Exploiting CVE-2018-8611

Windows Kernel Transaction Manager (KTM) Race Condition

Aaron Adams - Power Of Community 2019

About

- Exploit Development Group (EDG), NCC Group
- Occasionally publish stuff: Cisco ASA, Xen, Samba, Stagefright, win32k
- Write exploits to help consultants do their job
- Focus on patched vulns

Aaron Adams

- Presenting
- @fidgetingbits, aaron.adams@nccgroup.com

Cedric Halbronn

- Unable to attend
- @saidelike, cedric.halbronn@nccgroup.com

This talk

- Discuss an interesting race condition affecting Microsoft Kernel Transaction Manager (KTM)
  - Found used in the wild by Kaspersky
- 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

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 - Kaspersky blog
- 2019 - Proton Bot malware uses KTM
  - Used transacted versions of common functions to evade API inspection

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
- Slides: Remarkjs

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 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

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);
    [...]
    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 manager 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 manager
- 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

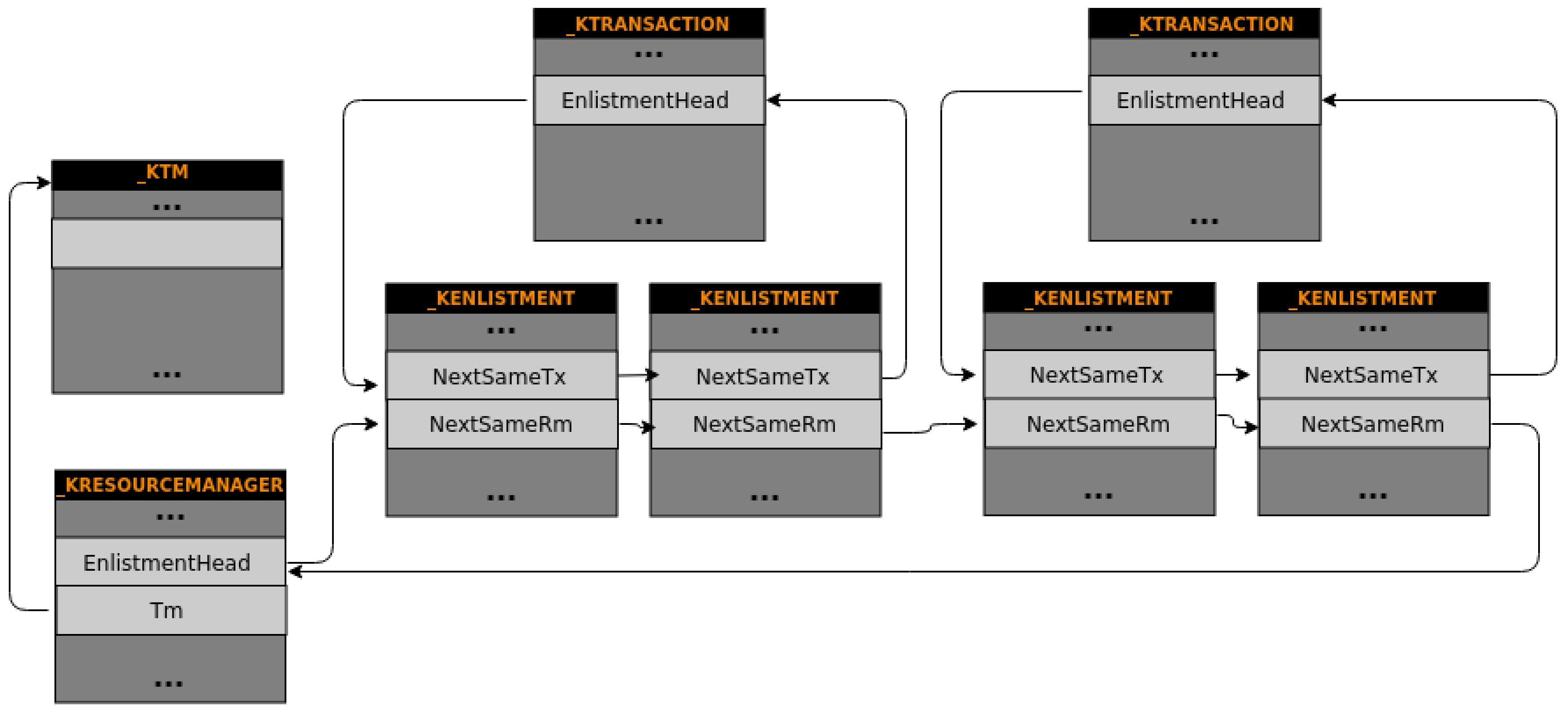
- The `Flags` field uses undocumented flags

```
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 counts
  - 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

```
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

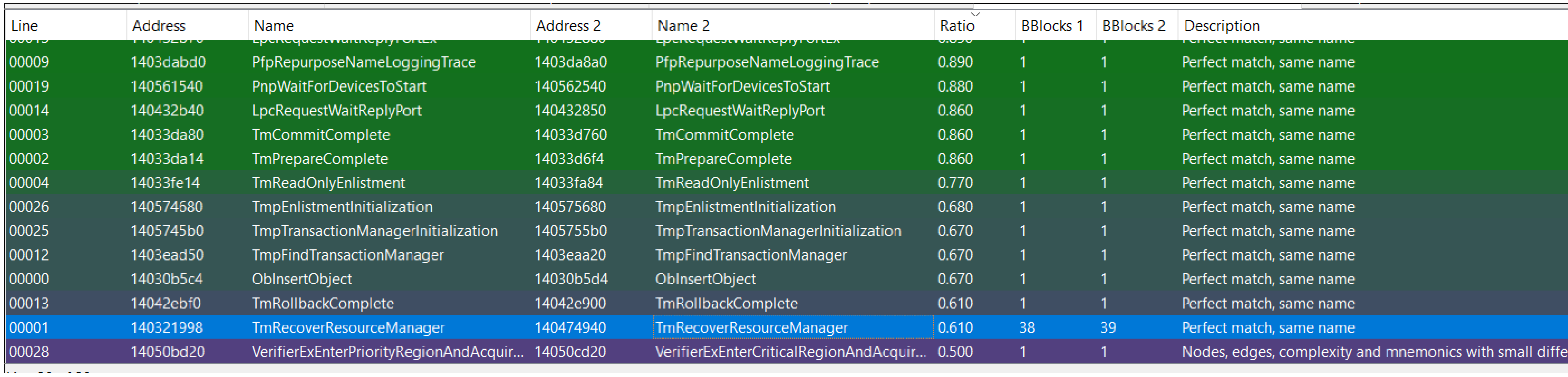
- If a Tx fails or is interrupted for whatever reason, it can be possible to recover
- Recovery in part possible by calling `RecoverResourceManager()`

```
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

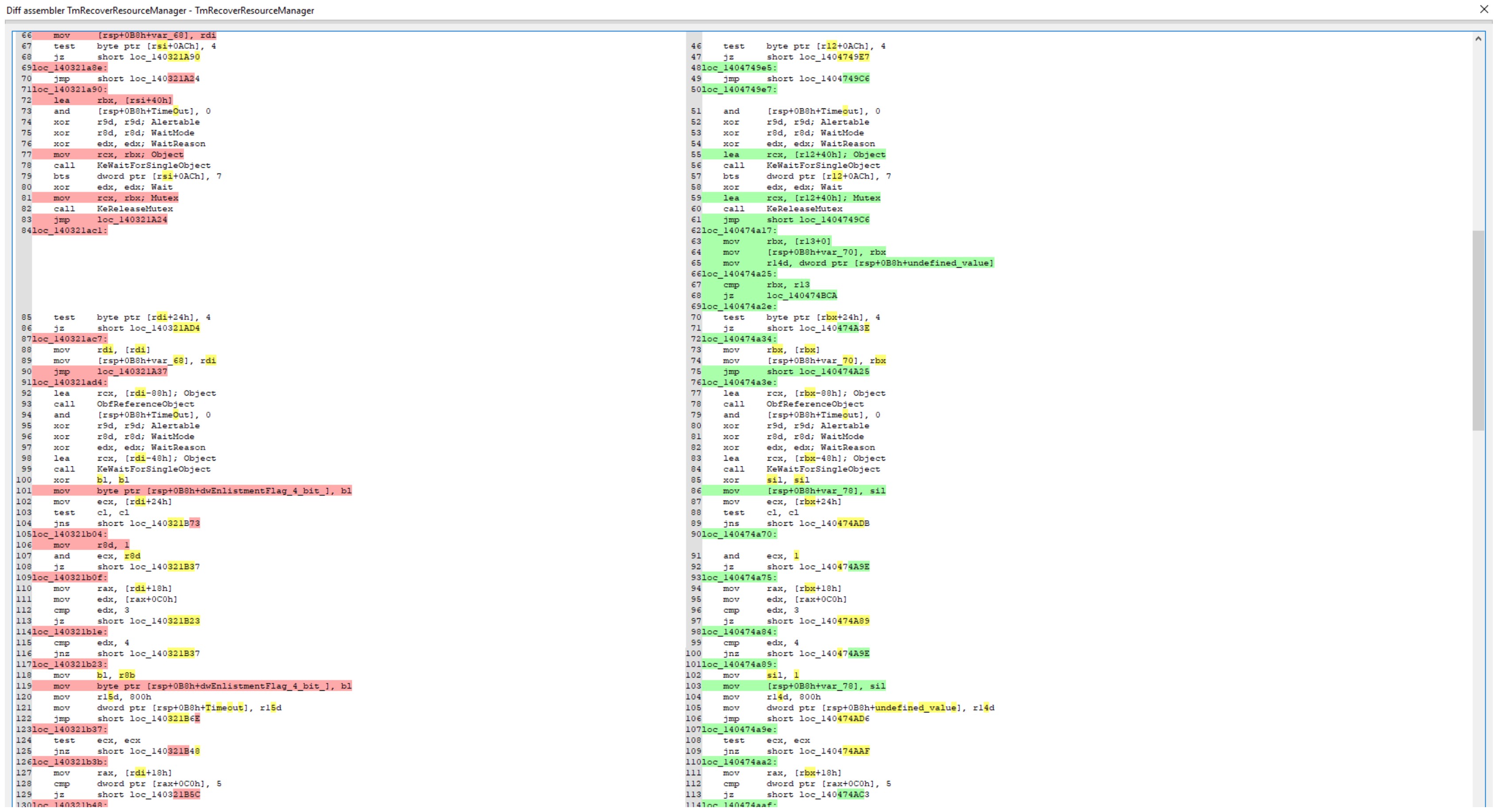
Understanding CVE-2018-8611

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

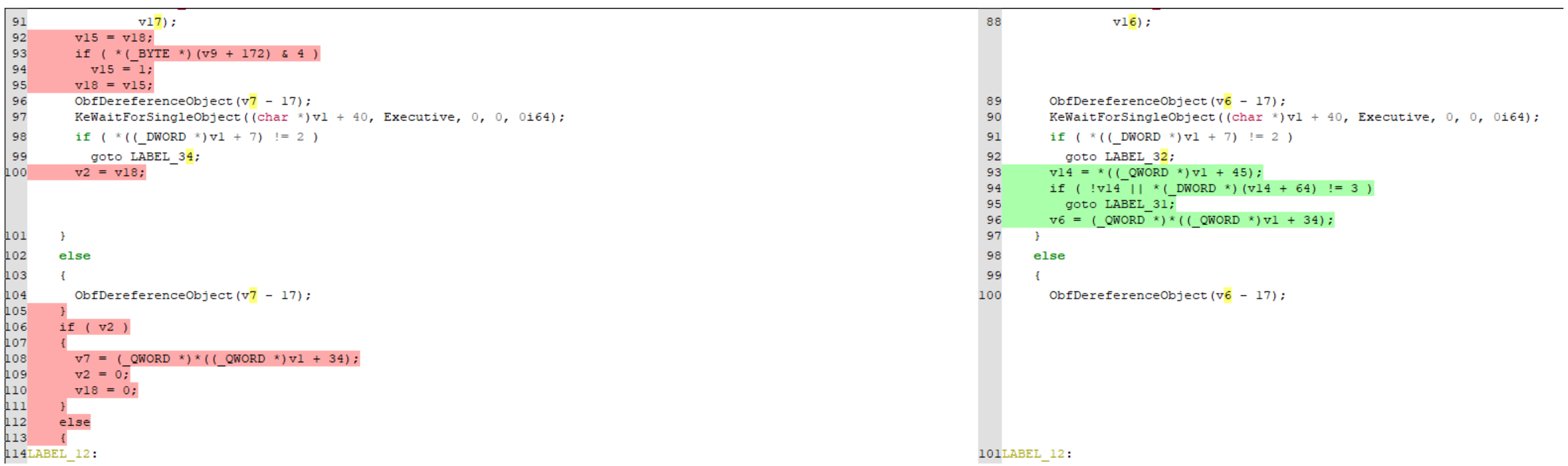


Diff assembler TmRecoverResourceManager - TmRecoverResourceManager

