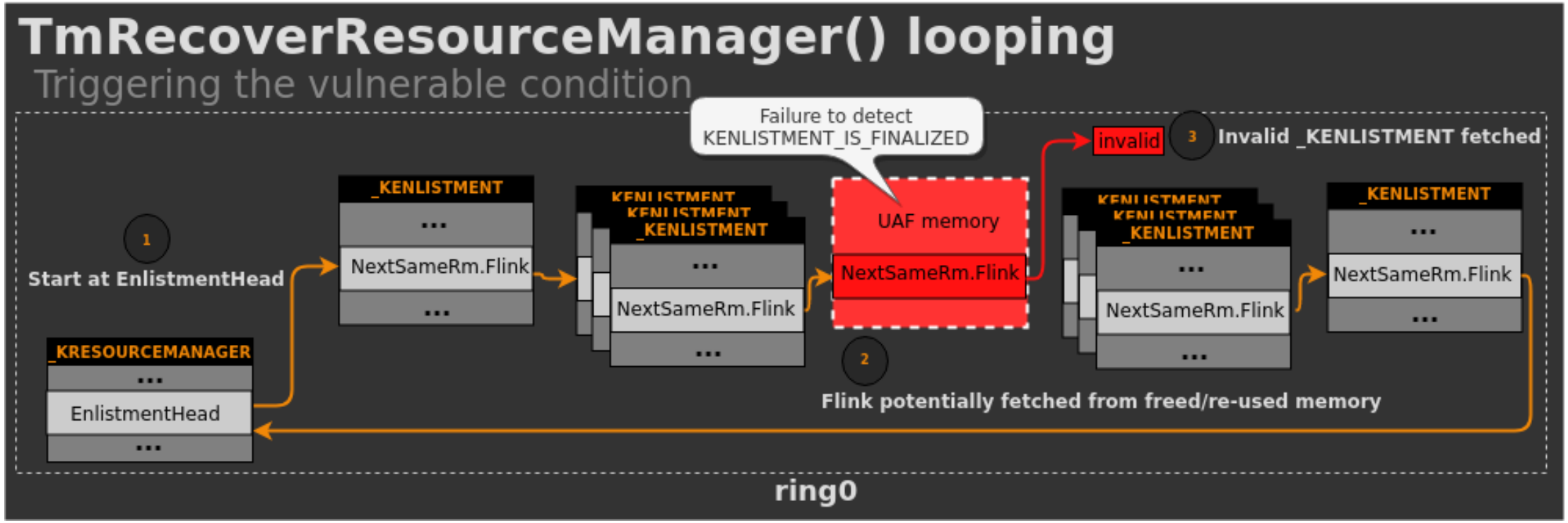
Feng shui layout #5



Faking a _KENLISTMENT with a named pipe chunk



End result


TmRecoverResourceManager() looping
Triggering the vulnerable condition
Failure to detect
KENLISTMENT_IS_FINALIZED
invalid
3
Invalid _KENLISTMENT fetched
_KENLISTMENT
...
NextSameRm.Flink
...
1
Start at EnlistmentHead
_KRESOURCEMANAGER
...
EnlistmentHead
...
UAF memory
NextSameRm.Flink
2
Flink potentially fetched from freed/re-used memory
ring0

What about delayed free list?

- Allocator optimisation to free chunks when the delayed free list is full only
  - Since Windows Vista
  - Waits for it to contain 32 chunks
  - Delayed free list is flushed and all 32 chunks are actually freed

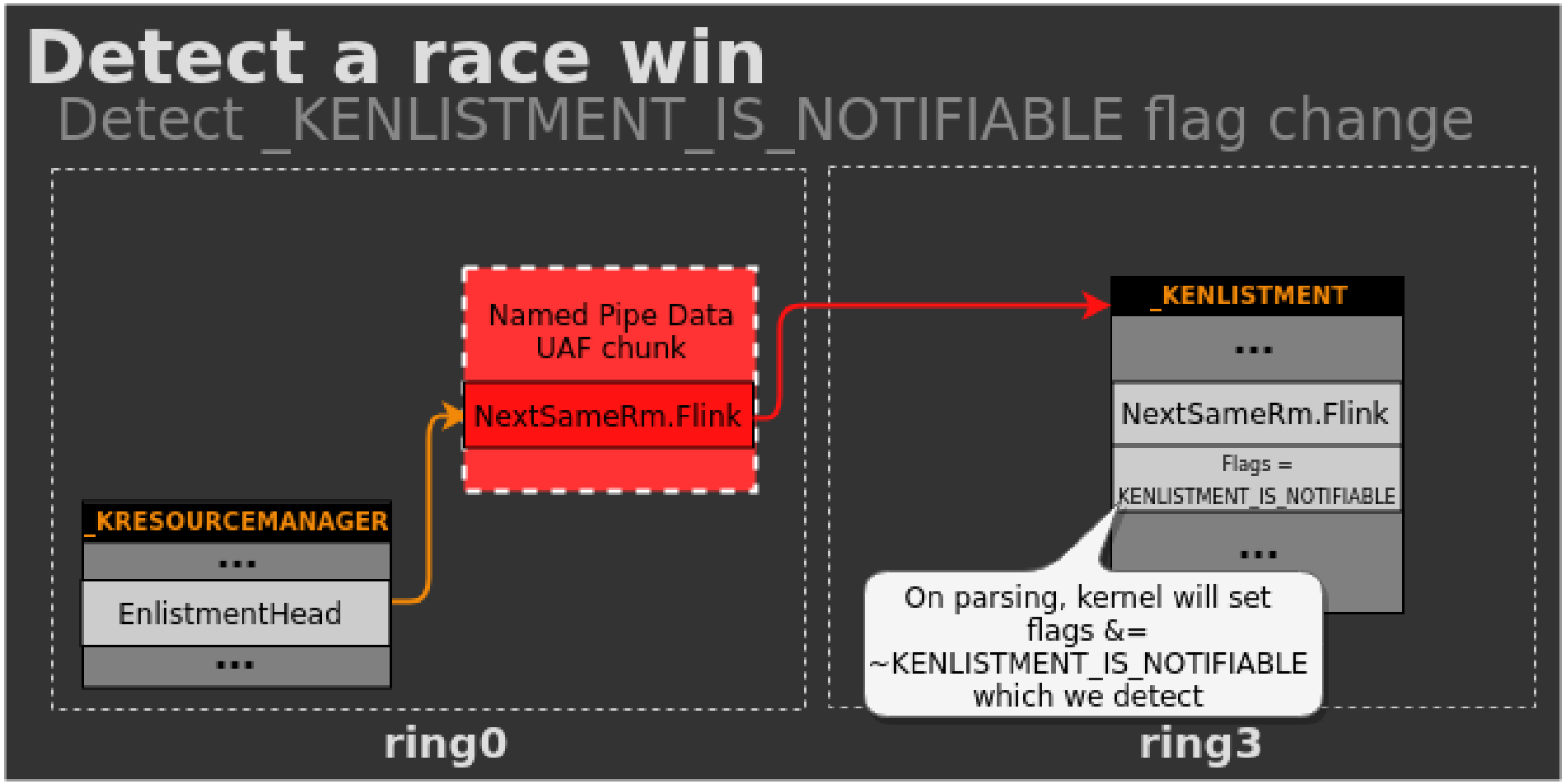
What about delayed free list?

- Turns out it does not matter for us since
  - If we win the race but UAF enlistment chunk is not effectively freed
    - Chunk content will not be replaced by anything else so `NextSameRm.Flink` used
    - `TmRecoverResourceManager` loop will just continue with following enlistment in list
  - We can win the race again later
- Can still improve triggering the UAF fast by just freeing 32 chunks to bypass delayed free

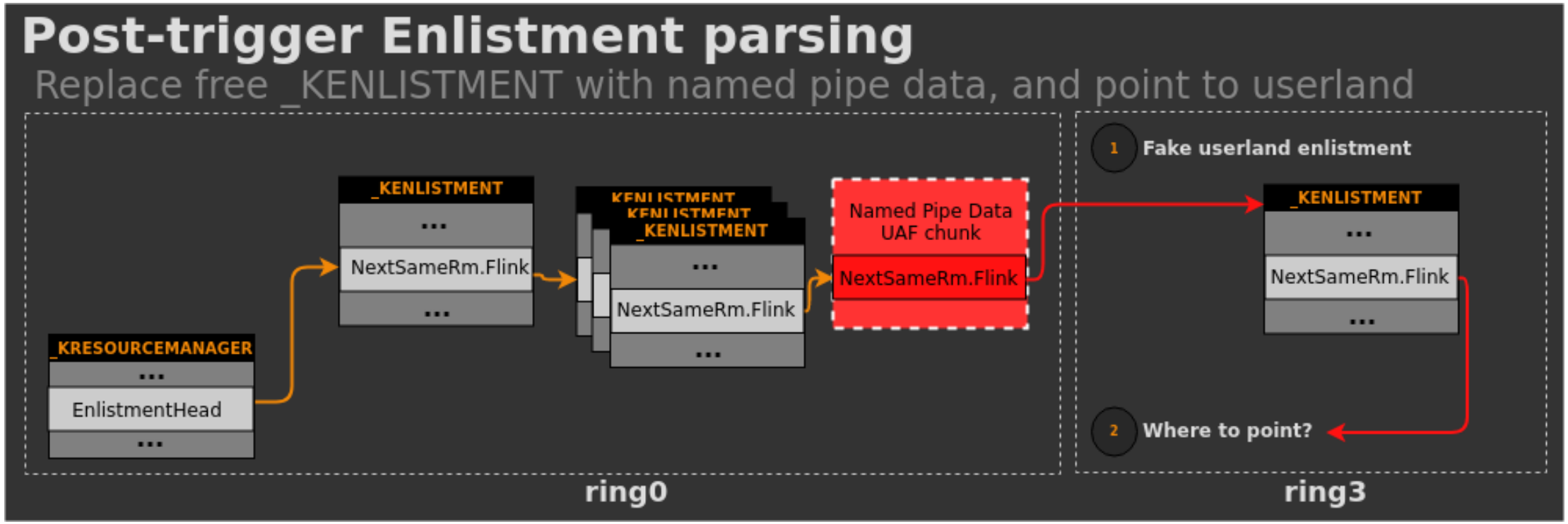
Detecting a race win

- How seize control of loop?
- No SMAP on Windows!