```
    mov     [rsp+0B8h+var_68], rdi
    test    byte ptr [rsi+0ACh], 4
    jz      short loc_140321A90
loc_140321a8e:
    jmp     short loc_140321A24
loc_140321a90:
    lea     rbx, [rsi+40h]
    and     [rsp+0B8h+TimeOut], 0
    xor     r9d, r9d; Alertable
    xor     r8d, r8d; WaitMode
    xor     edx, edx; WaitReason
    mov     rcx, rbx; Object
    call    KeWaitForSingleObject
    bts     dword ptr [rsi+0ACh], 7
    xor     edx, edx; Wait
    mov     rcx, rbx; Mutex
    call    KeReleaseMutex
    jmp     loc_140321A24
loc_140321ac1:







    test    byte ptr [rdi+24h], 4
    jz      short loc_140321AD4
loc_140321ac7:
    mov     rdi, [rdi]
    mov     [rsp+0B8h+var_68], rdi
    jmp     loc_140321A37
loc_140321ad4:
    lea     rcx, [rdi-88h]; Object
    call    ObfReferenceObject
    and     [rsp+0B8h+TimeOut], 0
    xor     r9d, r9d; Alertable
    xor     r8d, r8d; WaitMode
    xor     edx, edx; WaitReason
    lea     rcx, [rdi-48h]; Object
    call    KeWaitForSingleObject
    xor     bl, bl
    mov     byte ptr [rsp+0B8h+dwEnlistmentFlag_4_bit_], bl
    mov     ecx, [rdi+24h]
    test    cl, cl
    jns     short loc_140321B73
loc_140321b04:
    mov     r8d, 1
    and     ecx, r8d
    jz      short loc_140321B37
loc_140321b0f:
    mov     rax, [rdi+18h]
    mov     edx, [rax+0C0h]
    cmp     edx, 3
    jz      short loc_140321B23
loc_140321b1e:
    cmp     edx, 4
    jnz     short loc_140321B37
loc_140321b23:
    mov     bl, r8b
    mov     byte ptr [rsp+0B8h+dwEnlistmentFlag_4_bit_], bl
    mov     r15d, 800h
    mov     dword ptr [rsp+0B8h+Timeout], r15d
    jmp     short loc_140321B6E
loc_140321b37:
    test    ecx, ecx
    jnz     short loc_140321B48
loc_140321b3b:
    mov     rax, [rdi+18h]
    cmp     dword ptr [rax+0C0h], 5
    jz      short loc_140321B5C
loc_140321b48:
```

```

    test    byte ptr [r12+0ACh], 4
    jz      short loc_1404749E7
loc_1404749e5:
    jmp     short loc_1404749C6
loc_1404749e7:

    and     [rsp+0B8h+Timeout], 0
    xor     r9d, r9d; Alertable
    xor     r8d, r8d; WaitMode
    xor     edx, edx; WaitReason
    lea     rcx, [r12+40h]; Object
    call    KeWaitForSingleObject
    bts     dword ptr [r12+0ACh], 7
    xor     edx, edx; Wait
    lea     rcx, [r12+40h]; Mutex
    call    KeReleaseMutex
    jmp     short loc_1404749C6
loc_140474a17:
    mov     rbx, [r13+0]
    mov     [rsp+0B8h+var_70], rbx
    mov     r14d, dword ptr [rsp+0B8h+undefined_value]
loc_140474a25:
    cmp     rbx, r13
    jz      loc_140474BCA
loc_140474a2e:
    test    byte ptr [rbx+24h], 4
    jz      short loc_140474A3E
loc_140474a34:
    mov     rbx, [rbx]
    mov     [rsp+0B8h+var_70], rbx
    jmp     short loc_140474A25
loc_140474a3e:
    lea     rcx, [rbx-88h]; Object
    call    ObfReferenceObject
    and     [rsp+0B8h+Timeout], 0
    xor     r9d, r9d; Alertable
    xor     r8d, r8d; WaitMode
    xor     edx, edx; WaitReason
    lea     rcx, [rbx-48h]; Object
    call    KeWaitForSingleObject
    xor     sil, sil
    mov     [rsp+0B8h+var_78], sil
    mov     ecx, [rbx+24h]
    test    cl, cl
    jns     short loc_140474ADB
loc_140474a70:

    and     ecx, 1
    jz      short loc_140474A9E
loc_140474a75:
    mov     rax, [rbx+18h]
    mov     edx, [rax+0C0h]
    cmp     edx, 3
    jz      short loc_140474A89
loc_140474a84:
    cmp     edx, 4
    jnz     short loc_140474A9E
loc_140474a89:
    mov     sil, 1
    mov     [rsp+0B8h+var_78], sil
    mov     r14d, 800h
    mov     dword ptr [rsp+0B8h+undefined_value], r14d
    jmp     short loc_140474AD6
loc_140474a9e:
    test    ecx, ecx
    jnz     short loc_140474AAF
loc_140474aa2:
    mov     rax, [rbx+18h]
    cmp     dword ptr [rax+0C0h], 5
    jz      short loc_140474AC3
loc_140474aaf:
```

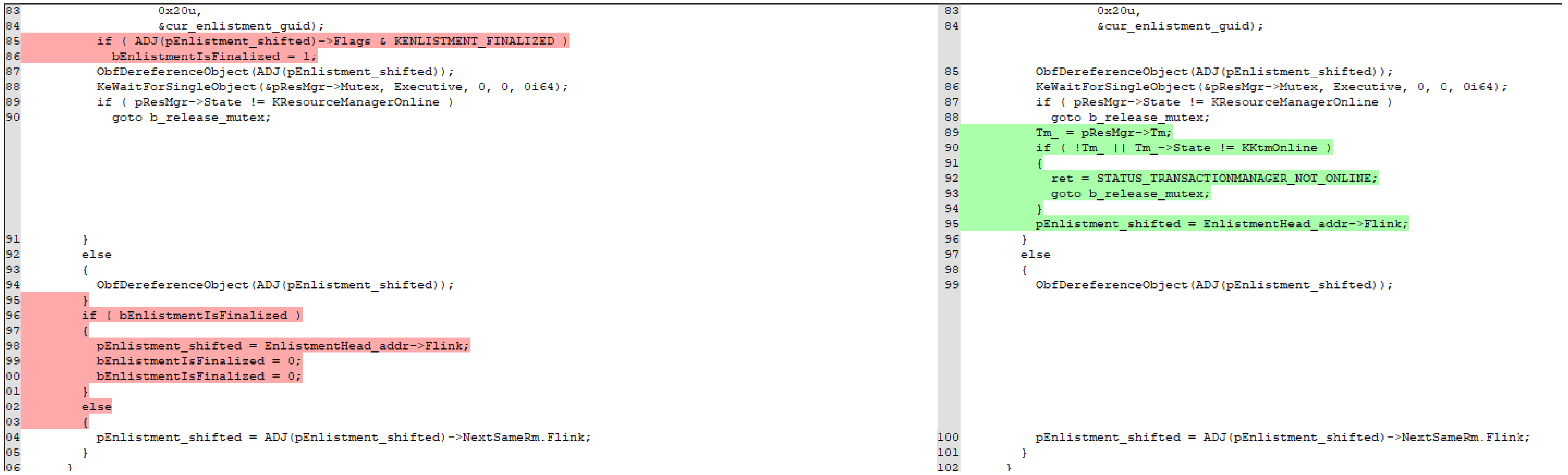
Diffing - Hex-Rays pre-cleanup



```
              v17);
      v15 = v18;
      if ( *(_BYTE *)(v9 + 172) & 4 )
        v15 = 1;
      v18 = v15;
      ObfDereferenceObject(v7 - 17);
      KeWaitForSingleObject((char *)v1 + 40, Executive, 0, 0, 0i64);
      if ( *((_DWORD *)v1 + 7) != 2 )
        goto LABEL_34;
      v2 = v18;
    }
    else
    {
      ObfDereferenceObject(v7 - 17);
    }
    if ( v2 )
    {
      v7 = (_QWORD *)*((_QWORD *)v1 + 34);
      v2 = 0;
      v18 = 0;
    }
    else
    {
LABEL_12:
```

```
              v16);
      ObfDereferenceObject(v6 - 17);
      KeWaitForSingleObject((char *)v1 + 40, Executive, 0, 0, 0i64);
      if ( *((_DWORD *)v1 + 7) != 2 )
        goto LABEL_32;
      v14 = *((_QWORD *)v1 + 45);
      if ( !v14 || *(_DWORD *)(v14 + 64) != 3 )
        goto LABEL_31;
      v6 = (_QWORD *)*((_QWORD *)v1 + 34);
    }
    else
    {
      ObfDereferenceObject(v6 - 17);
LABEL_12:
```

Diffing - Hex-Rays post-cleanup

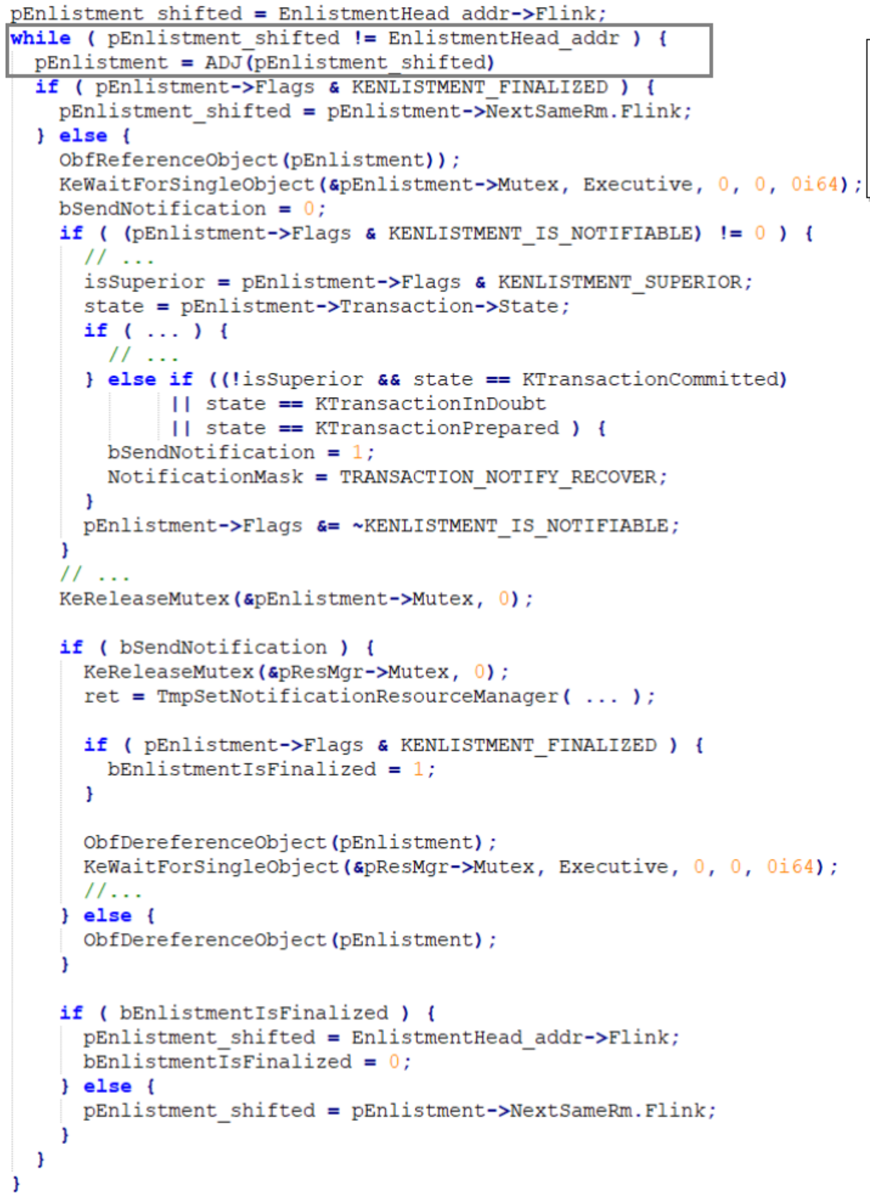


```
                0x20u,
                &cur_enlistment_guid);
        if ( ADJ(pEnlistment_shifted)->Flags & KENLISTMENT_FINALIZED )
          bEnlistmentIsFinalized = 1;
        ObfDereferenceObject(ADJ(pEnlistment_shifted));
        KeWaitForSingleObject(&pResMgr->Mutex, Executive, 0, 0, 0i64);
        if ( pResMgr->State != KResourceManagerOnline )
          goto b_release_mutex;
      }
      else
      {
        ObfDereferenceObject(ADJ(pEnlistment_shifted));
      }
      if ( bEnlistmentIsFinalized )
      {
        pEnlistment_shifted = EnlistmentHead_addr->Flink;
        bEnlistmentIsFinalized = 0;
        bEnlistmentIsFinalized = 0;
      }
      else
      {
        pEnlistment_shifted = ADJ(pEnlistment_shifted)->NextSameRm.Flink;
      }
    }
```

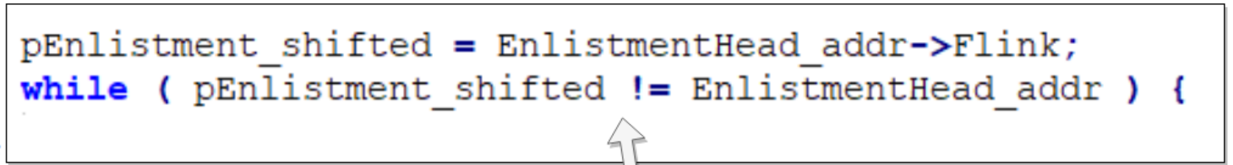
```
                0x20u,
                &cur_enlistment_guid);
        ObfDereferenceObject(ADJ(pEnlistment_shifted));
        KeWaitForSingleObject(&pResMgr->Mutex, Executive, 0, 0, 0i64);
        if ( pResMgr->State != KResourceManagerOnline )
          goto b_release_mutex;
        Tm_ = pResMgr->Tm;
        if ( !Tm_ || Tm_->State != KKtmOnline )
        {
          ret = STATUS_TRANSACTIONMANAGER_NOT_ONLINE;
          goto b_release_mutex;
        }
        pEnlistment_shifted = EnlistmentHead_addr->Flink;
      }
      else
      {
        ObfDereferenceObject(ADJ(pEnlistment_shifted));
        pEnlistment_shifted = ADJ(pEnlistment_shifted)->NextSameRm.Flink;
      }
    }
```