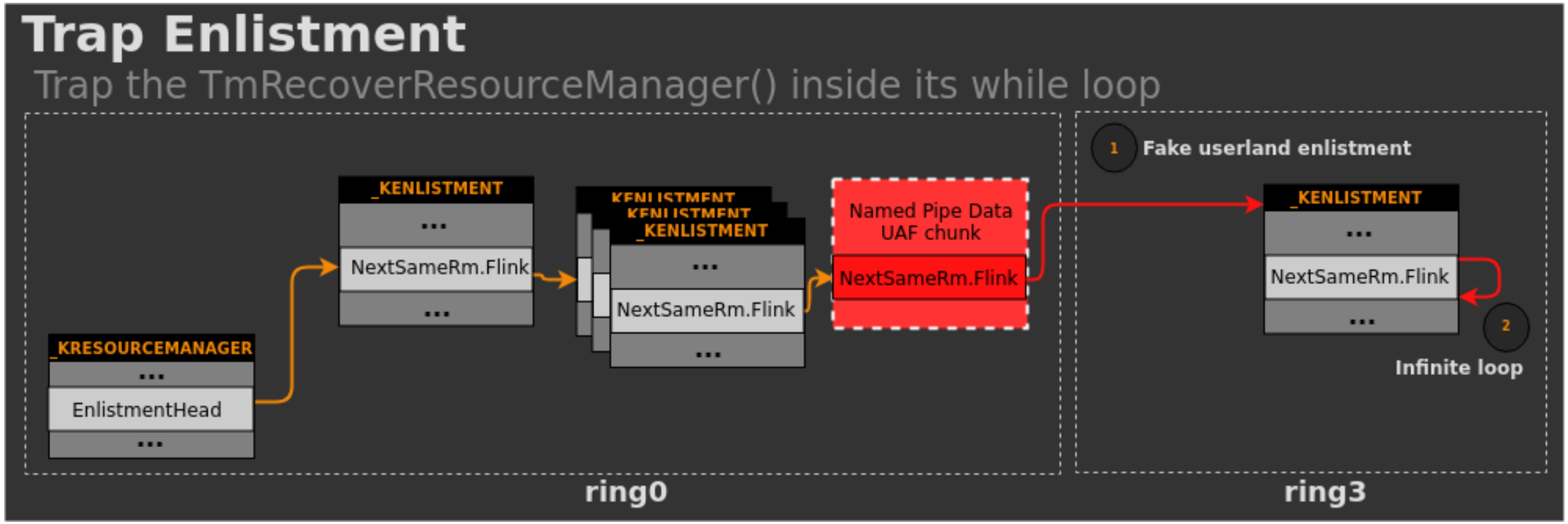
Detecting a race win



Now what?


Post-trigger Enlistment parsing
Replace free _KENLISTMENT with named pipe data, and point to userland
_KRESOURCEMANAGER
...
EnlistmentHead
...
_KENLISTMENT
...
NextSameRm.Flink
...
_KENLISTMENT
_KENLISTMENT
_KENLISTMENT
...
NextSameRm.Flink
...
Named Pipe Data
UAF chunk
NextSameRm.Flink
1
Fake userland enlistment
_KENLISTMENT
...
NextSameRm.Flink
...
2
Where to point?
ring0
ring3

Trap enlistment

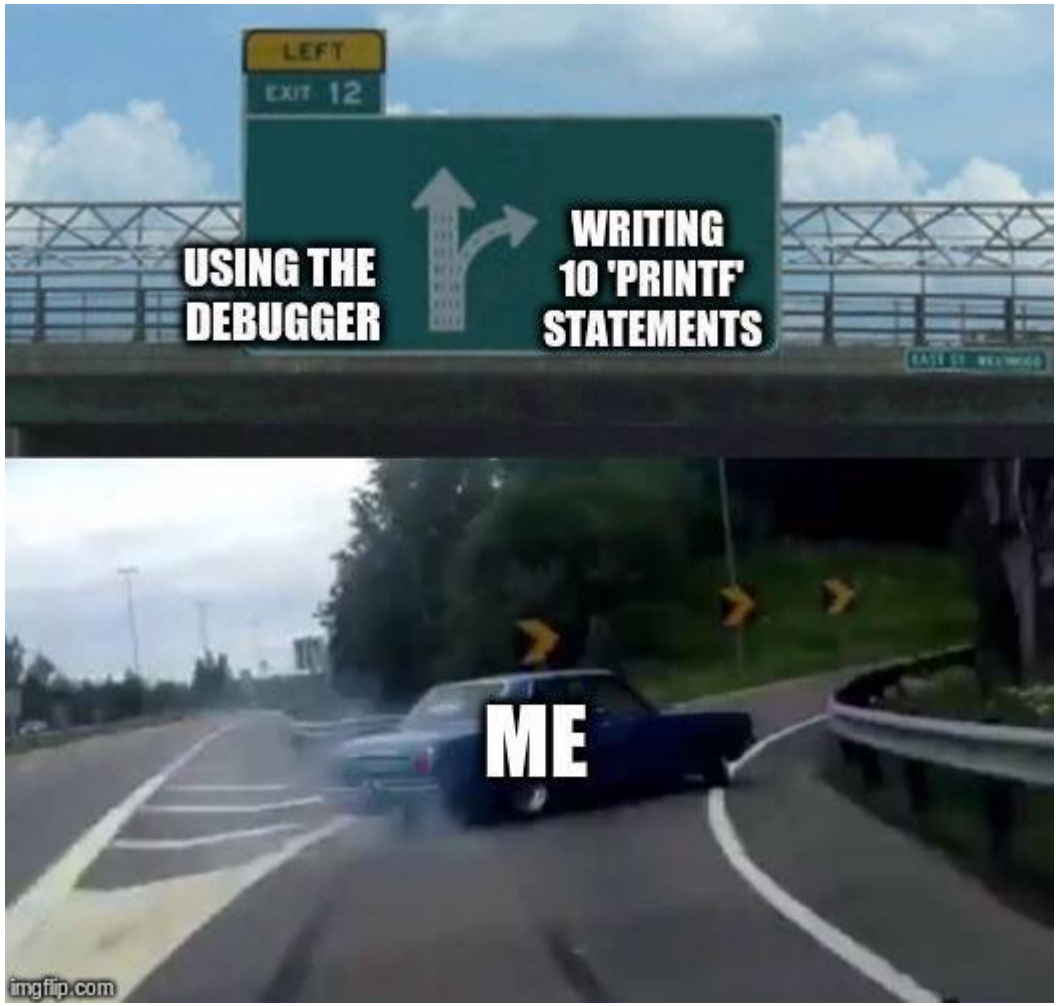


- Inject list of new enlistments into `Flink` when ready
- Tail of new list of enlistments can be another trap

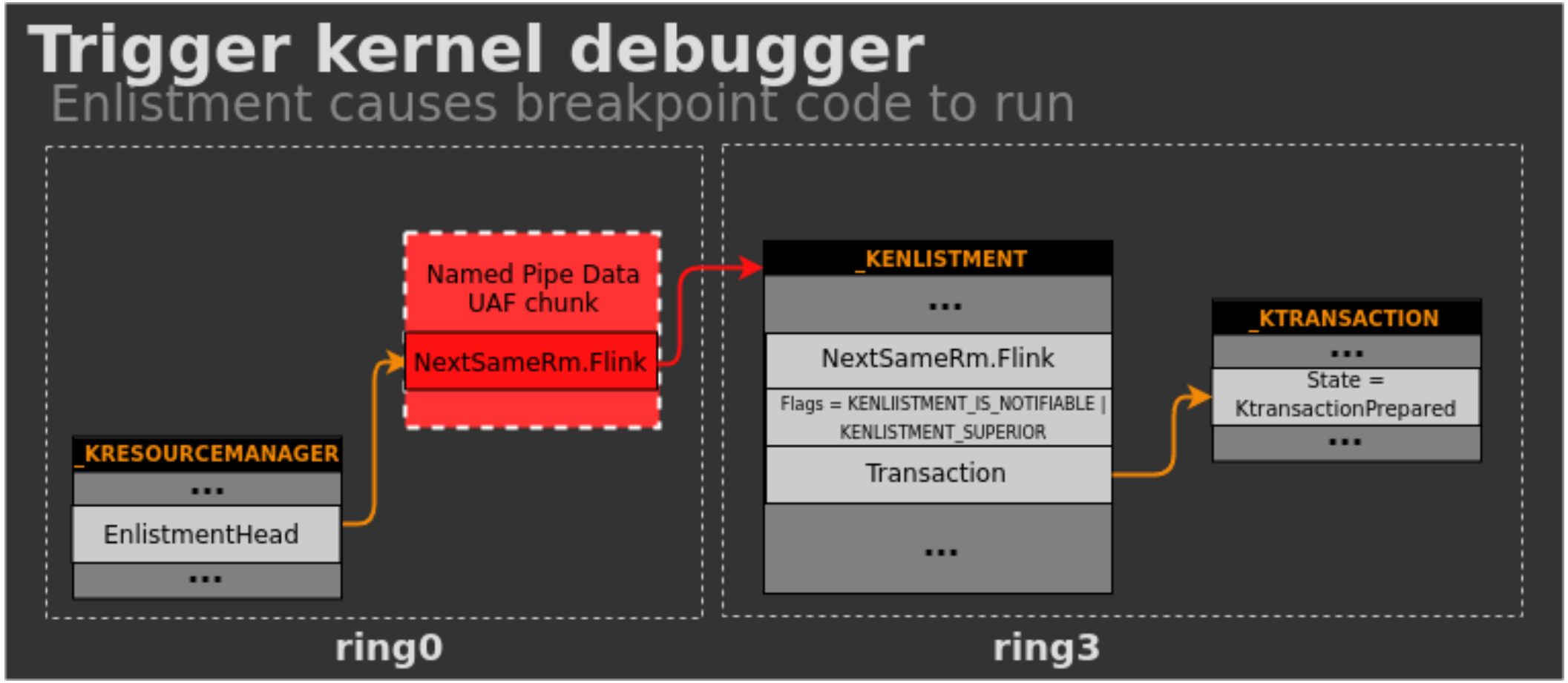
Detecting a race win - key points

- No SMAP on Windows!
- Replacement `_KENLISTMENT->NextSameRM` points to yet another fake userland `_KENLISTMENT`
- Userland `_KENLISTMENT->NextSameRM` points to itself
- We refer to this as a 'trap' enlistment
- Kernel is now temporarily stuck in an infinite loop
- Kernel unsets `KENLISTMENT_IS_NOTIFIABLE` flag on userland enlistment
  - This modification in userland tells us we won!

Debugging a race win?