```
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



```
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->Transaction->State;
  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.

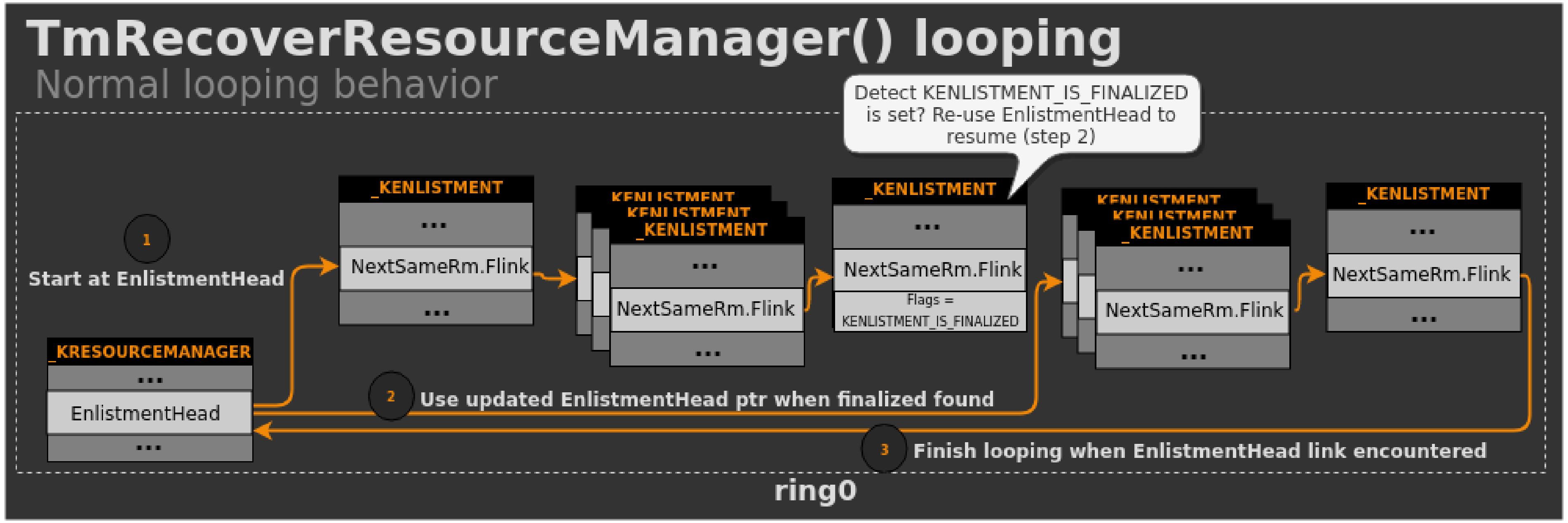
```
pEnlistment_shifted = EnlistmentHead_addr->Flink;
while ( pEnlistment_shifted != EnlistmentHead_addr ) {
  pEnlistment = ADJ(pEnlistment_shifted)
  if ( pEnlistment->Flags & KENLISTMENT_FINALIZED ) {
    pEnlistment_shifted = pEnlistment->NextSameRm.Flink;
  } else {
    ObfReferenceObject(pEnlistment));
    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?



TmRecoverResourceManager() looping
Normal looping behavior
Detect KENLISTMENT_IS_FINALIZED is set? Re-use EnlistmentHead to resume (step 2)
1
Start at EnlistmentHead
_KRESOURCEMANAGER
EnlistmentHead
_KENLISTMENT
NextSameRm.Flink
Flags = KENLISTMENT_IS_FINALIZED
2
Use updated EnlistmentHead ptr when finalized found
3
Finish looping when EnlistmentHead link encountered
ring0

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 when if 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

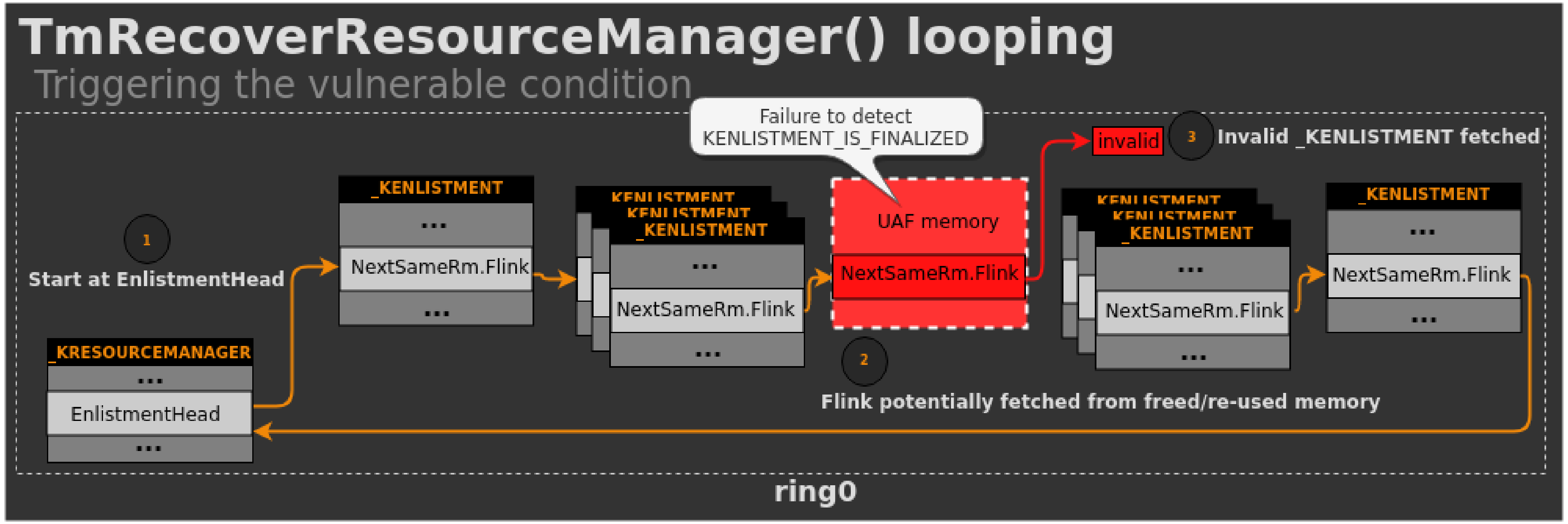
Faking a race win

- 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

Exploitable loop state



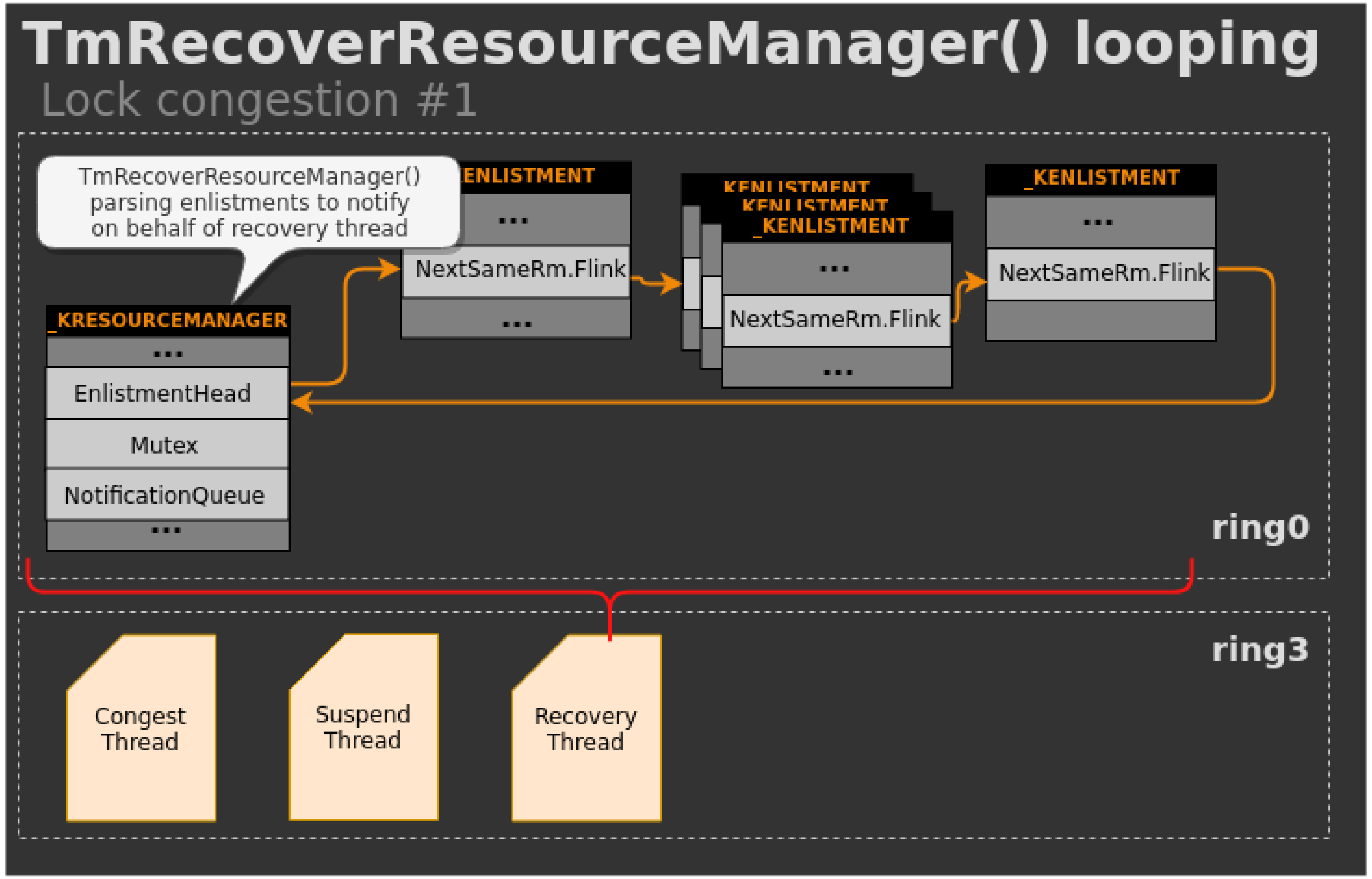
Which _KENLISTMENT to free?

- 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

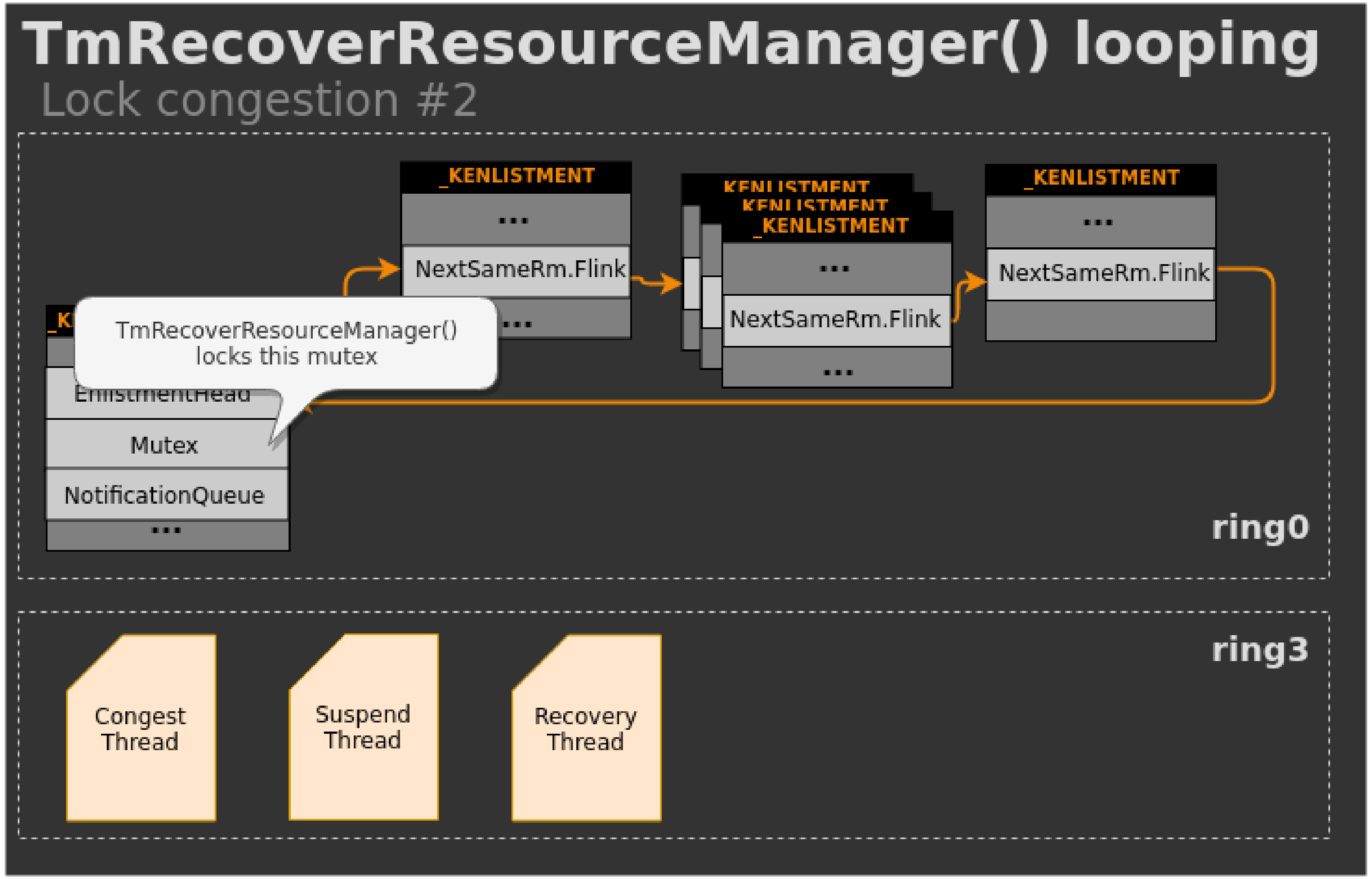
Actually winning the race

- 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
  - If it blocks at a time when the RM is unlocked, we are free to free
  - If not, no UAF happens, and we keep trying
- 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