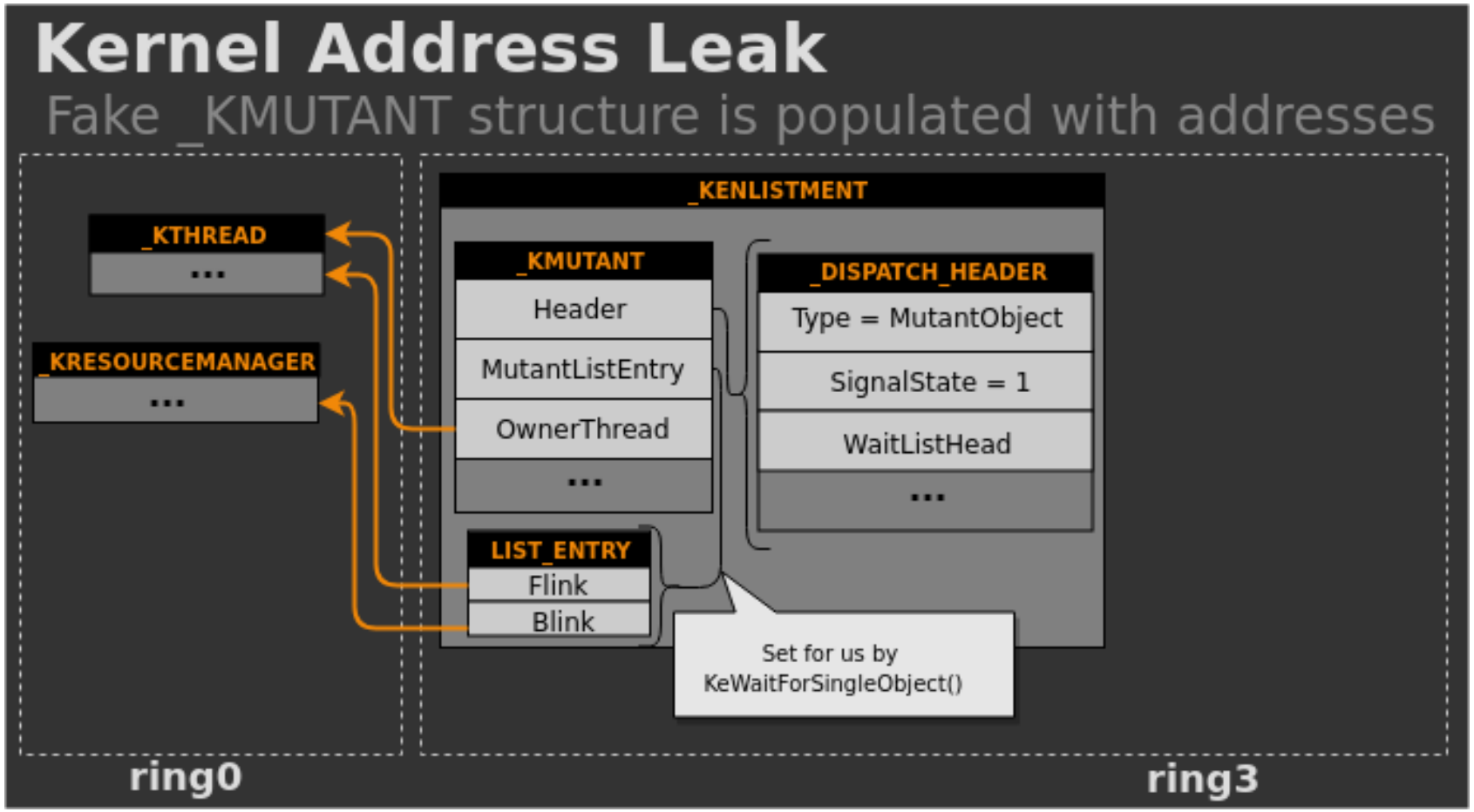
Debugging a race win?



How to escape the loop?

- We have control of the loop now
- We need a write primitive of some kind
- But also need to escape the loop?

Initial kernel pointer leak

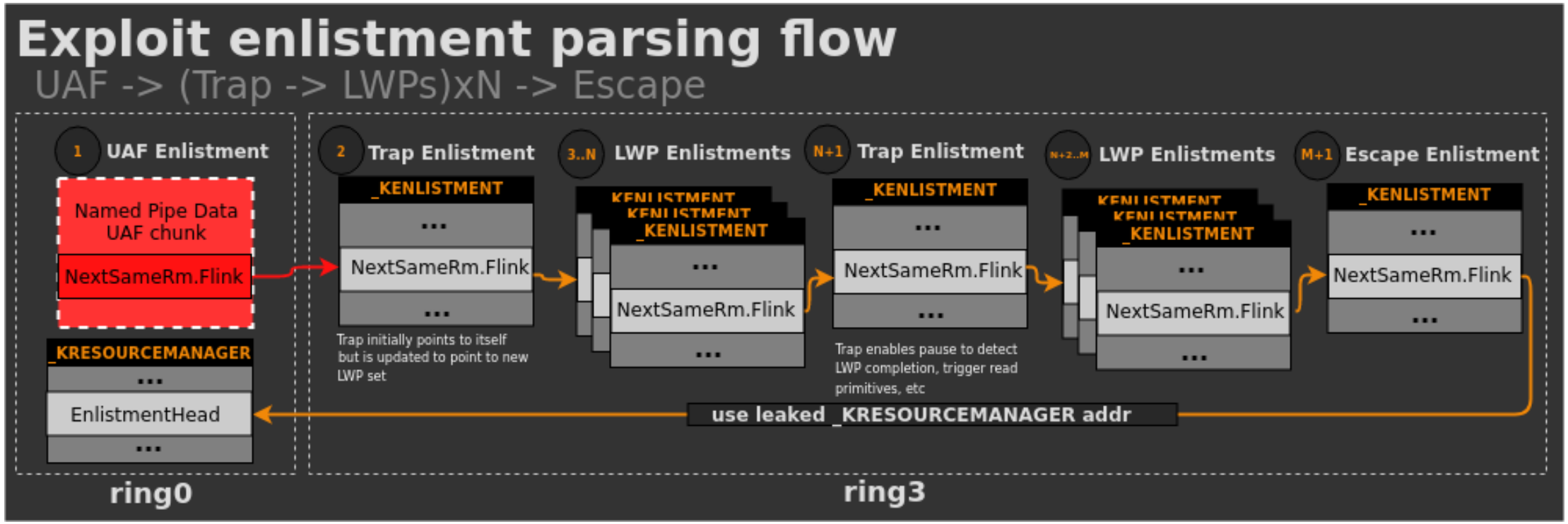


- Thank you `KeWaitForSingleObject()`

Escaping the loop

- We can now exit the loop!
- Introduce an 'escape' enlistment
- Set `KENLISTMENT->NextSameRm = &_KRESOURCEMANAGER.EnlistmentHead`
- Exit cleanly
- No crashes.. reproducable testing, etc.

What an escape looks like



- LWP = Limited Write Primitive (explained soon)

Building a write primitive

Vulnerable loop constraints

- Finding a write primitive is somewhat limited
- We are stuck inside this recovery loop
- What code paths do we follow?
- KeReleaseMutex() seemed best
  - List-based mirror-write primitives are safe unlinked after Windows 7 :(
  - Keep looking...
- Found an arbitrary increment inside `KiTryUnwaitThread()` call

```
if ( (OwnerThread->WaitRegister.Flags & 3) == 1 ) {
  ThreadQueue = OwnerThread->Queue;
  if ( ThreadQueue )
    _InterlockedAdd(&ThreadQueue->CurrentCount, 1u);
```

- But things get complicated..

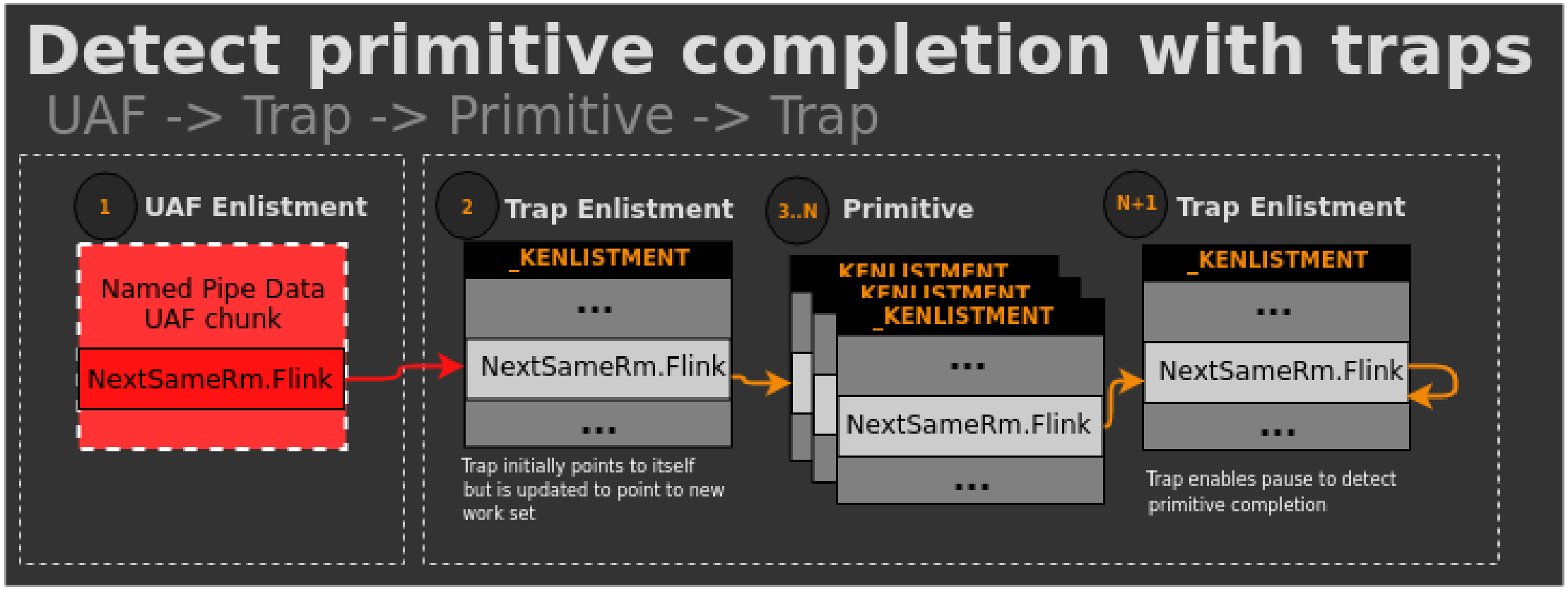
Arbitrary increment primitive

- `KeReleaseMutex()` - `KeReleaseMutant()` wrapper
  - `KeReleaseMutant()` - Our high level primitive function
    - `KiTryUnwaitThread()` - Gives us our increment primitive
    - `KiProcessThreadWaitList()` - Unavoidable because of increment primitive
      - `KiUnlinkWaitBlocks()` - Have to satisfy its attempt to unlink
      - `KiReadyThread()` - Unavoidable call on our fake thread
        - `KiRequestProcessInSwap()` - Have to satisfy early exit

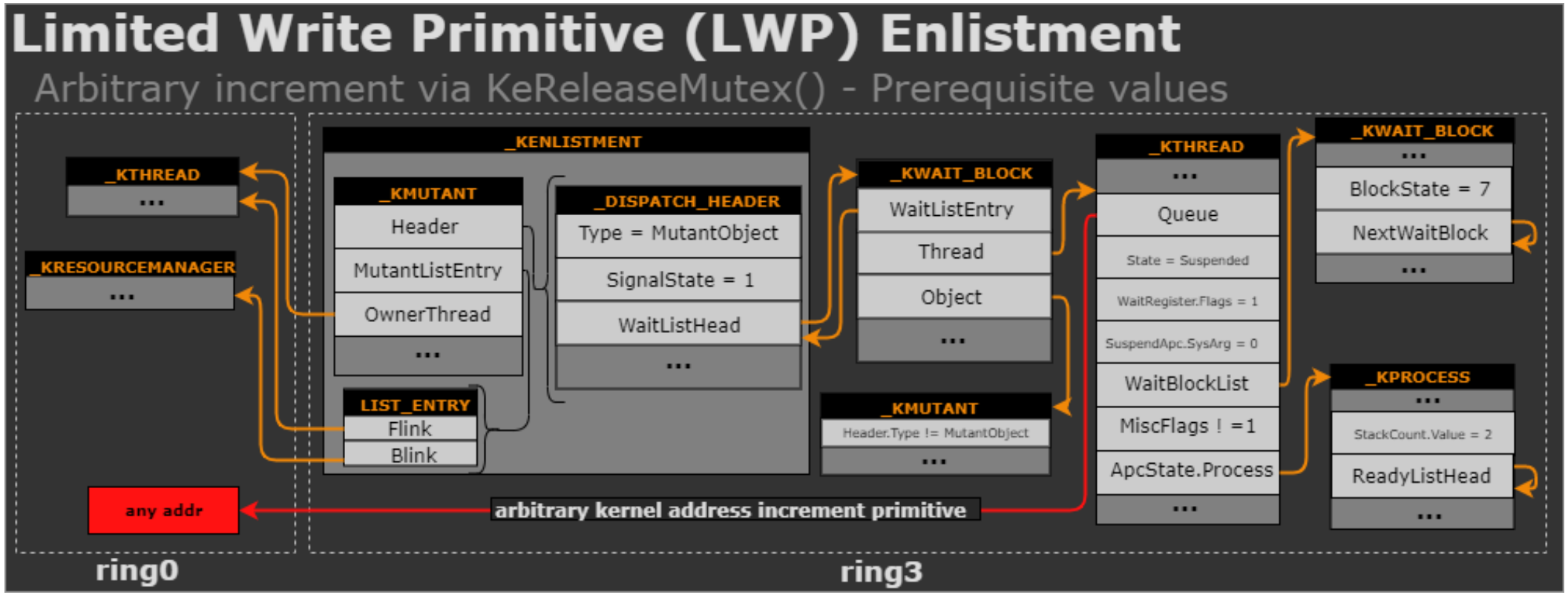
Repeatable arbitrary address increment

- Too complicated to explain in detail
- Follow up blog series covers line by line
- Positives
  - Can chain multiple increments together
  - Effectively an arbitrary write primitive
- Negatives
  - Need to know the starting contents of the address being written to
  - Some risks related to running at `DISPATCH_LEVEL`

Primitive injection at a glance



What does our increment primitive look like?