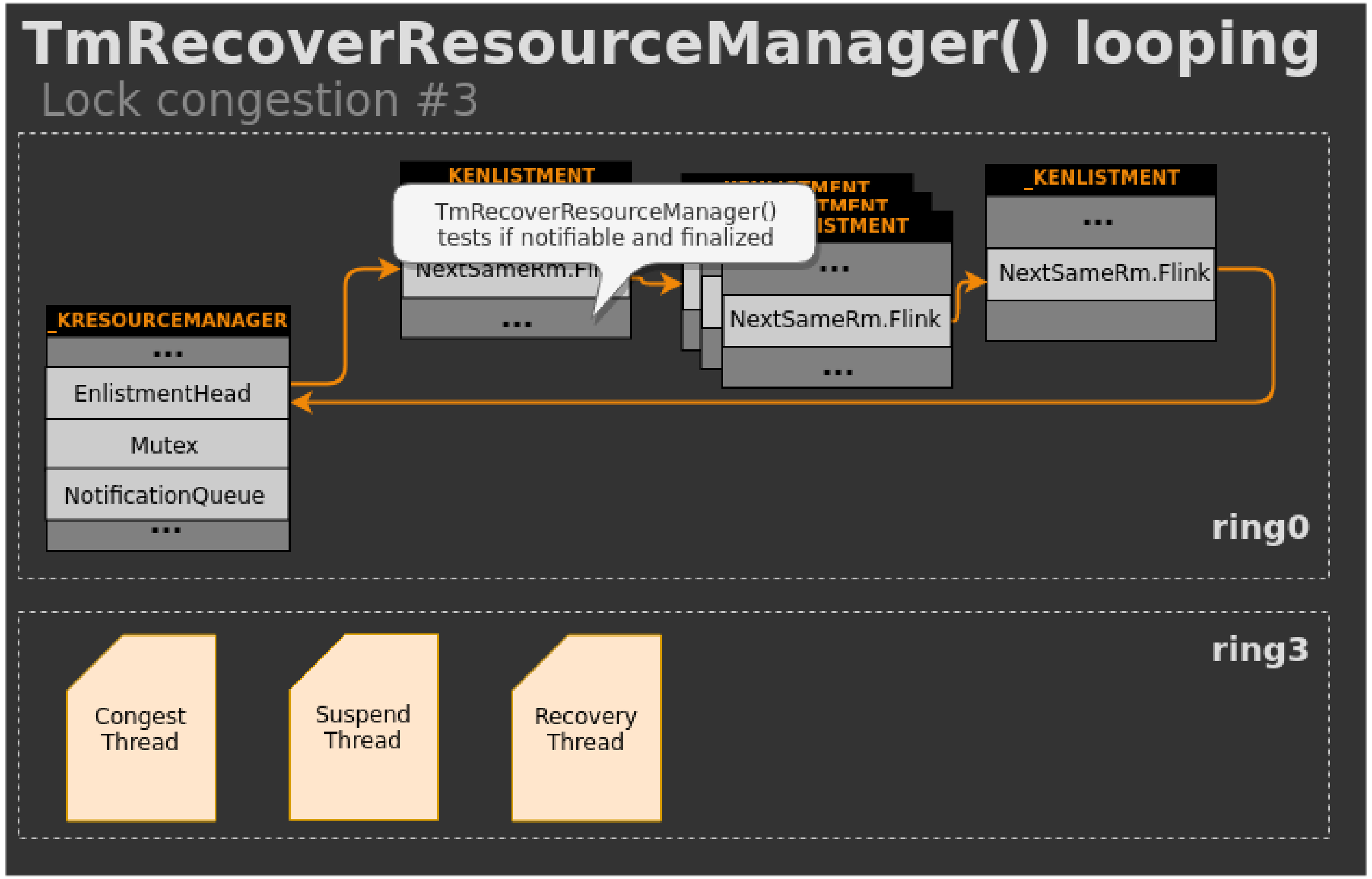
Lock congestion



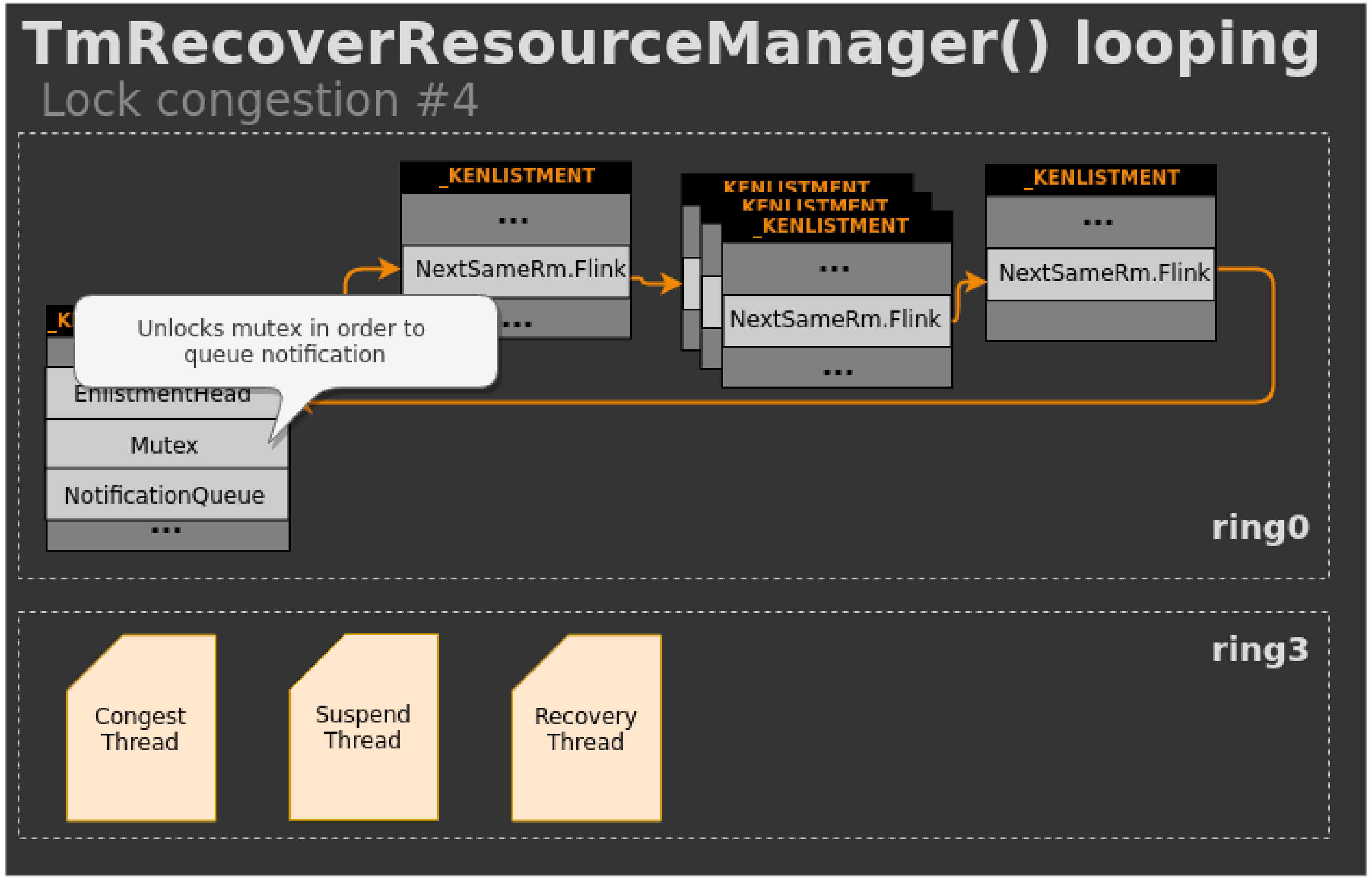
Lock congestion



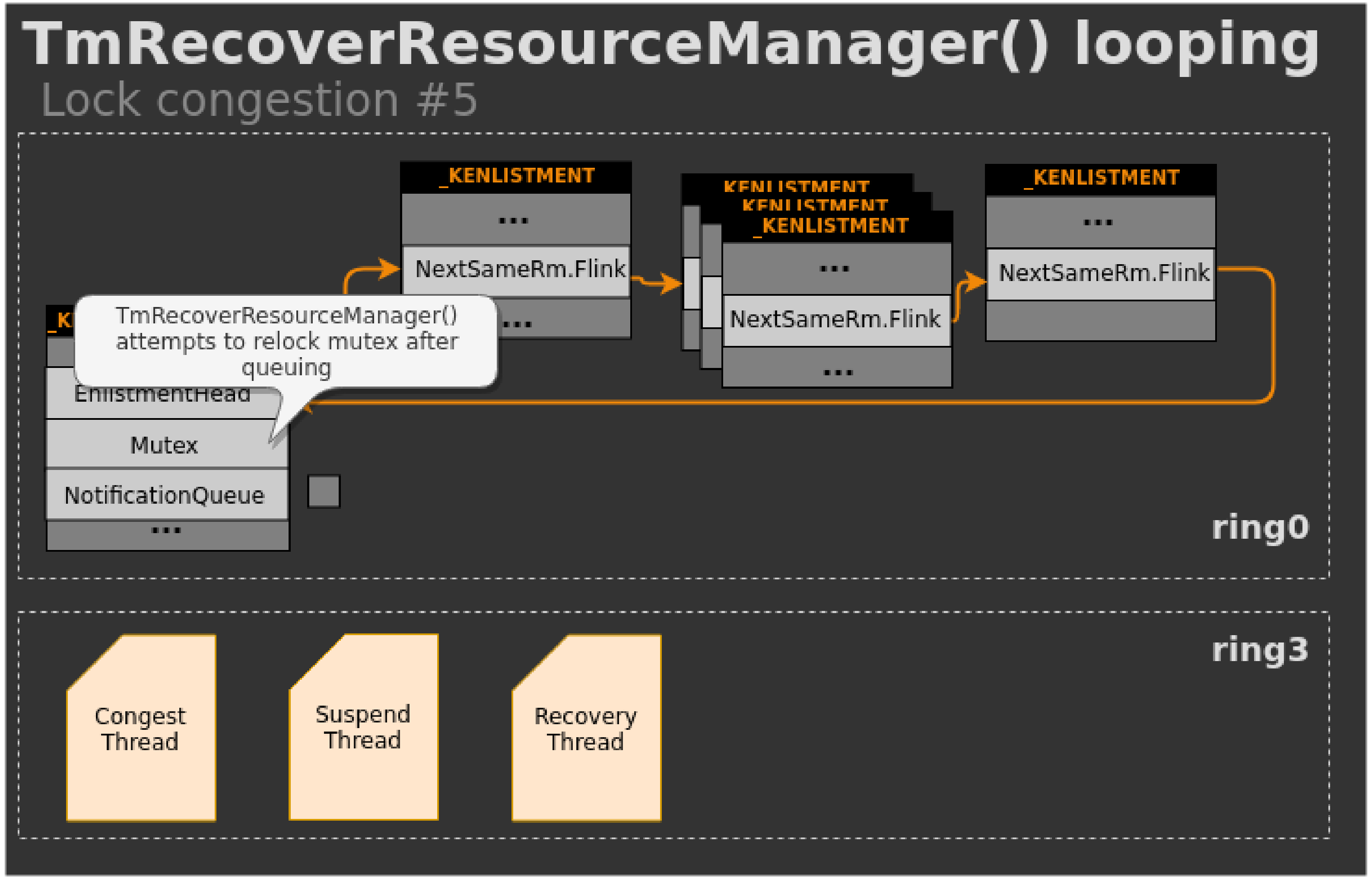
Lock congestion



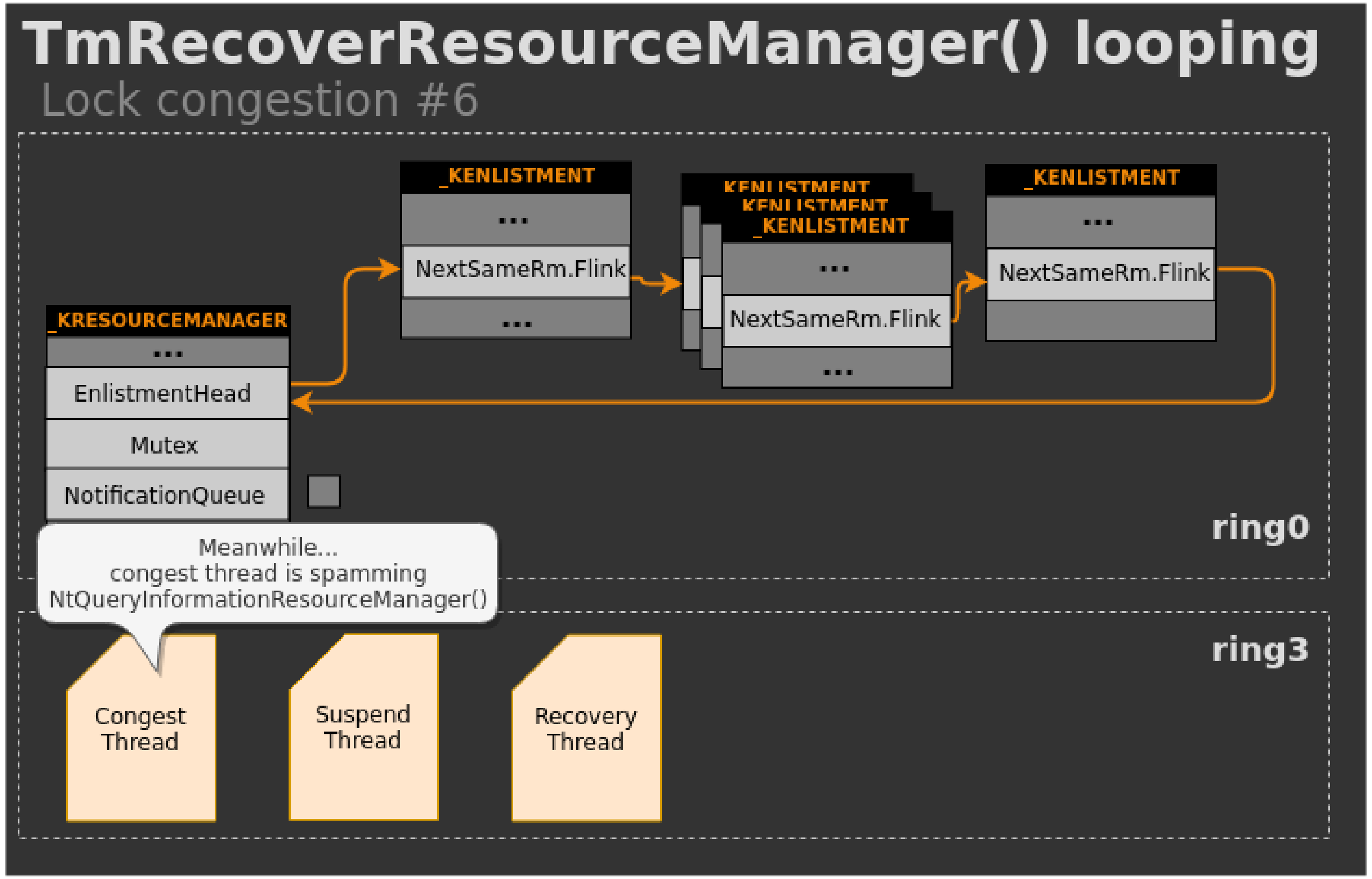
Lock congestion