- Lots of constraints
- Some requirements change across OS versions

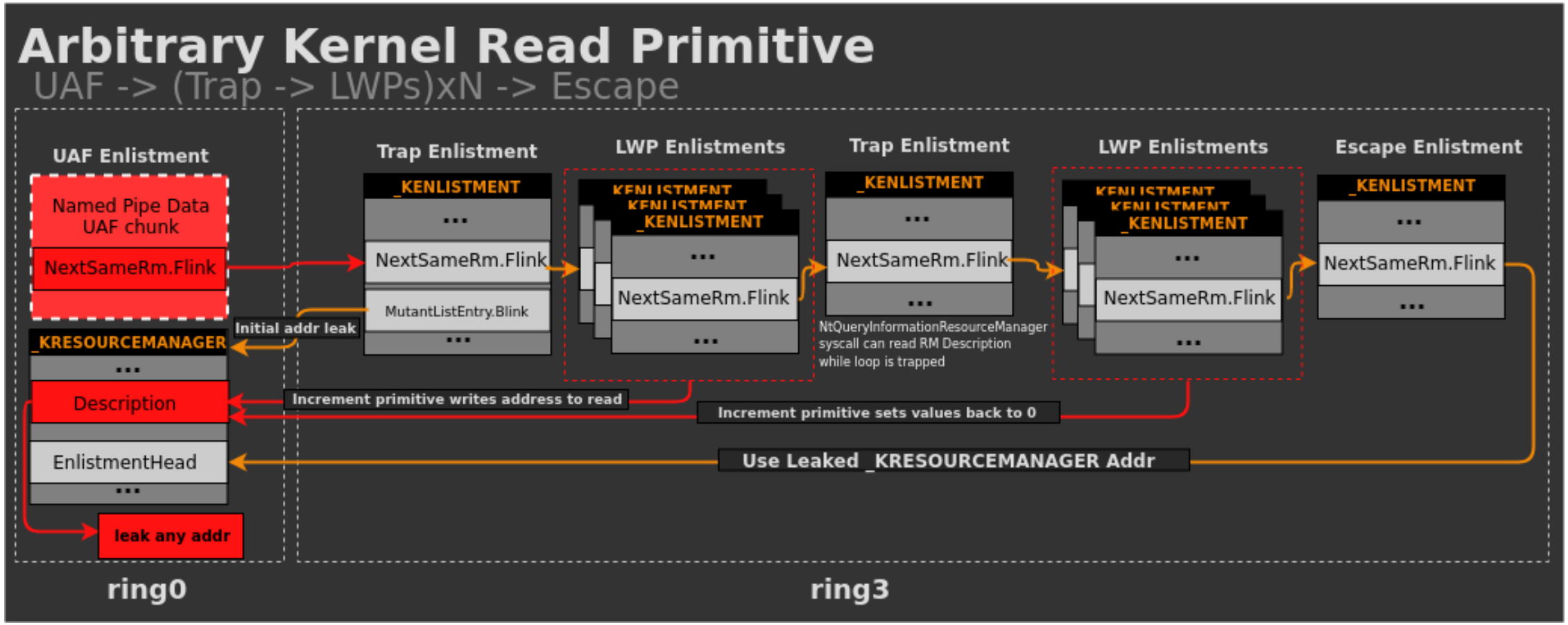
Me running the LWP primitive

Building an arbitrary kernel read primitive

What to do?

- We have an arbitrary write as long as we know original value
- We know where `_KRESOURCEMANAGER` is
- We can avoid setting a `Description` field when creating the RM
  - Means we know `_UNICODE_STRING` `Length` and `Name` are both zero
- Point anywhere we want
- Call `NtQueryInformationResourceManager` syscall to get description
- Rinse and repeat

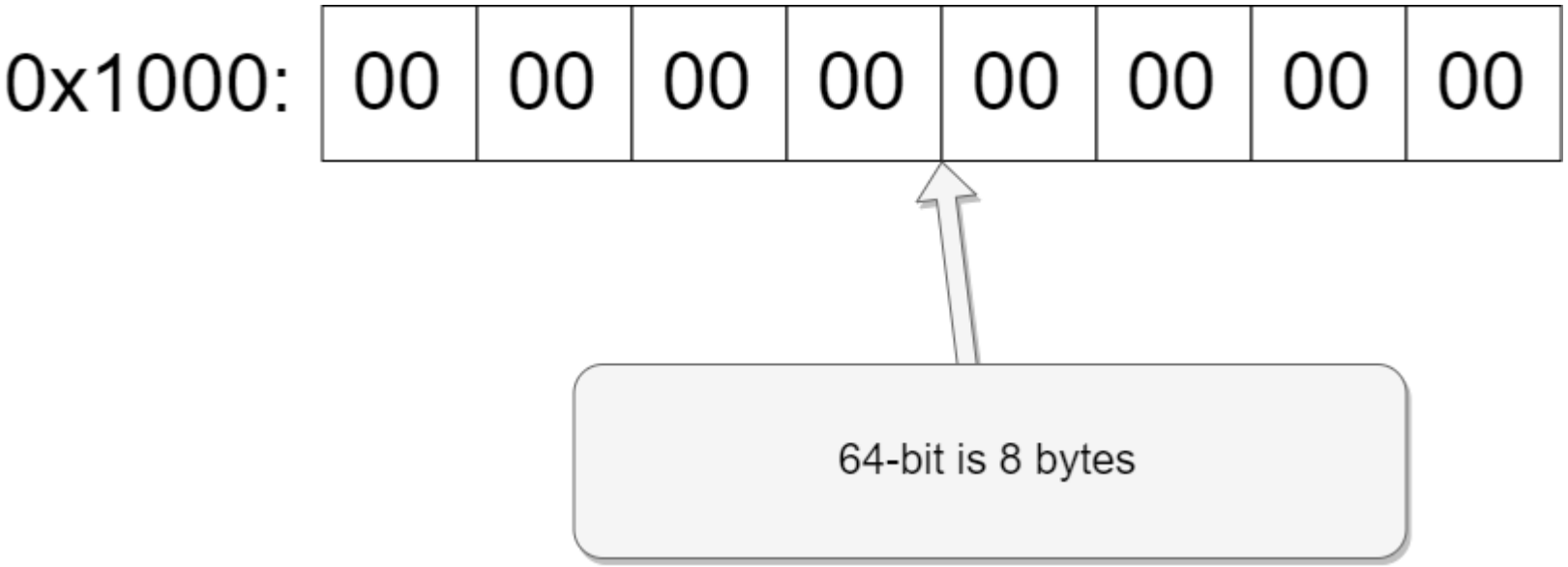
What does our read primitive look like?



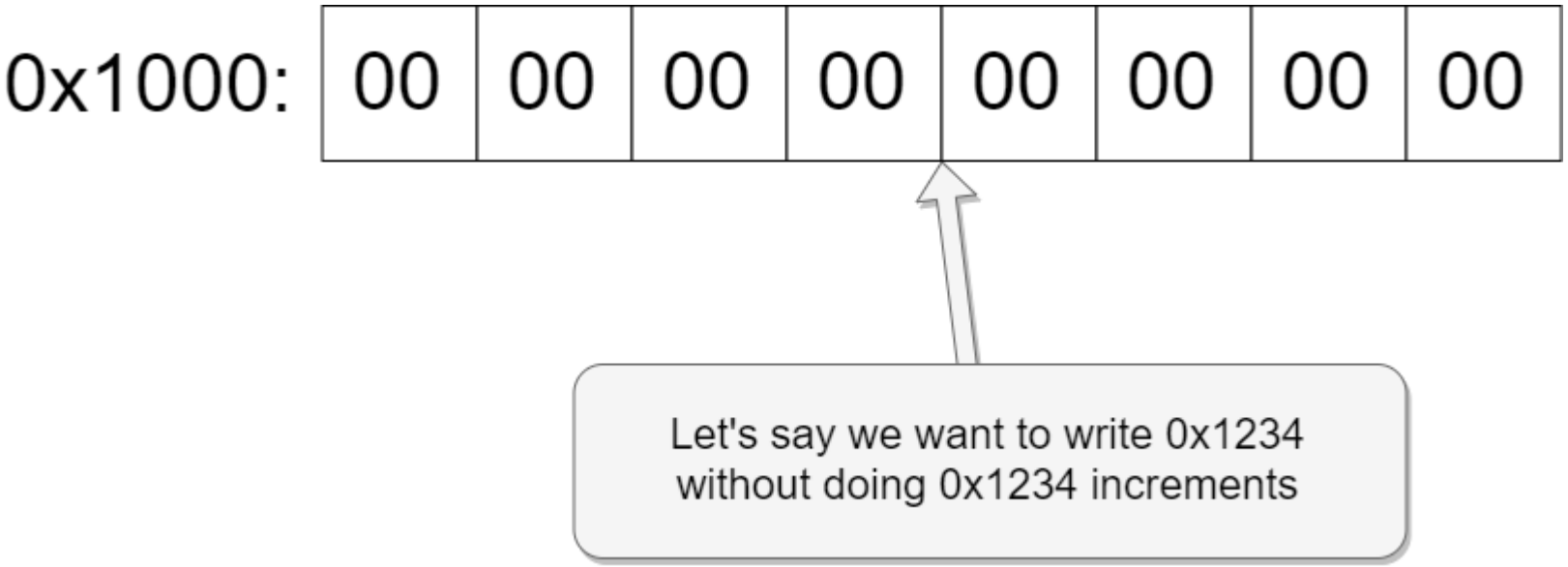
How is that possible in practice?

- What if we need to get a really big value?
- Or need to wrap around to get a smaller value?
- Are we going to need `2^64` increments?
  - Fortunately no, there is a trick (max `2*(256*8) = 2048` writes)
  - Split the increments across 8 different addresses
  - Each byte of the 64-bit value incremented 256 or less times

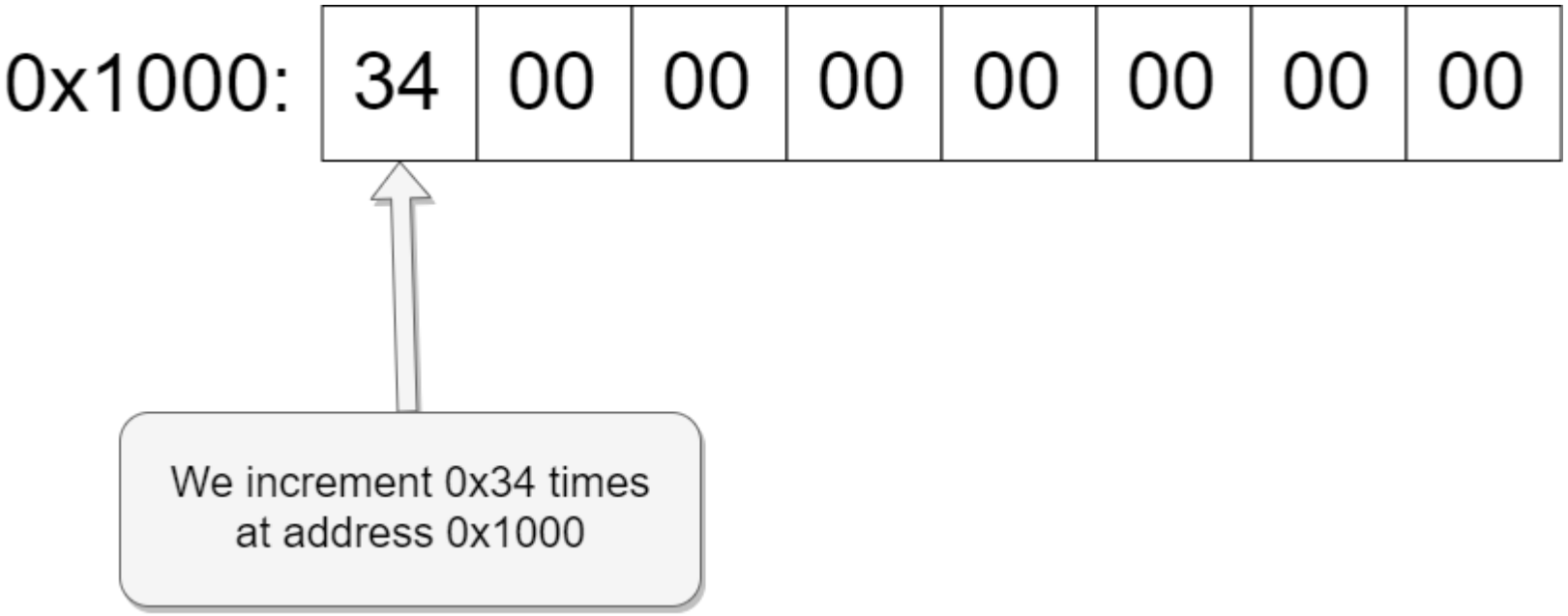
Efficient use of increment



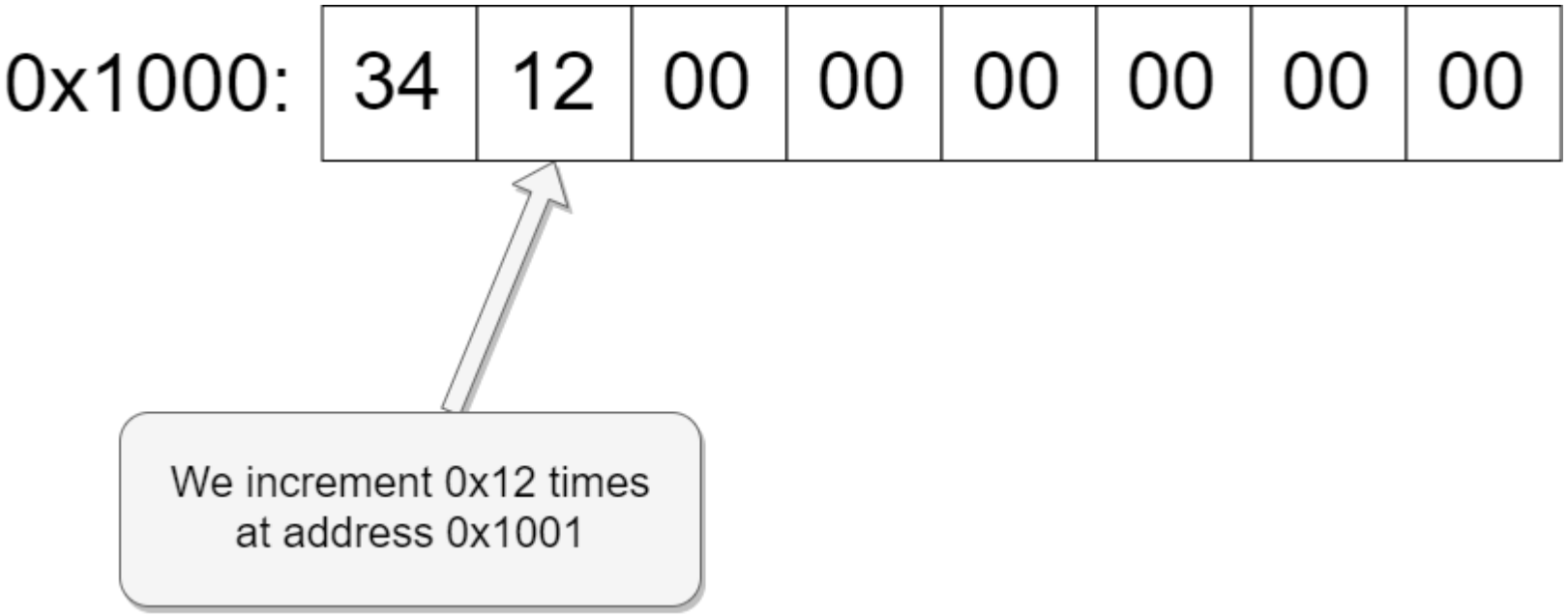
Efficient use of increment



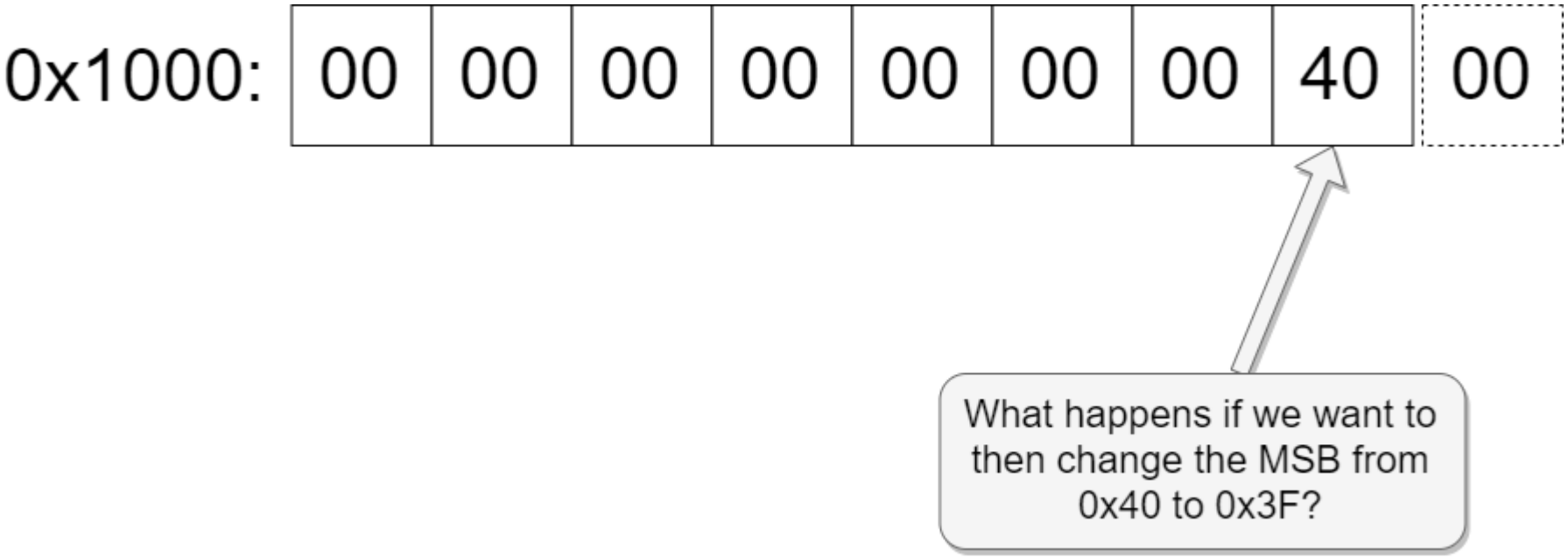
Efficient use of increment



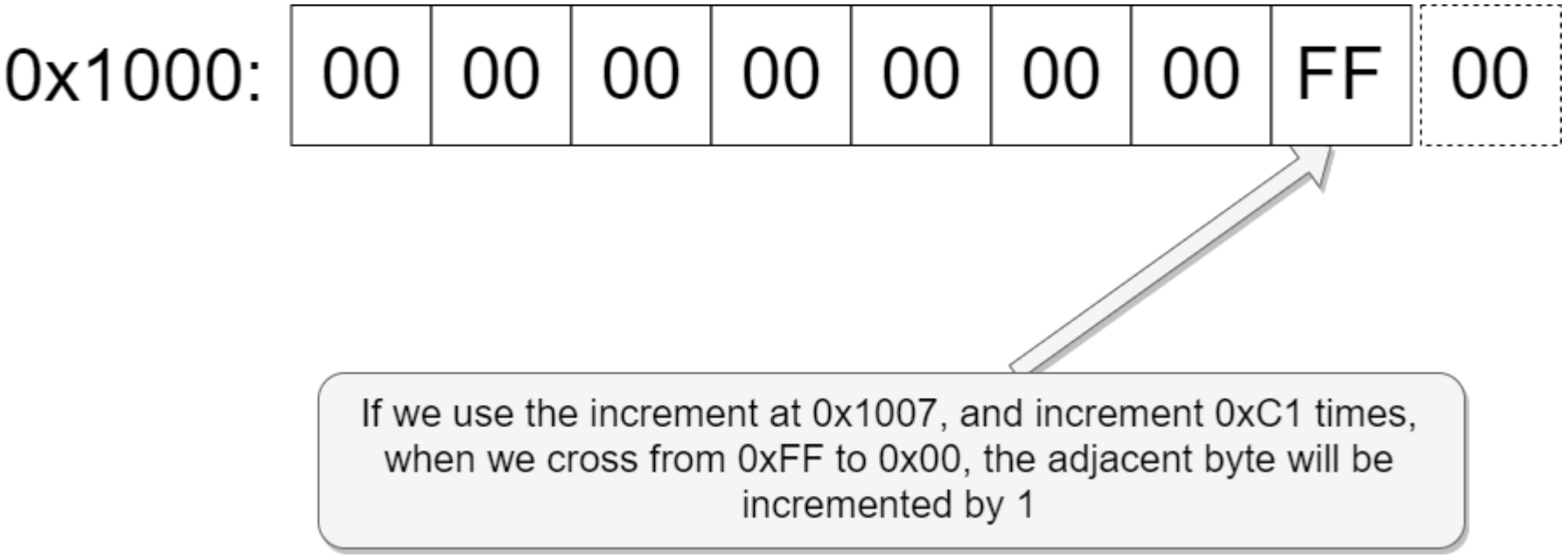
Efficient use of increment



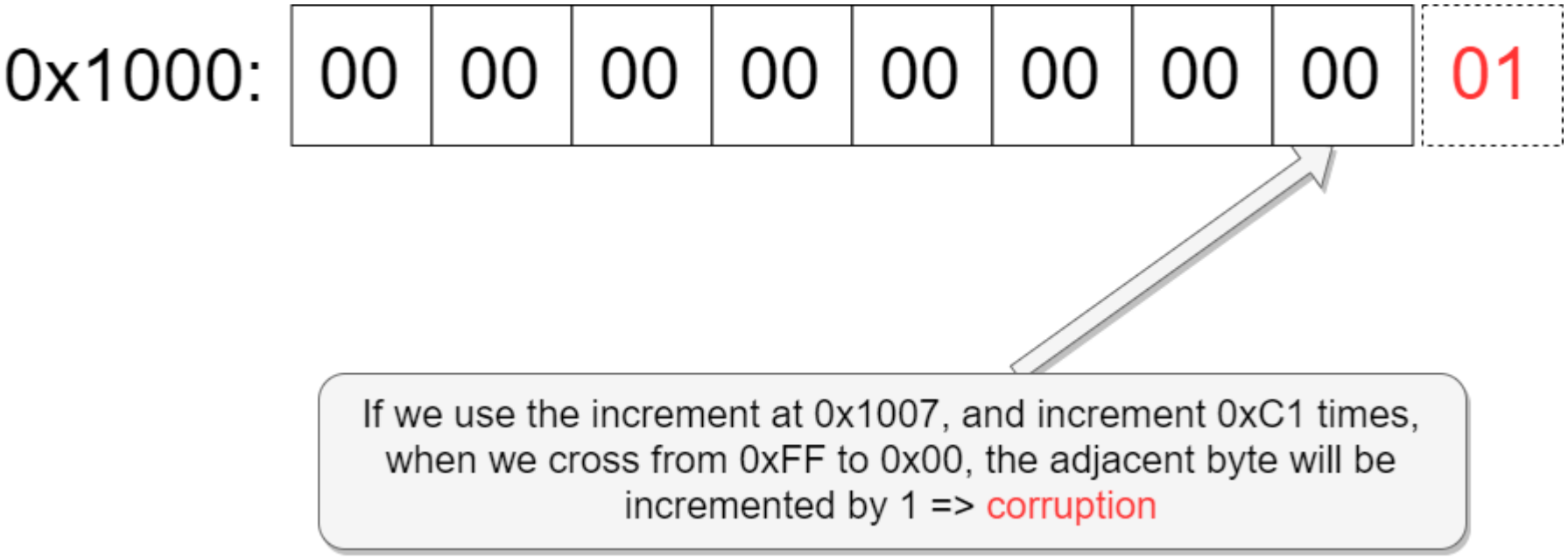
Efficient use of increment



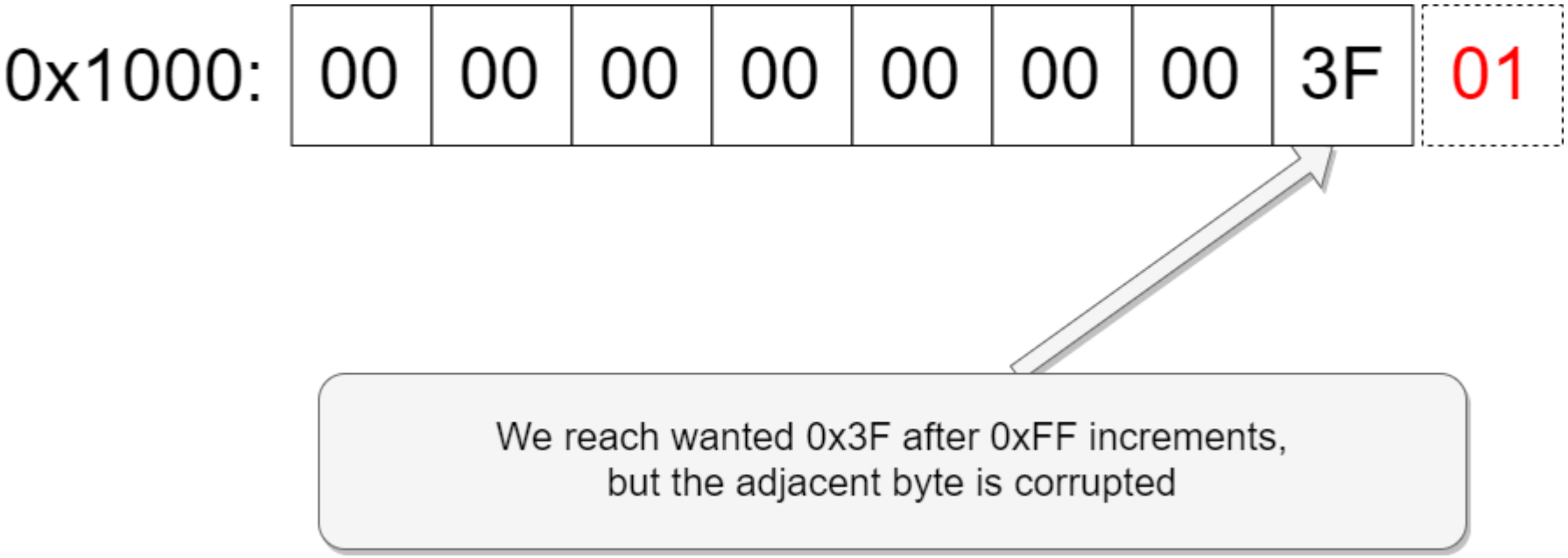
Efficient use of increment



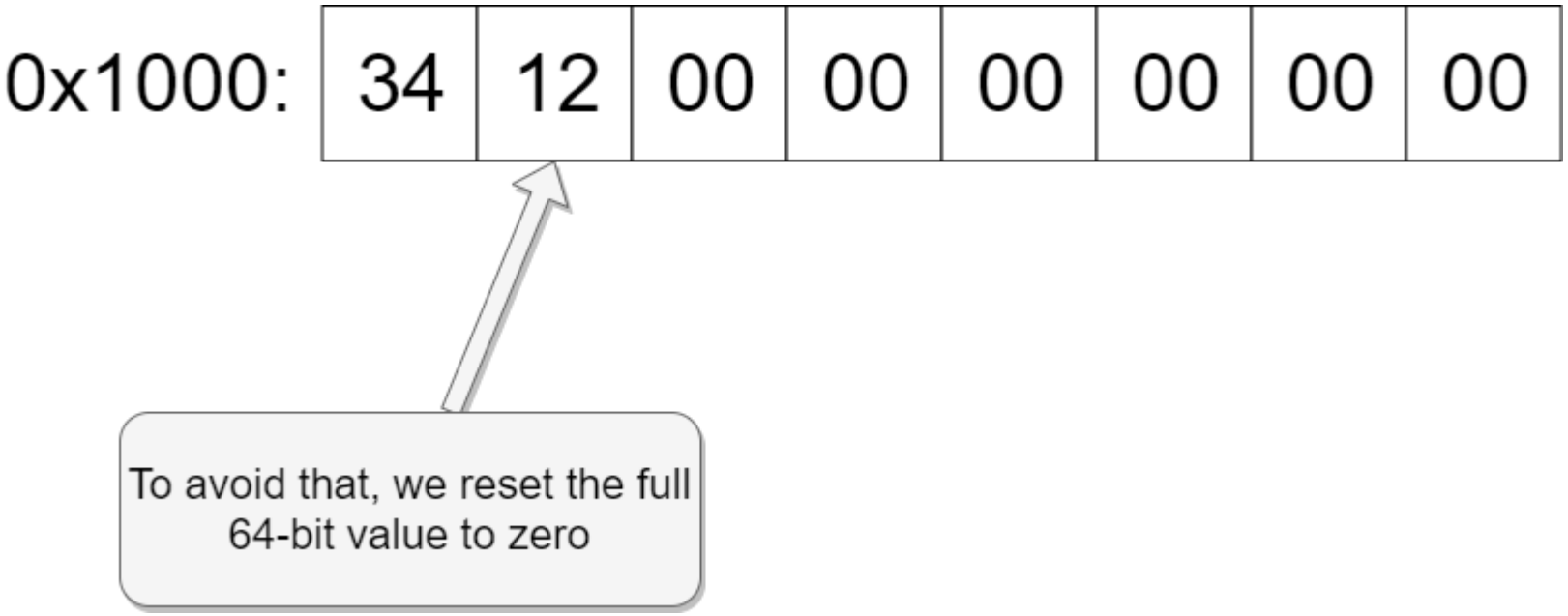
Efficient use of increment



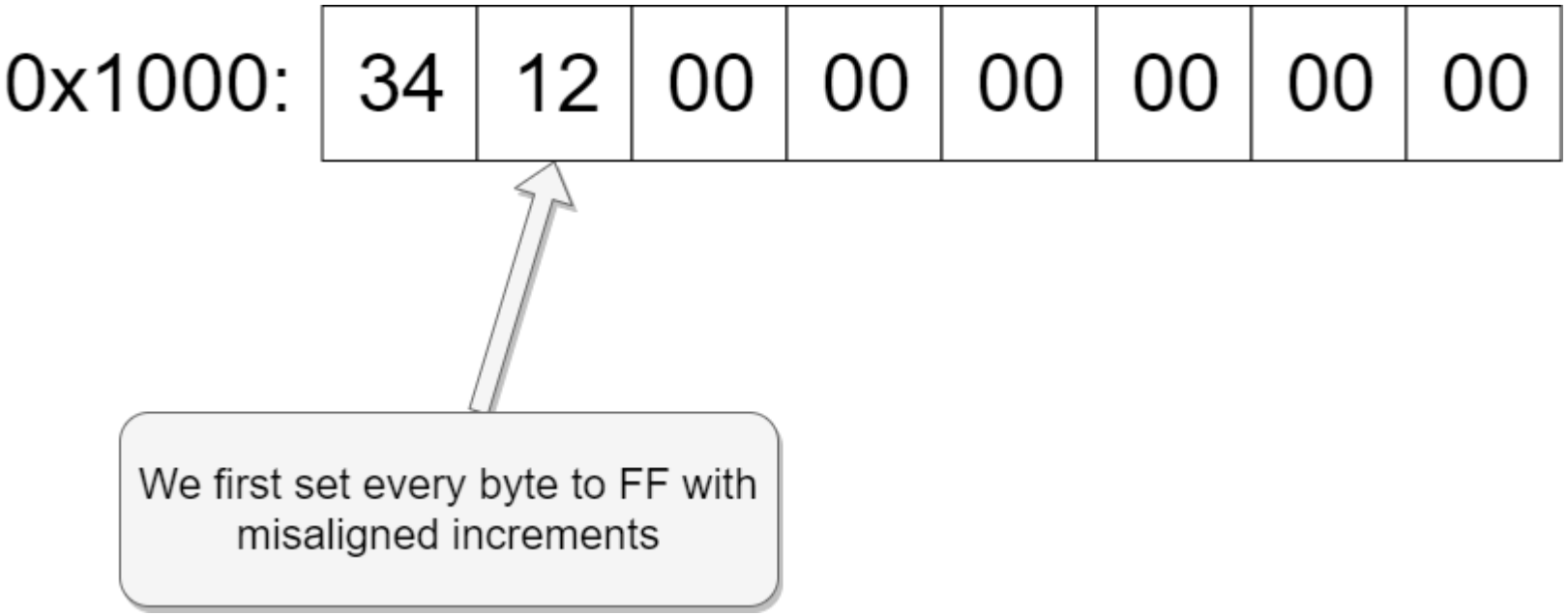
Efficient use of increment



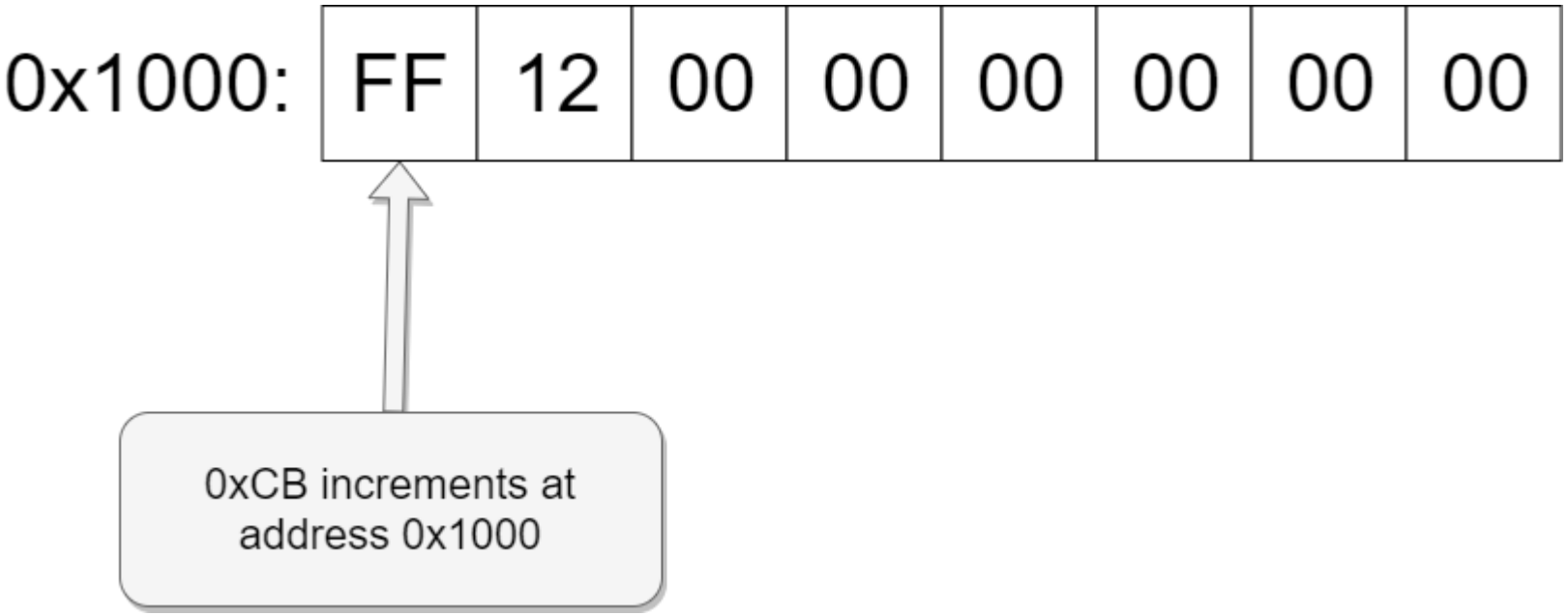
Efficient use of increment



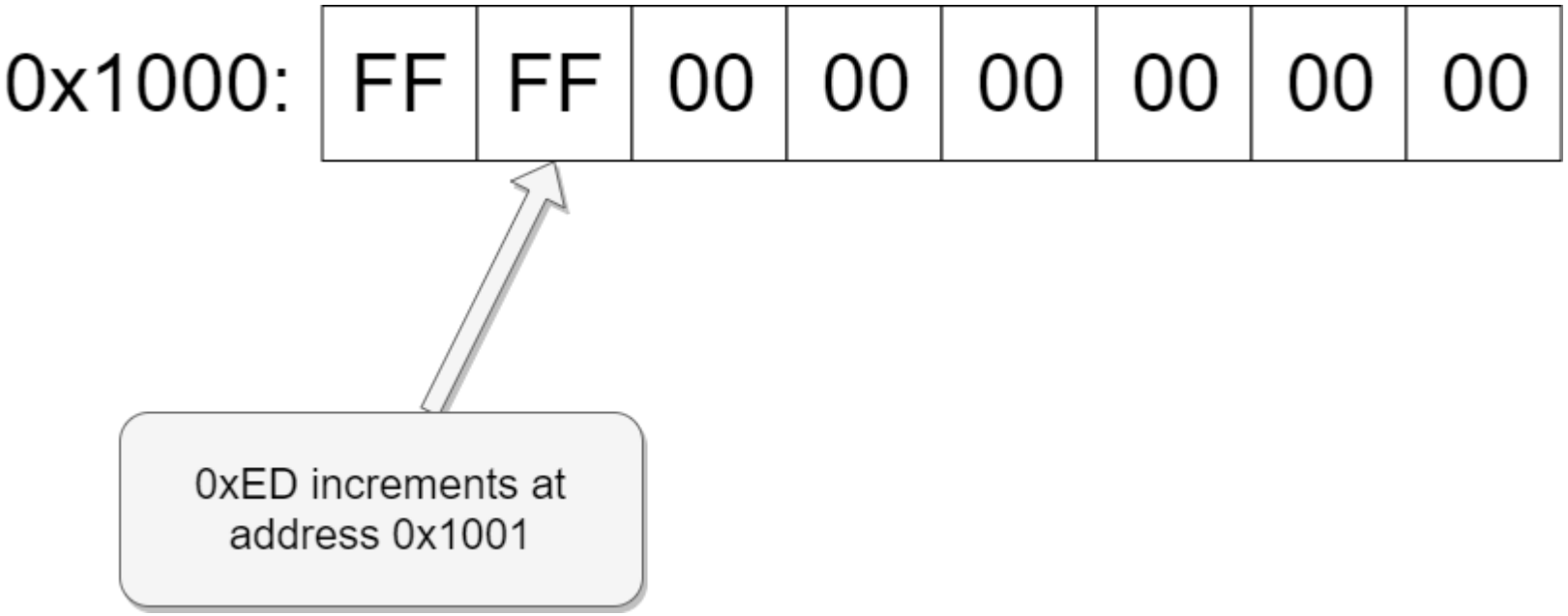
Efficient use of increment



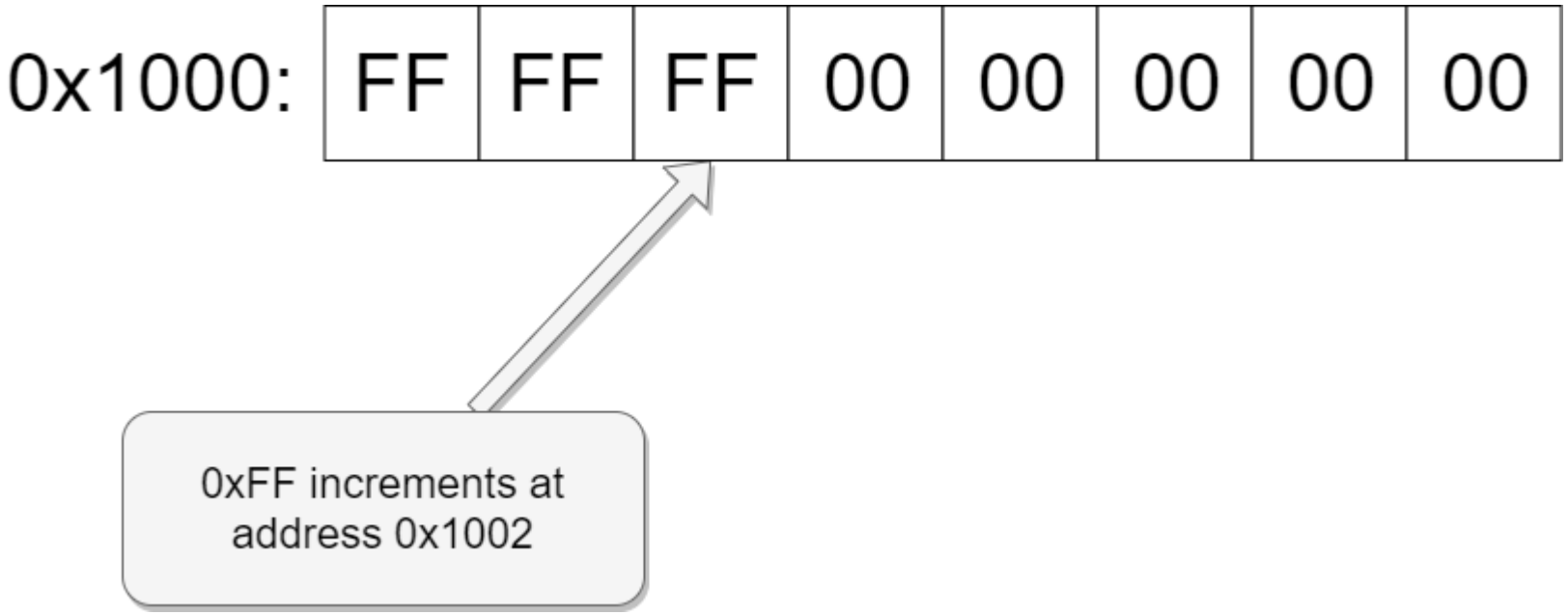
Efficient use of increment



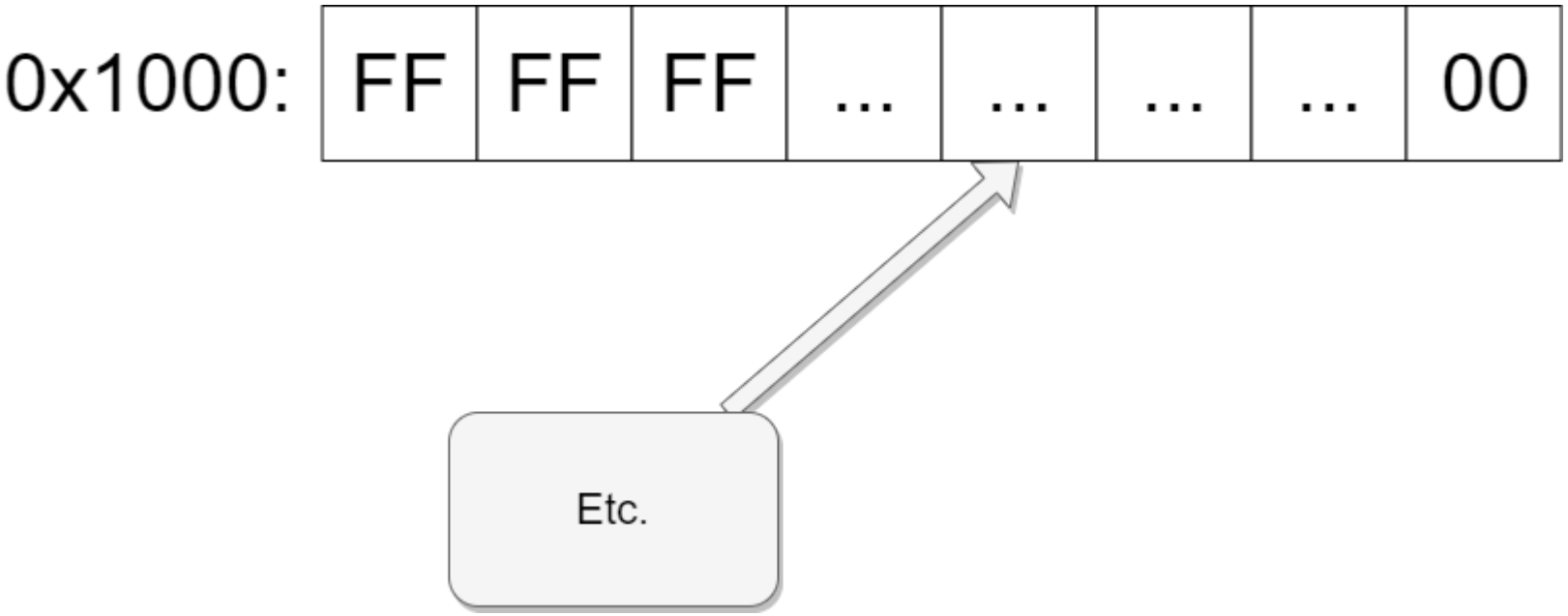
Efficient use of increment



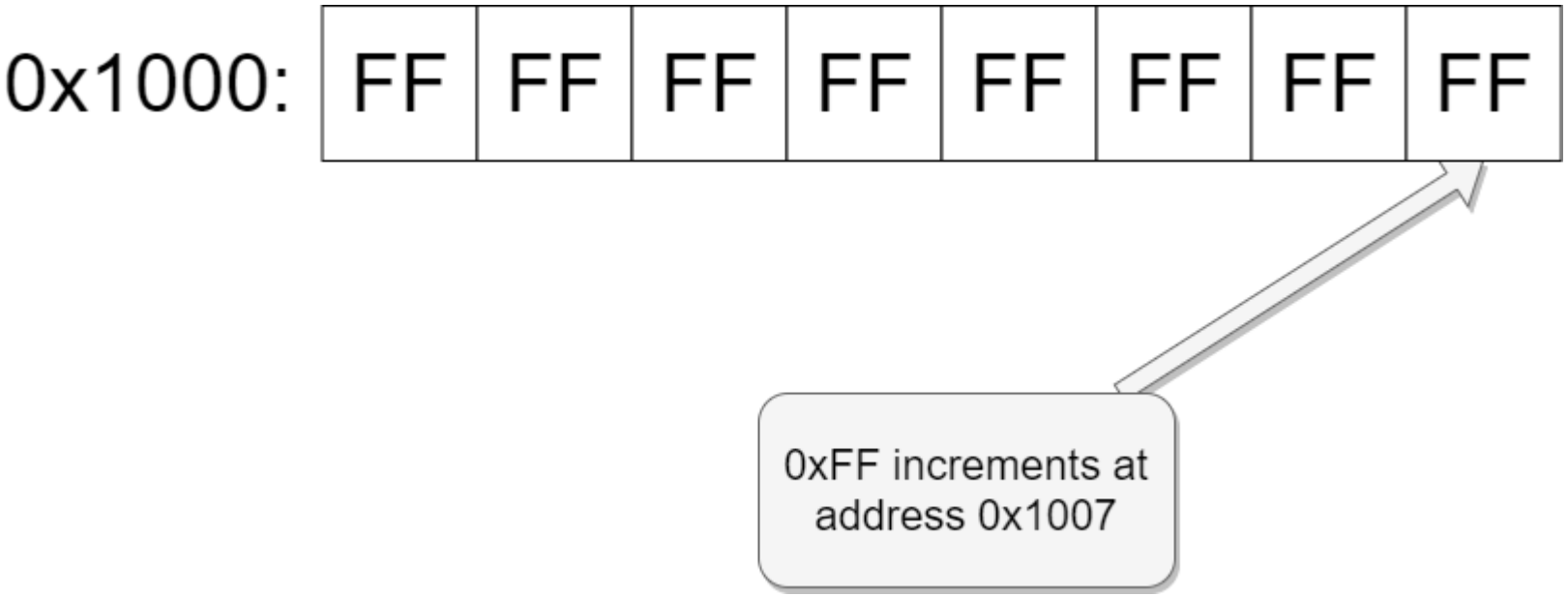