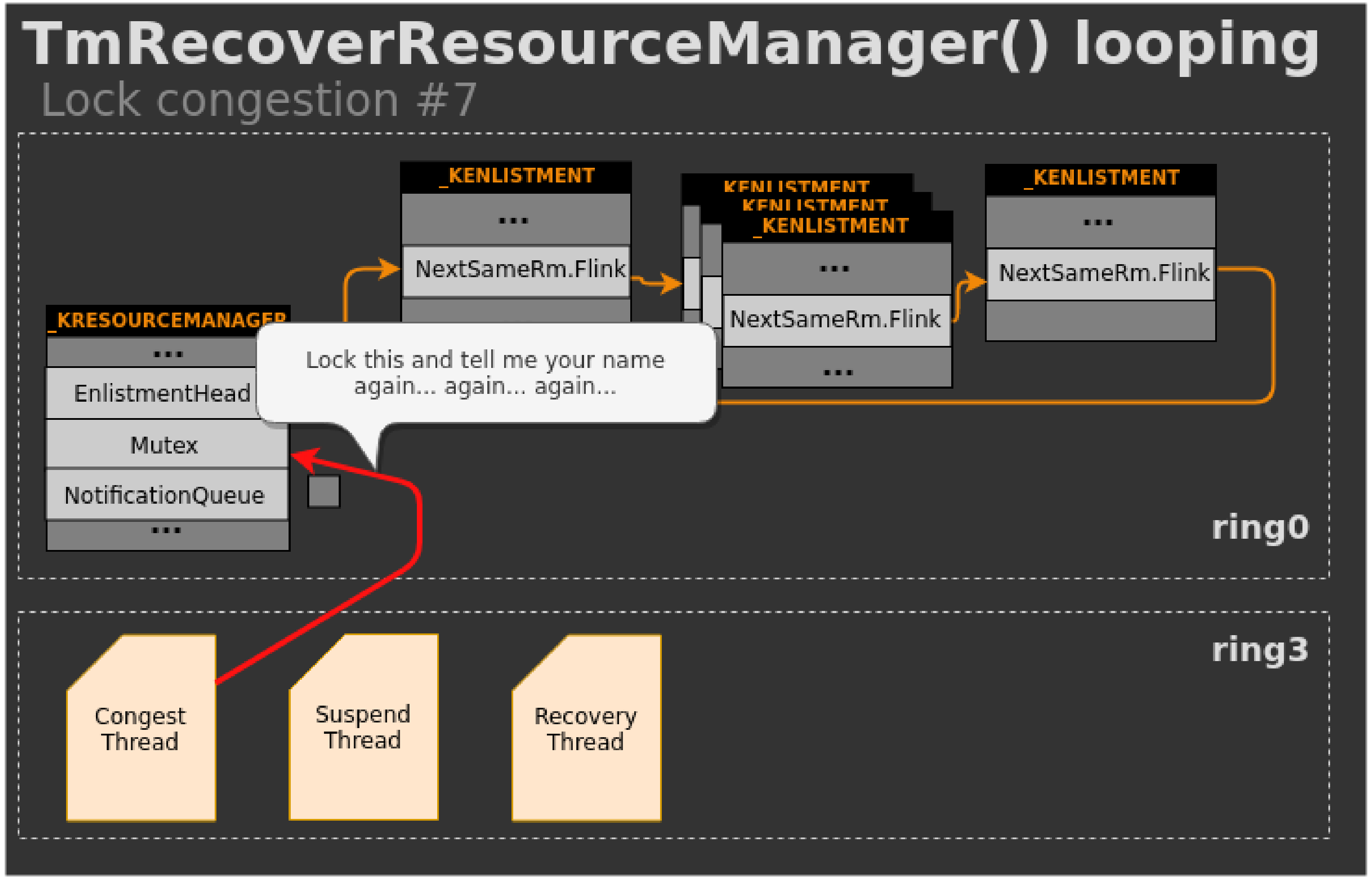
Lock congestion



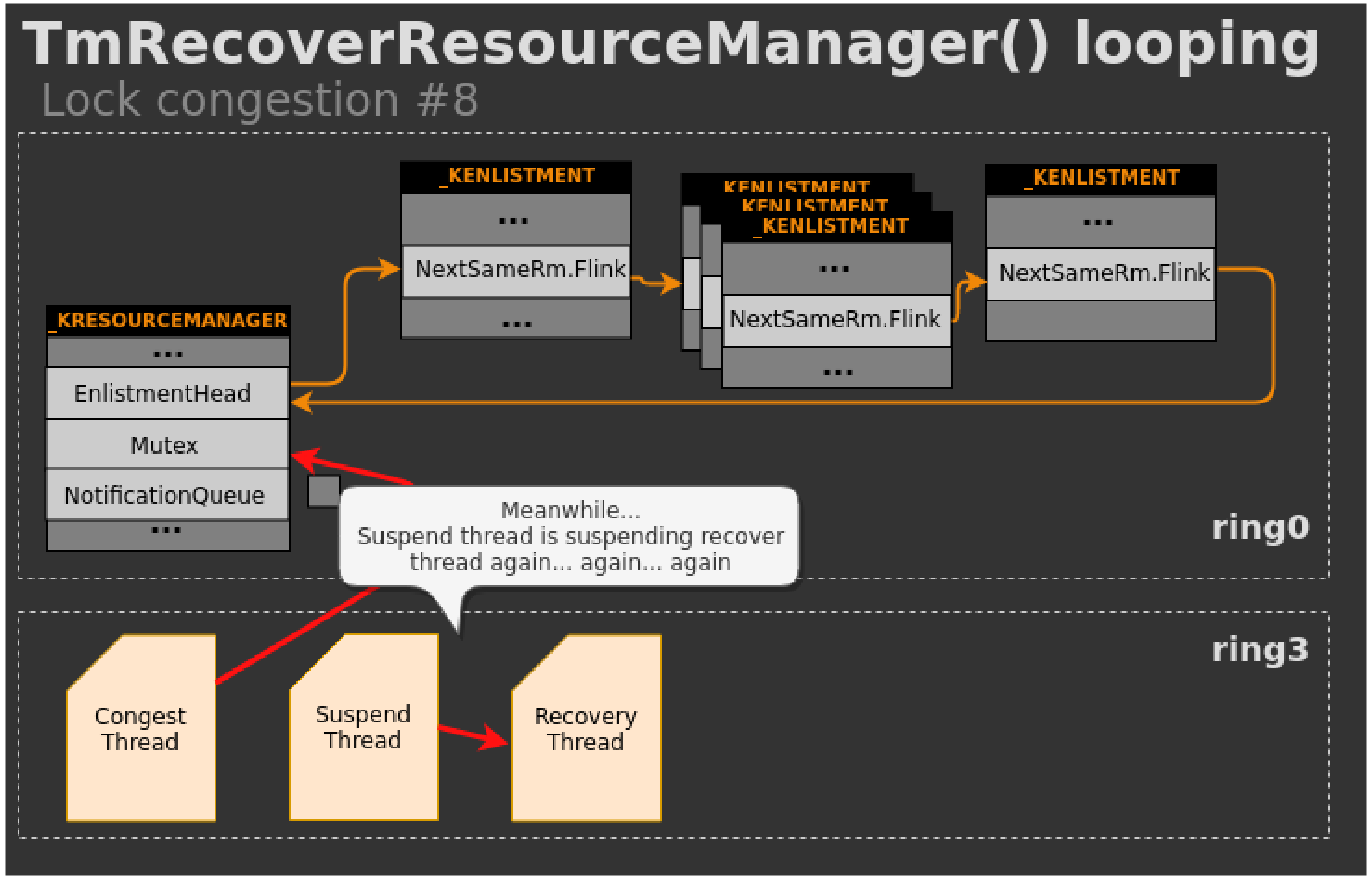
Lock congestion



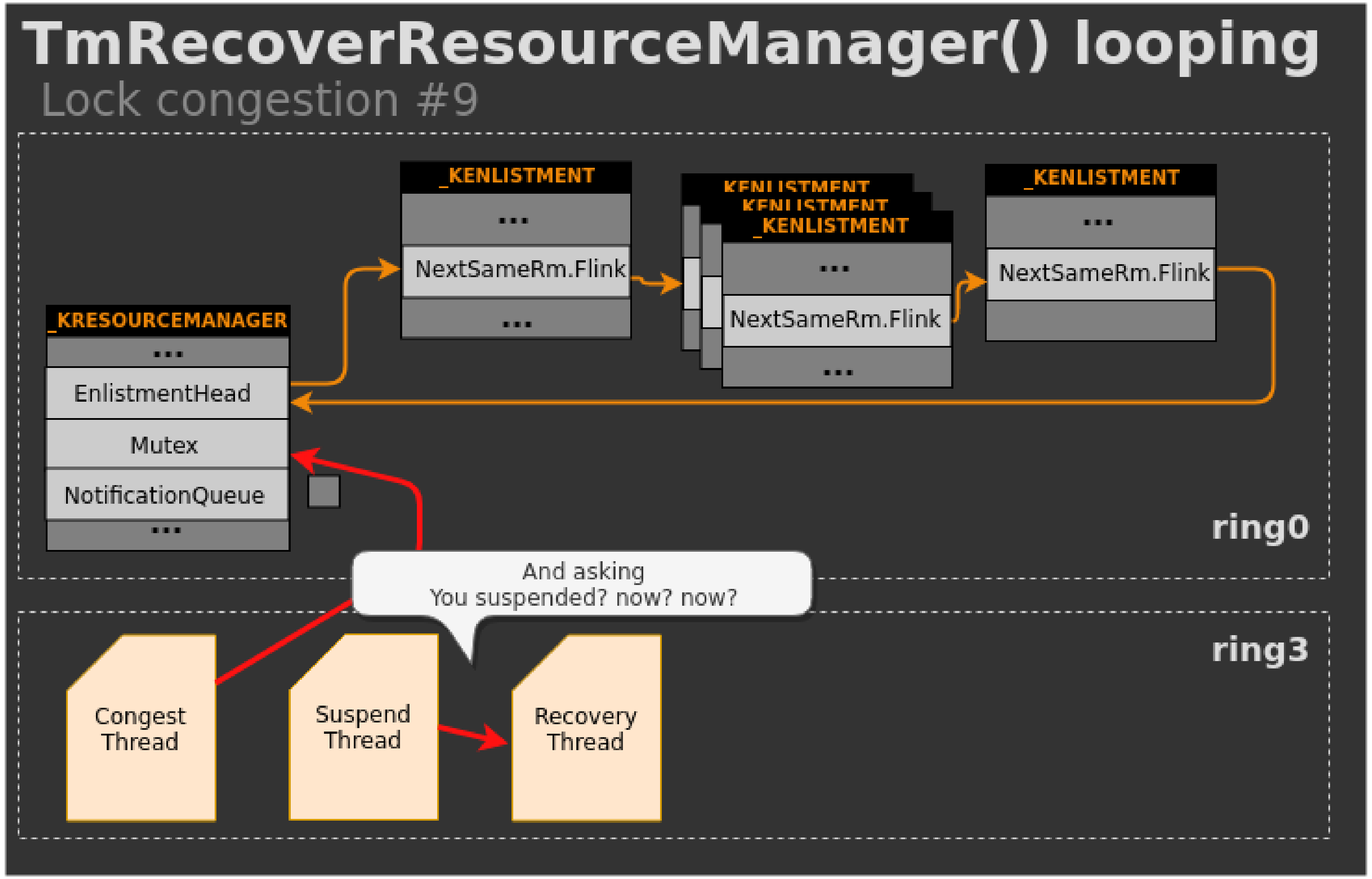
Lock congestion



Lock congestion



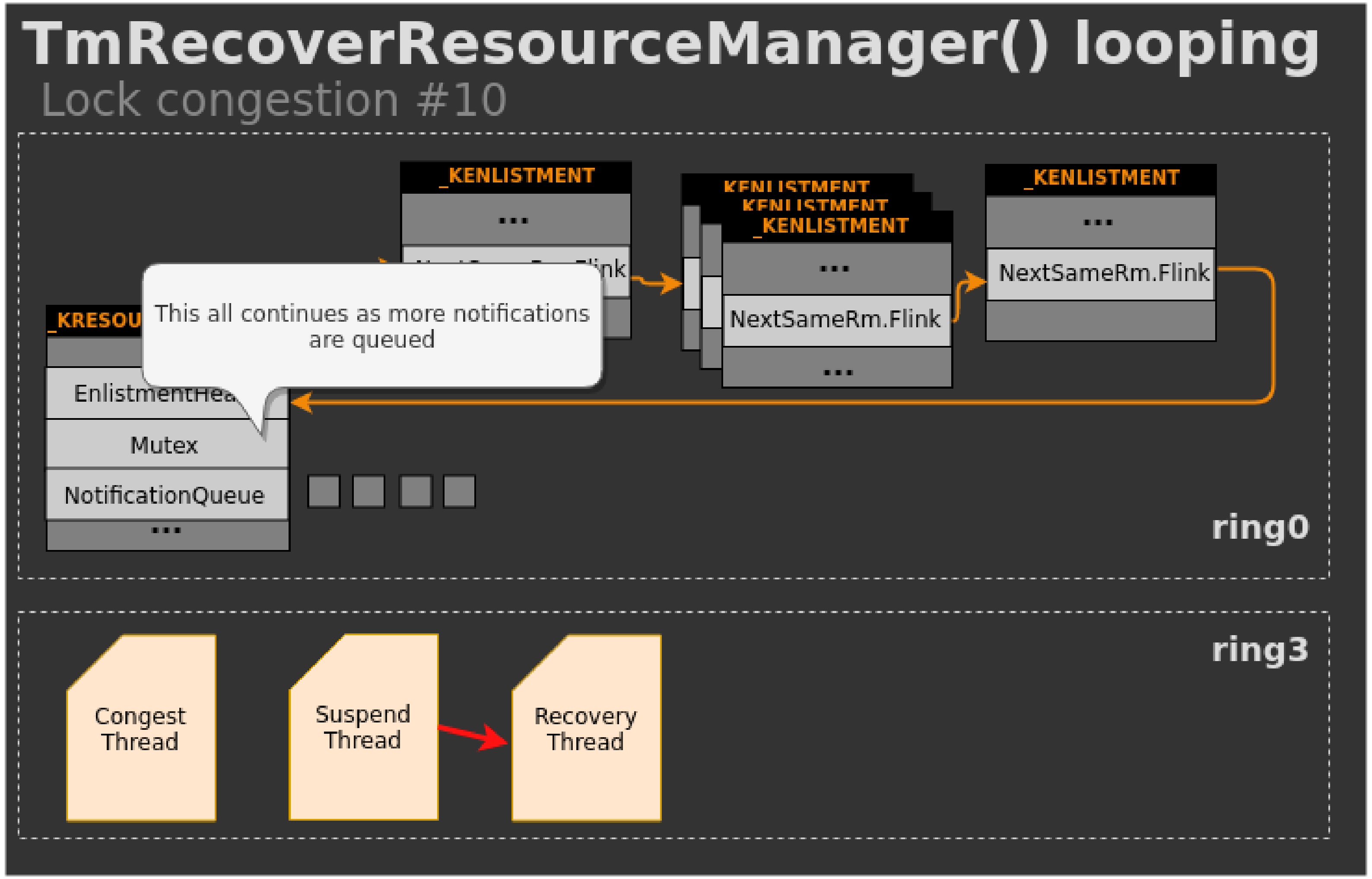
Lock congestion