Efficient use of increment



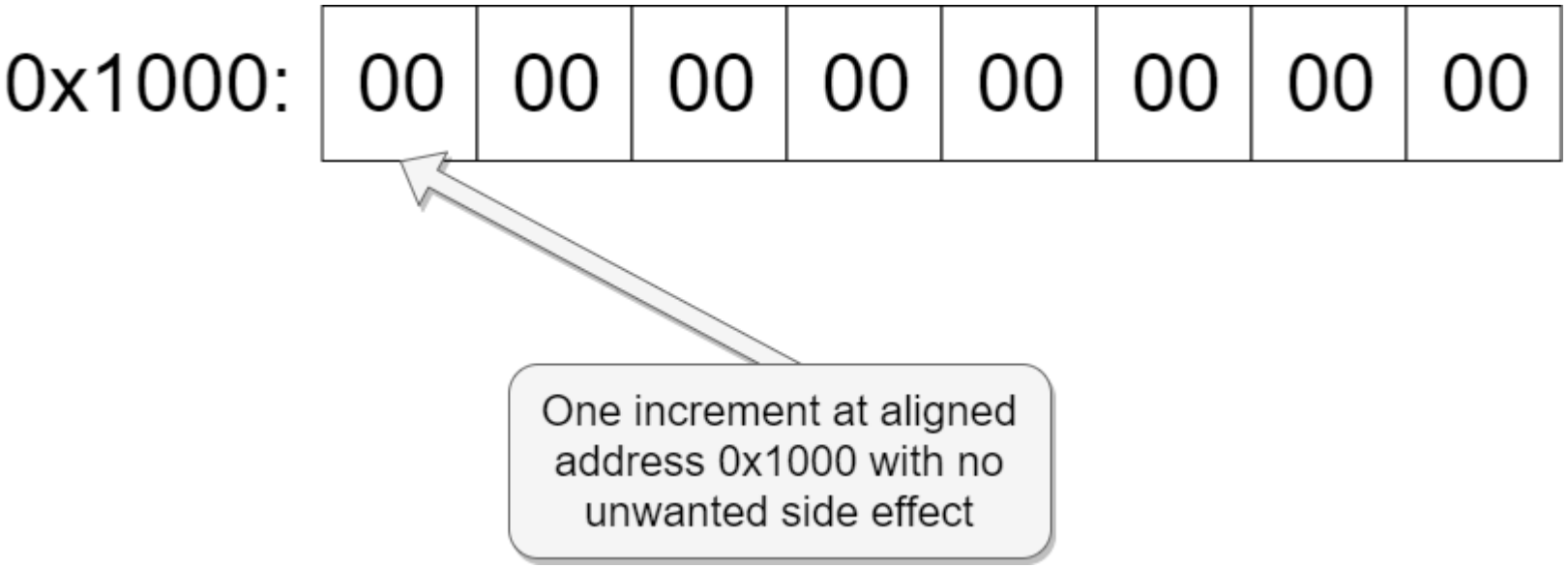
Efficient use of increment



Efficient use of increment



Efficient use of increment



How is that possible in practice?

- Max `2*(256*8) = 2048` writes
  - < `256*8` writes to reset value to zero
  - < `256*8` writes to set value to anything

Privilege escalation

Data only attack - Using the increment primitive

- We can trigger the increment primitive indefinitely
- Use the increment write primitive to enable an arbitrary read primitive
- Use the read primitive to read SYSTEM token
- Use the write primitive to adjust our EPROCESS token to SYSTEM
- Caveats: If EPROCESS token is read during our slow adjustment, we BSOD
  - If Task Manager is running
  - If Process Explorer is running

Exploiting Windows 10 1809 (RS5) x86/x64

- Data only attack
- Bypassing kernel CFG wasn't investigated (yet)
  - But primitives should make it doable
- Only major x64 and x86 differences is structure sizes and offset
  - Except for the following thing to come...
- Relatively easy to port to all versions back to Vista

Bonus - BlueHat Shanghai May 2019

Bonus - The "invisible" paper

- Turns out Kaspersky presented on this in May 2019 at BlueHat Shanghai 2019