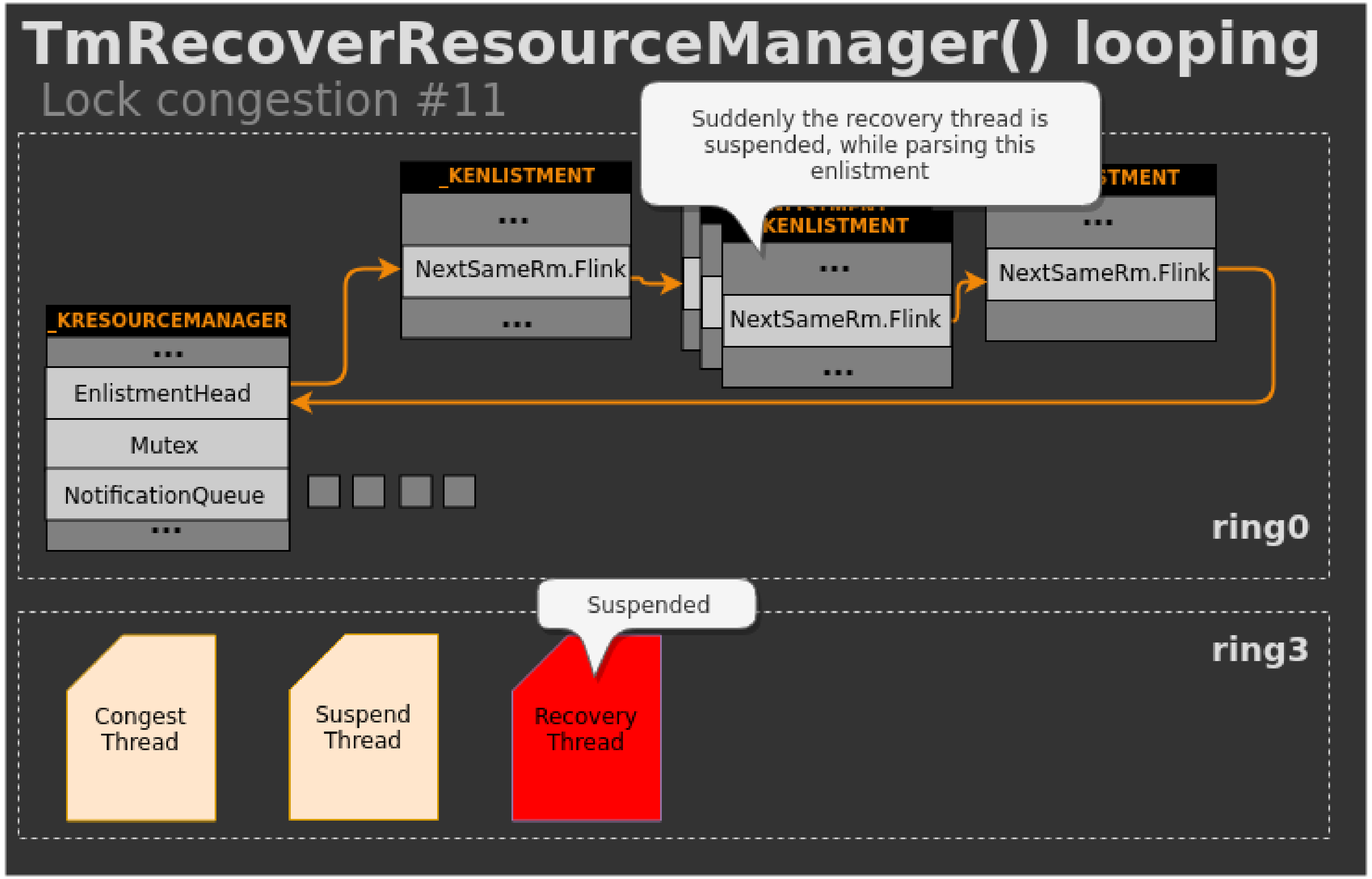
Thread suspension detection

- A thread will become blocked on some natural blocking point
  - Like waiting to lock the congested resource manager mutex
- 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

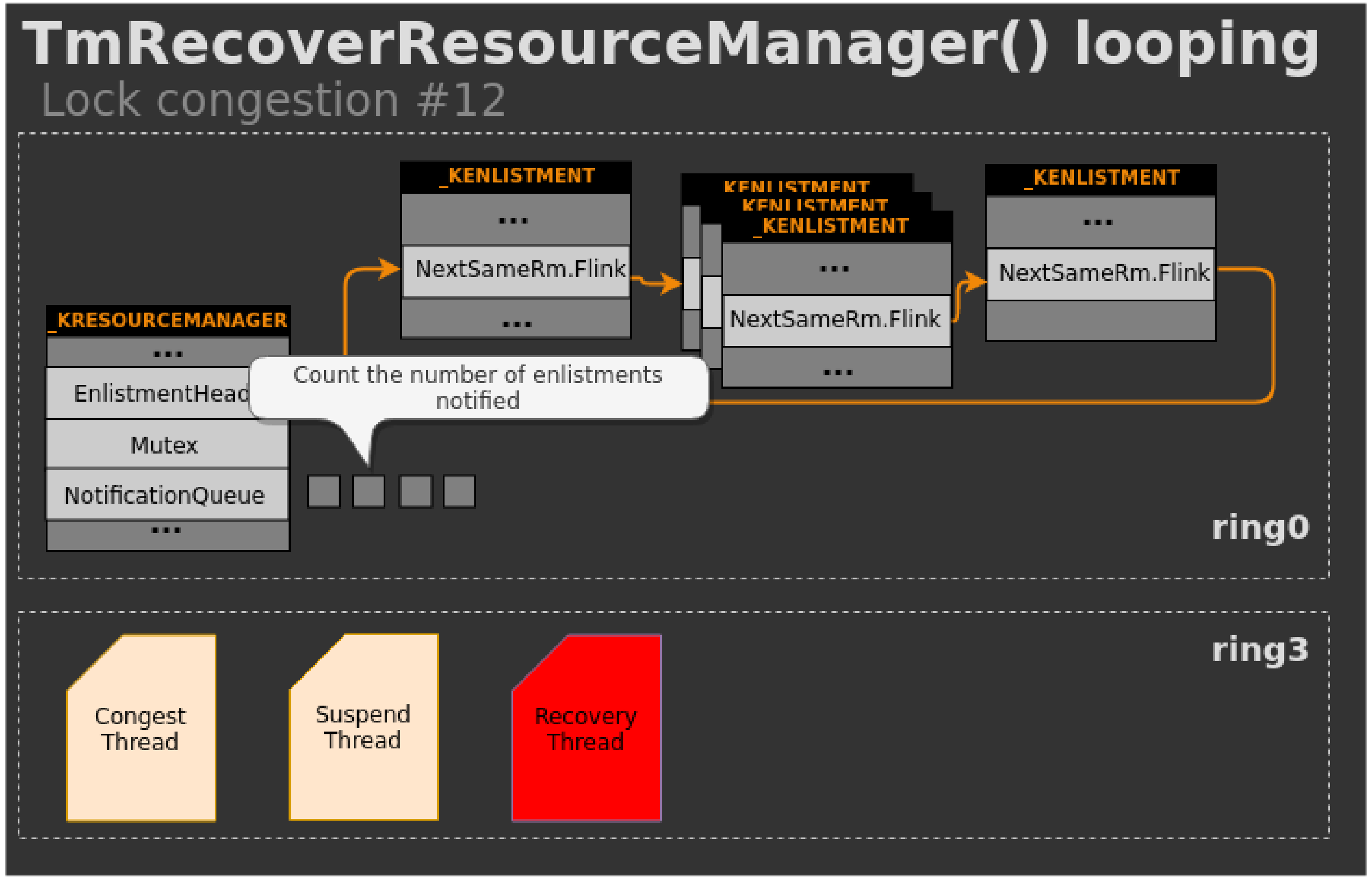
Lock congestion



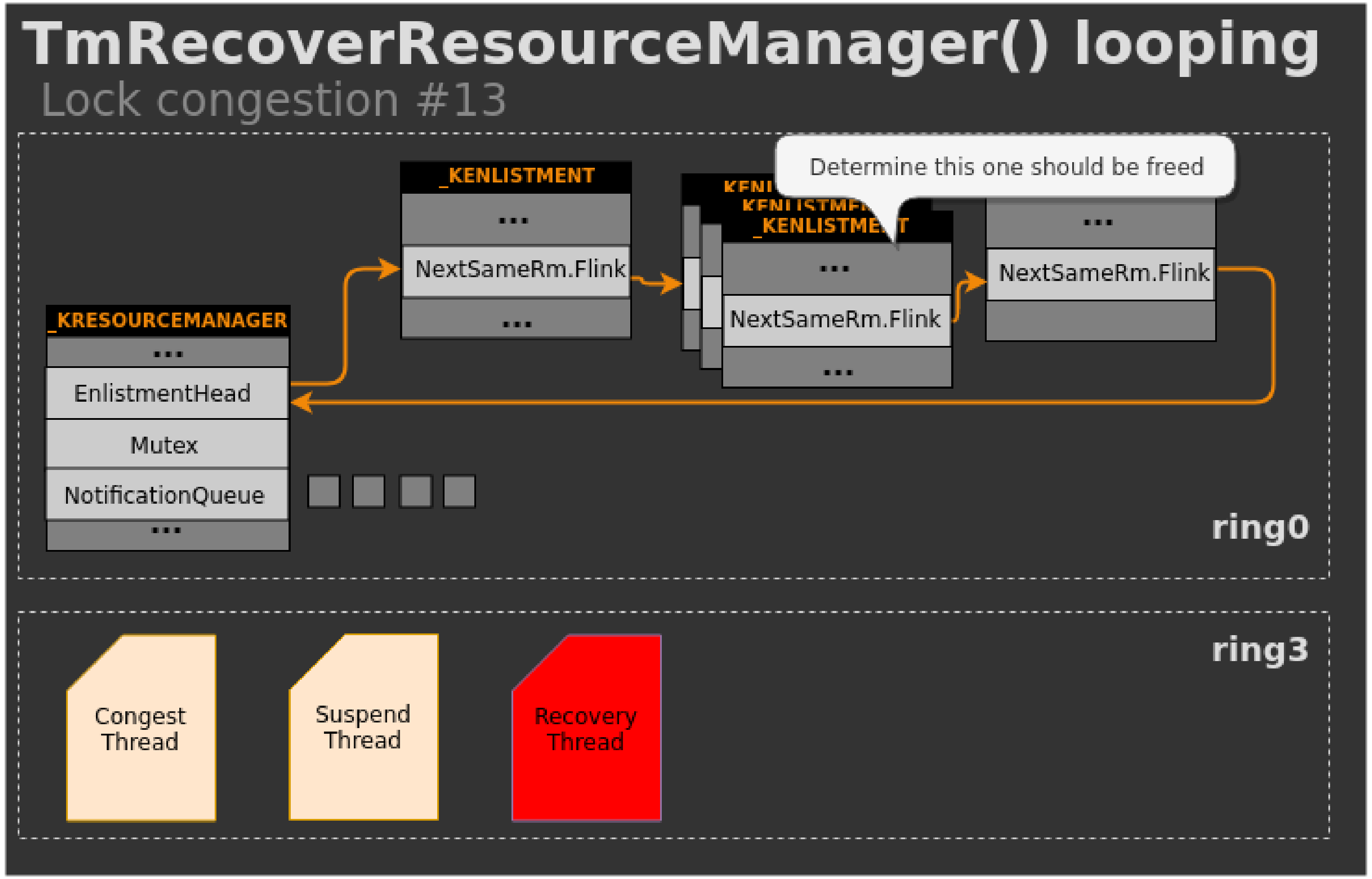
Lock congestion



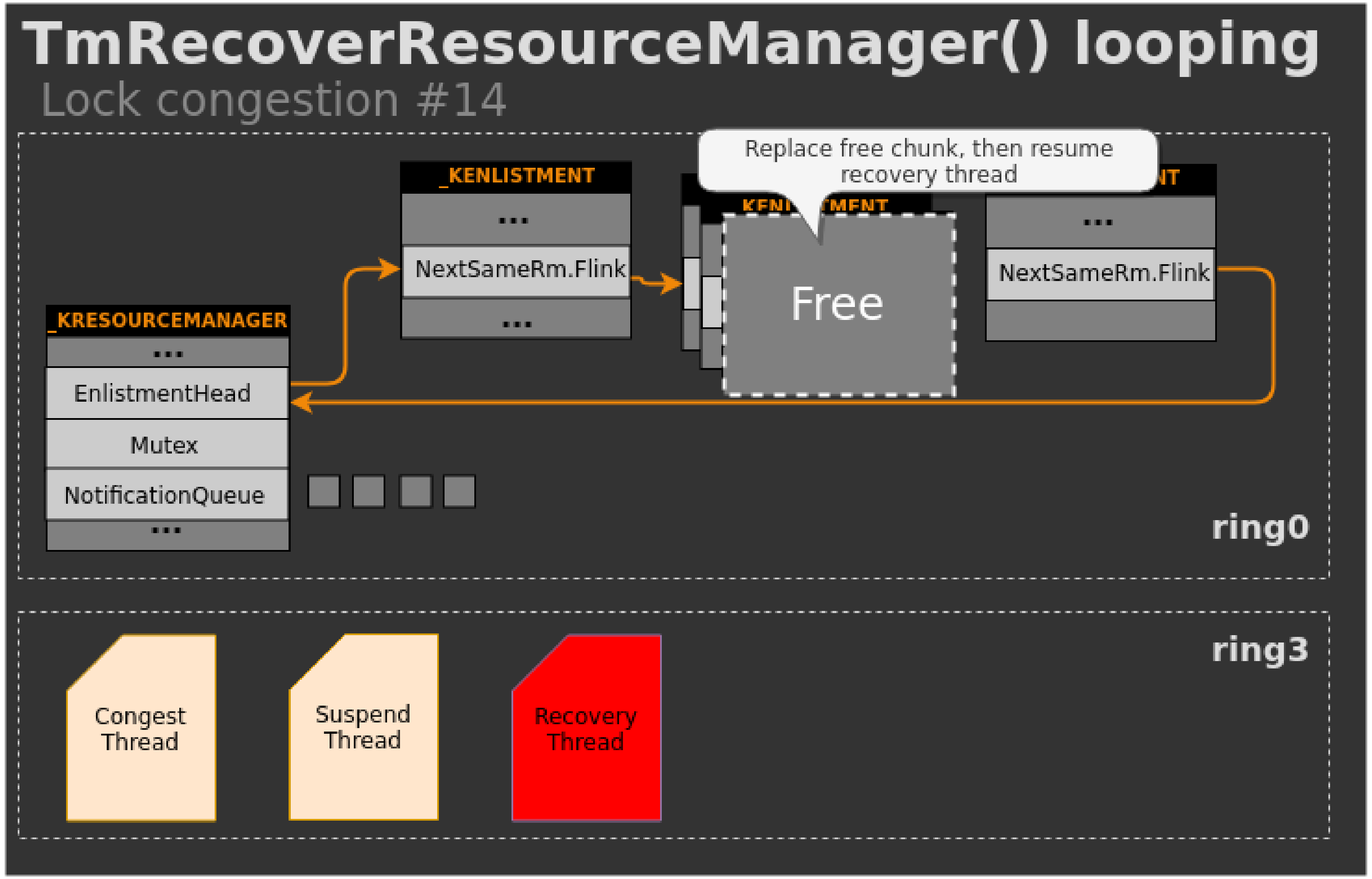
Lock congestion



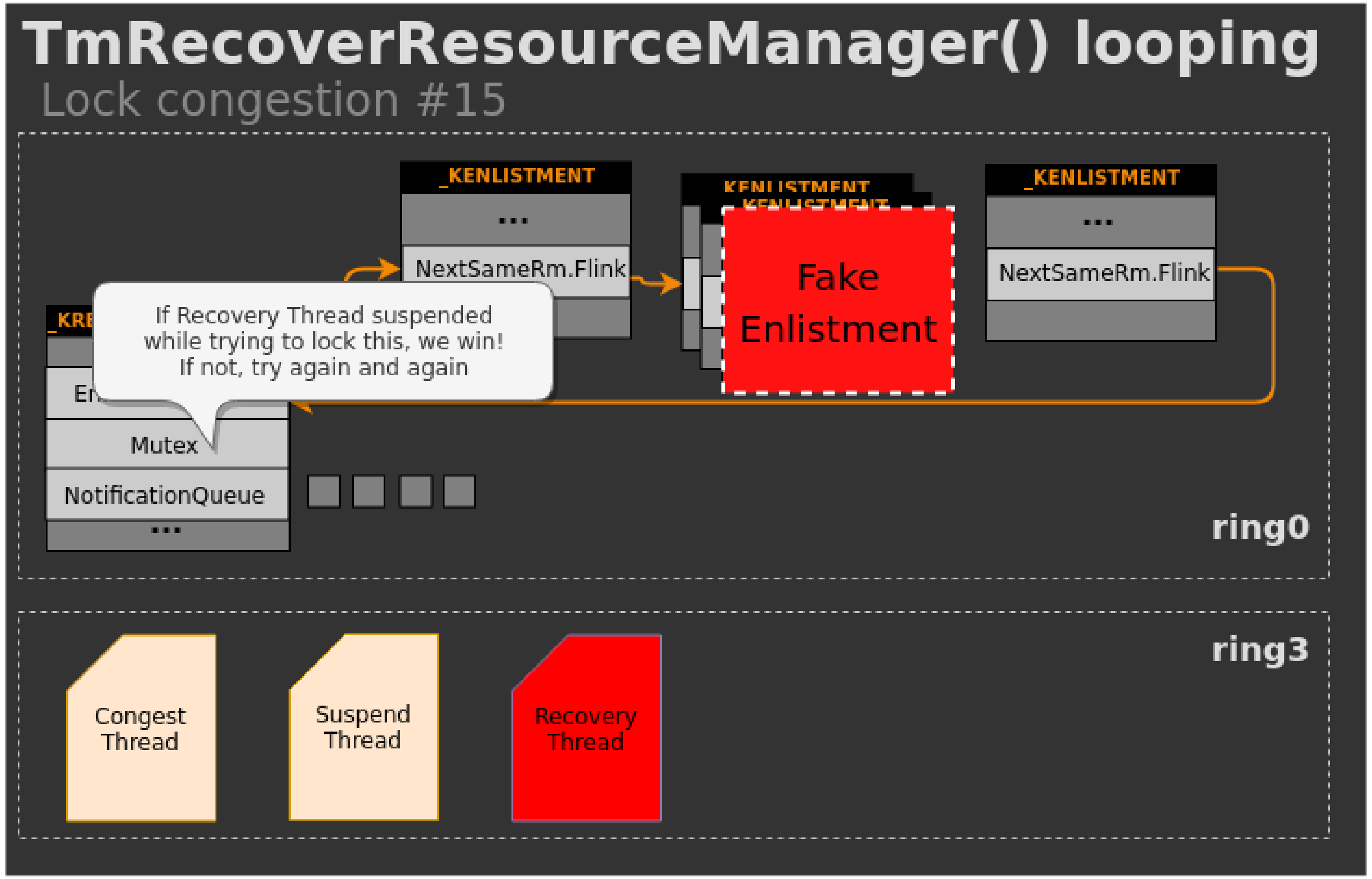
Lock congestion



Lock congestion



Lock congestion



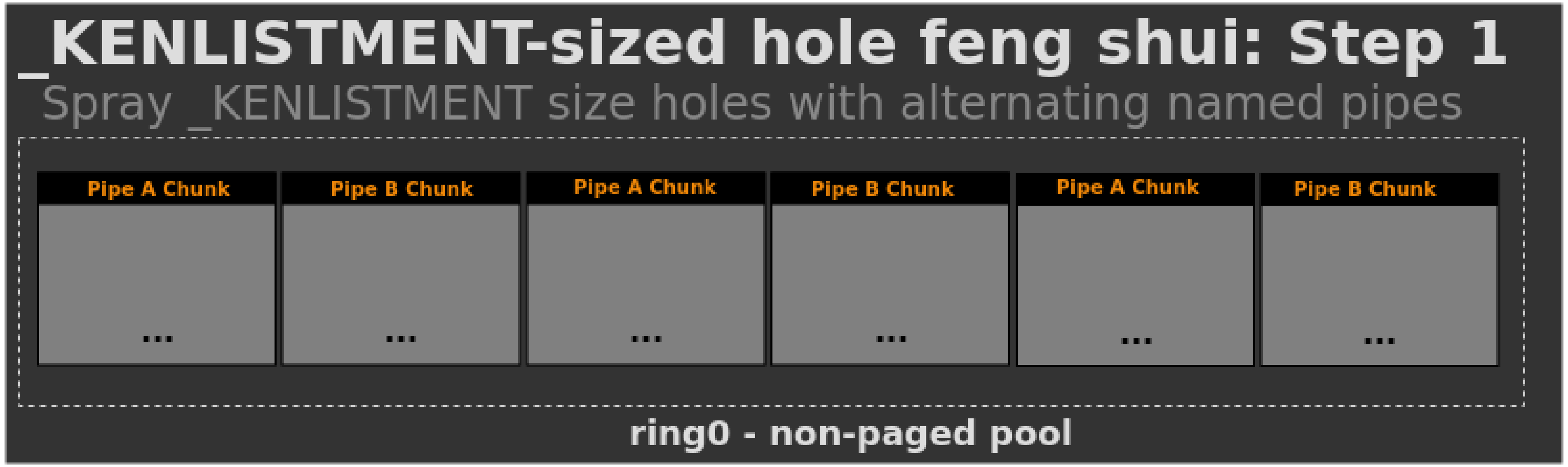
_KENLISTMENT replacement

- 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

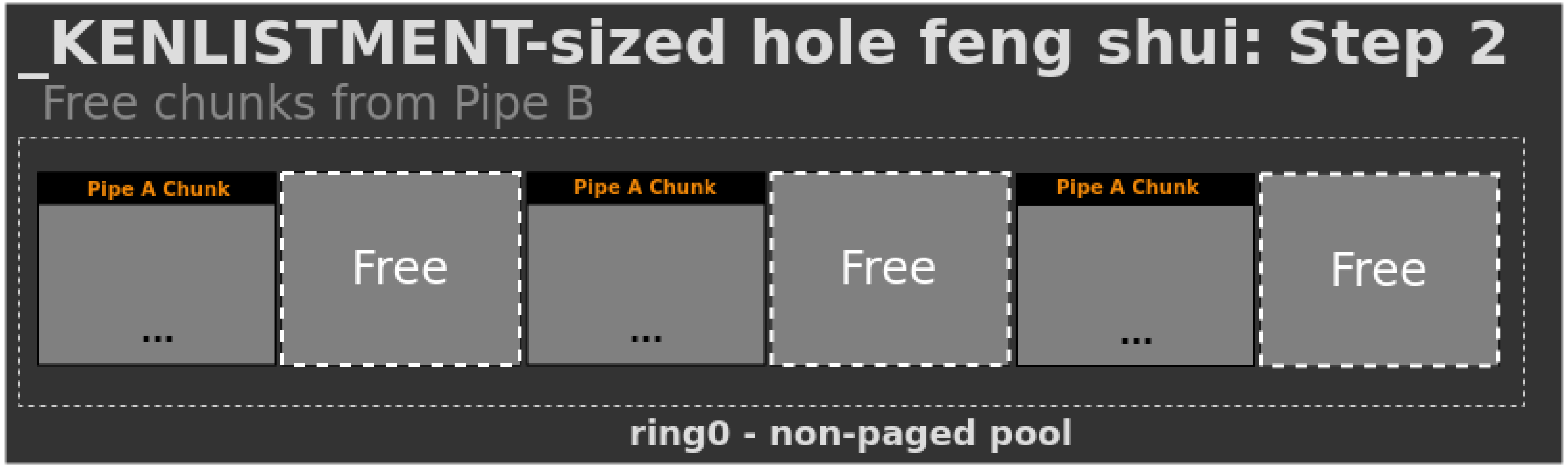
- Widely known, not too widely shared?
- Named Pipe writes allocate on non-paged pool
    - Code handled by `npfs.sys`
    - Tracked by `NpFr` pool tag
    - `!poolfind NpFr`
    - Persistent until other end of pipe reads data
    - Chunk free occurs when data is read
    - Allocates prefixed with an undocumented `DATA_ENTRY` structure
    - `DATA_ENTRY` layout has changed between Vista and Windows 10
    - Size of chunk is fully controlled
    - All data of chunk aside from `DATA_ENTRY` is fully controlled
    - ReactOS is best starting point
        - Reversing/hexdump for relevant changes