  - Boris Larin (@oct0xor) and Anton Ivanov (@antonivanovm)
  - Explains some of what we just described
- Found after we got accepted to speak at POC2019
  - win32k syscall filter search keywords found it by accident
  - Searching CVE-2018-8611 or KTM did not

Me reading the paper

Bonus - The "invisible" paper

- Actually quite happy in the end we never saw it!
- Most interesting highlight
  - 0day exploit used multiple different approaches from us

Bonus - Race winning detection

- 0day didn't use same trap enlistment approach to detect race win
- Used Event Notification object to trap kernel on `KeWaitForSingleObject()`
  - Swap object type after detection
  - Modified mutex allows write 0 primitive (similar code path to ours)
  - Positives
    - It's interesting to see a different approach
  - Negatives
    - Must modify every mutex that gets touched by loop
    - More complicated than our primitive

Bonus - Write primitive: No increment, write 0 only

- 0day didn't use the increment primitive either!
- Abused an earlier write 0 in same `KeReleaseMutex()` code path
  - Writes a `sizeof(void *)` 0 value to any address
    - Least significant bit must already be 0 to avoid deadlock
  - Positives
    - Reduced setup complexity to reach write primitive
  - Negatives
    - Doesn't work on all OS versions
      - Vista x64 due to code differences
      - Vista/7 x86 because whole pointer required to be 0 in the first place (different macro)
    - Situationally less powerful primitive

Bonus - What to write with 0?

- 0day targeted `KTHREAD.PreviousMode` field
  - First documented by Tarjei Mandt in 2011
  - Misaligned write to this field allows setting to 0
  - Unrestricted `NtReadVirtualMemory()` and `NtWriteVirtualMemory()`
    - Arbitrary kernel read/write
  - Positives:
    - Super powerful (64-bit)
    - Possibly first in-the-wild use?
  - Negatives
    - Doesn't work on x86 due to `PreviousMode` being reset to `UserMode` based on `CS` when syscall entered

Demo

Conclusion

- Quite reliably exploitable race condition leading to UAF
- Very interesting and fun to exploit
- Should be usable to bypass most kernel mitigations (if necessary)
  - KASLR, SMEP, CFG, etc.
- Our approach differed significantly from 0day
  - Both methods have a lot of value!
- Tons of details still missing
  - Follow up 5 part blog series coming

Questions?

- Cedric Halbronn - @saidelike, cedric.halbronn@nccgroup.com