Feng shui layout #1

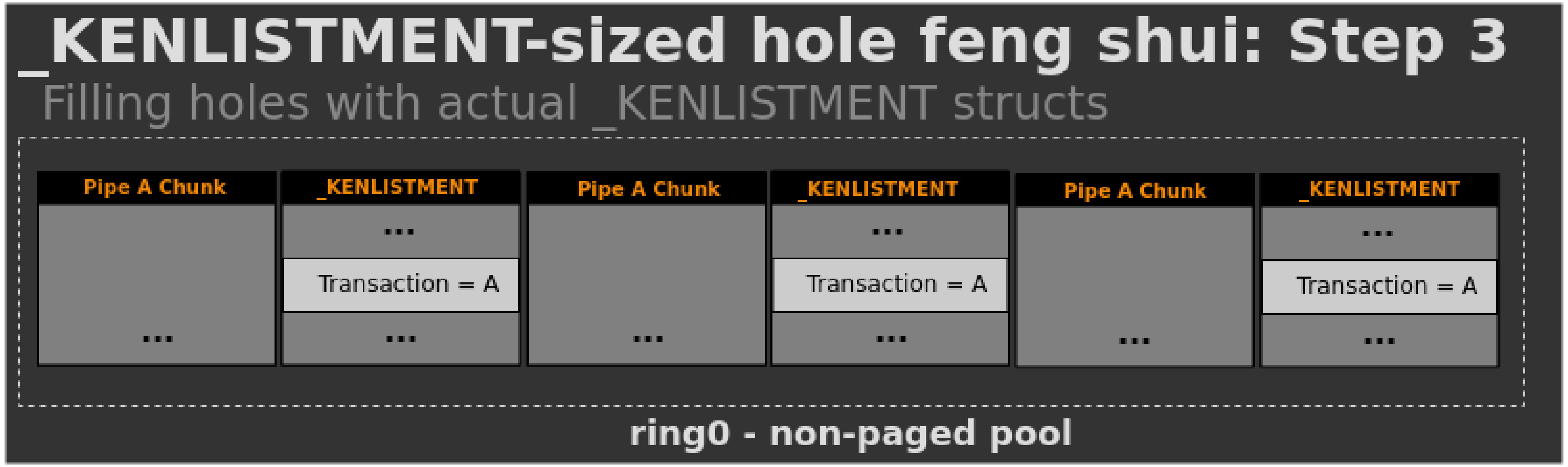

_KENLISTMENT-sized hole feng shui: Step 1
Spray _KENLISTMENT size holes with alternating named pipes
Pipe A Chunk
Pipe B Chunk
Pipe A Chunk
Pipe B Chunk
Pipe A Chunk
Pipe B Chunk
...
ring0 - non-paged pool

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

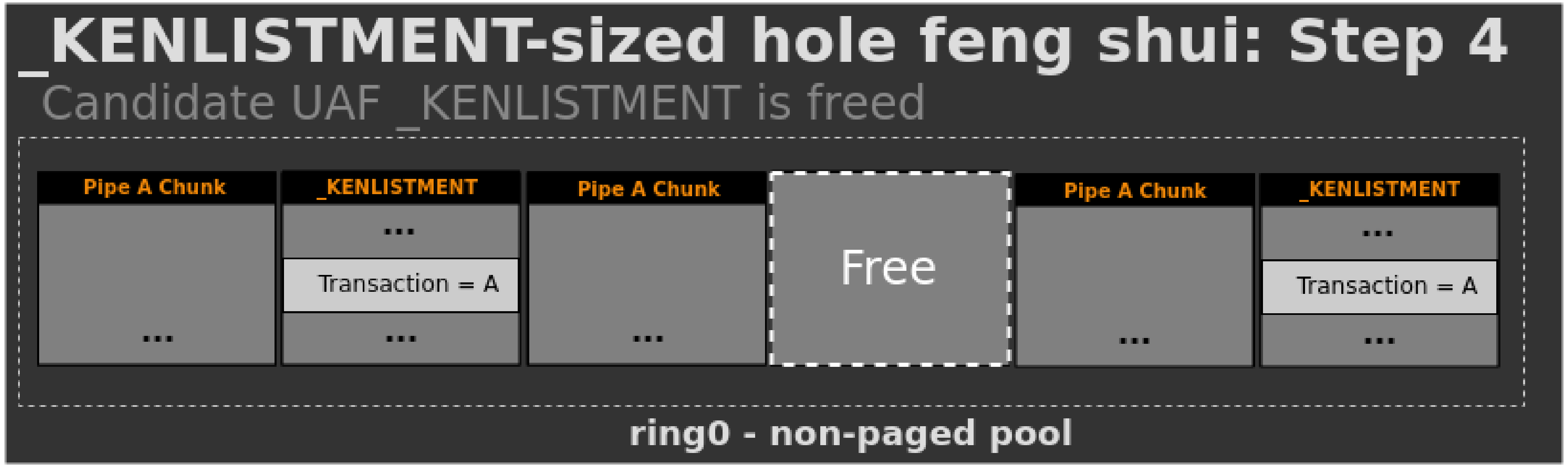
Feng shui layout #2



Feng shui layout #3

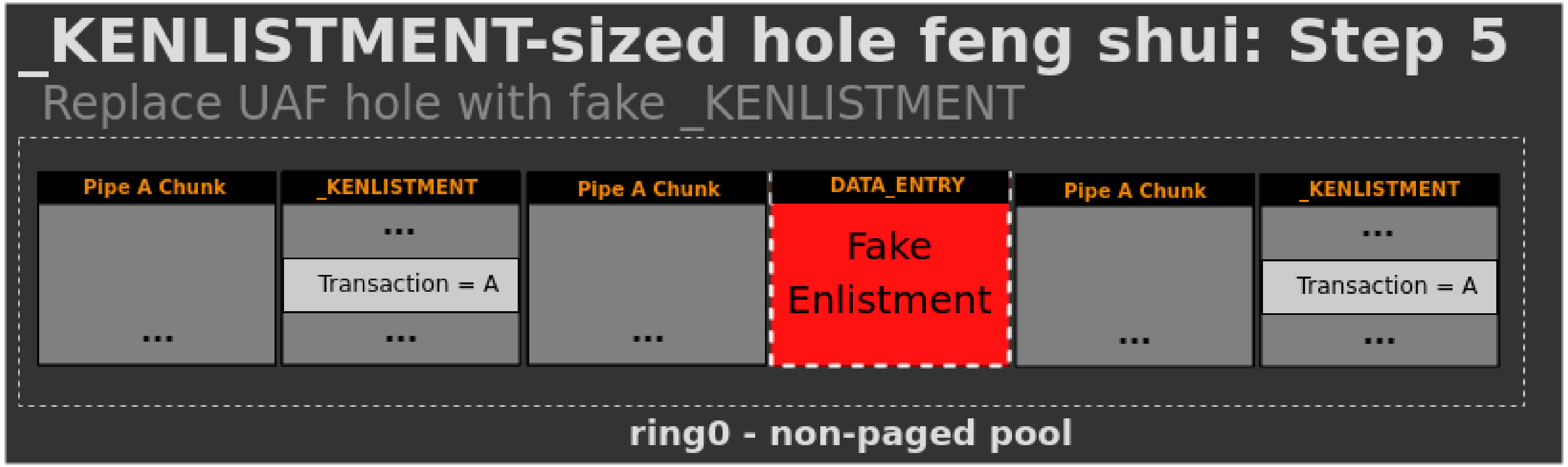


Feng shui layout #4

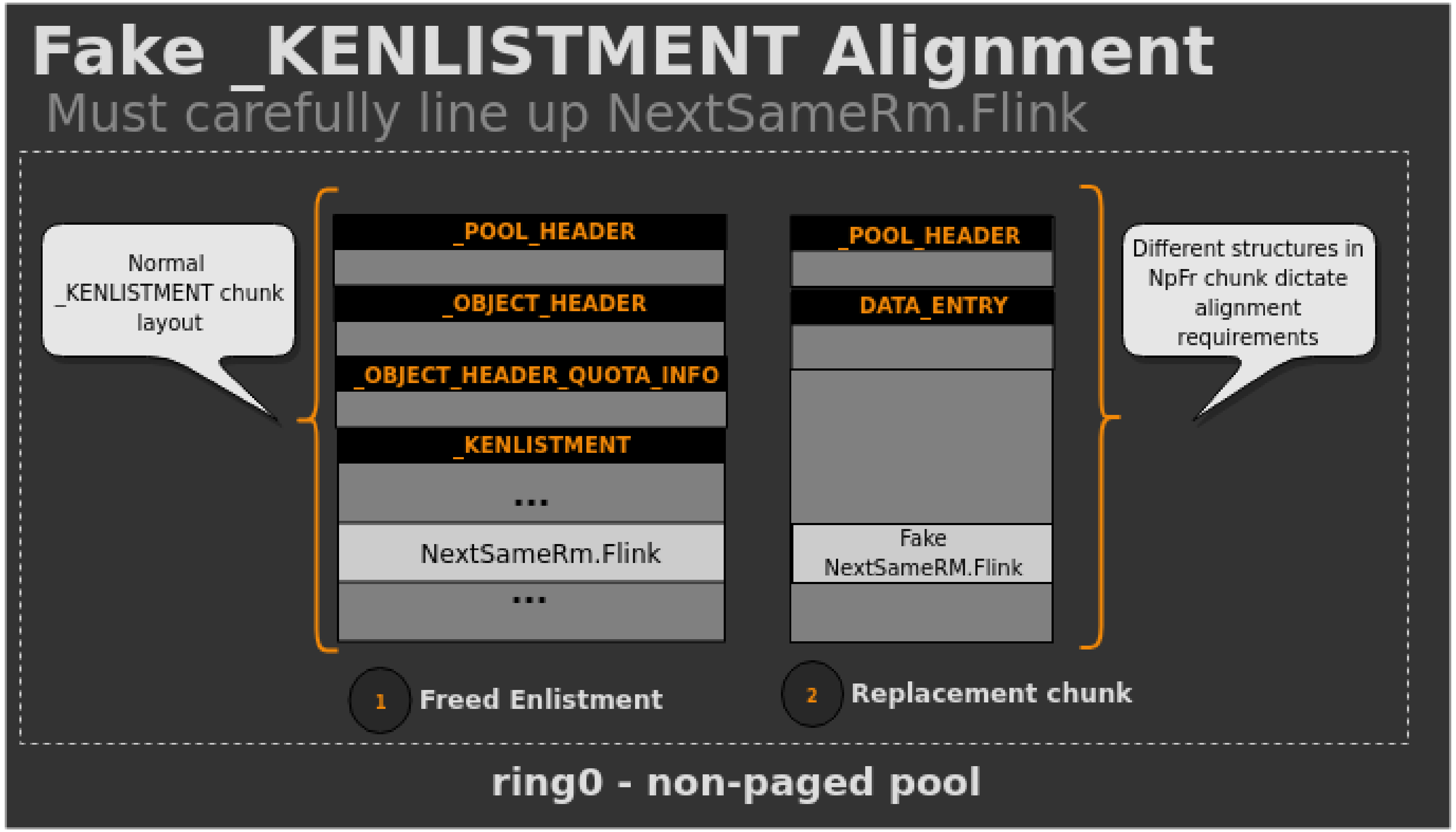


nccgroup

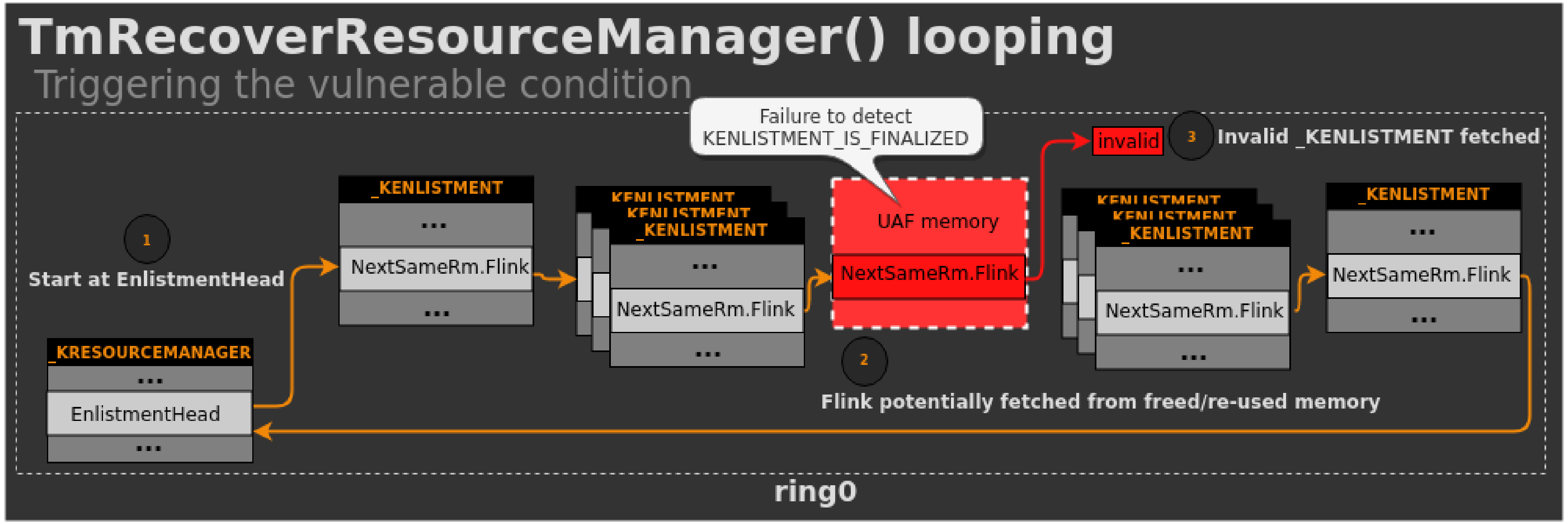
Feng shui layout #5



Faking a _KENLISTMENT with a named pipe chunk


Fake _KENLISTMENT Alignment
Must carefully line up NextSameRm.Flink
Normal _KENLISTMENT chunk layout
_POOL_HEADER
_OBJECT_HEADER
_OBJECT_HEADER_QUOTA_INFO
_KENLISTMENT
...
NextSameRm.Flink
...
_POOL_HEADER
DATA_ENTRY
Fake NextSameRM.Flink
Different structures in NpFr chunk dictate alignment requirements
1 Freed Enlistment
2 Replacement chunk
ring0 - non-paged pool

End result

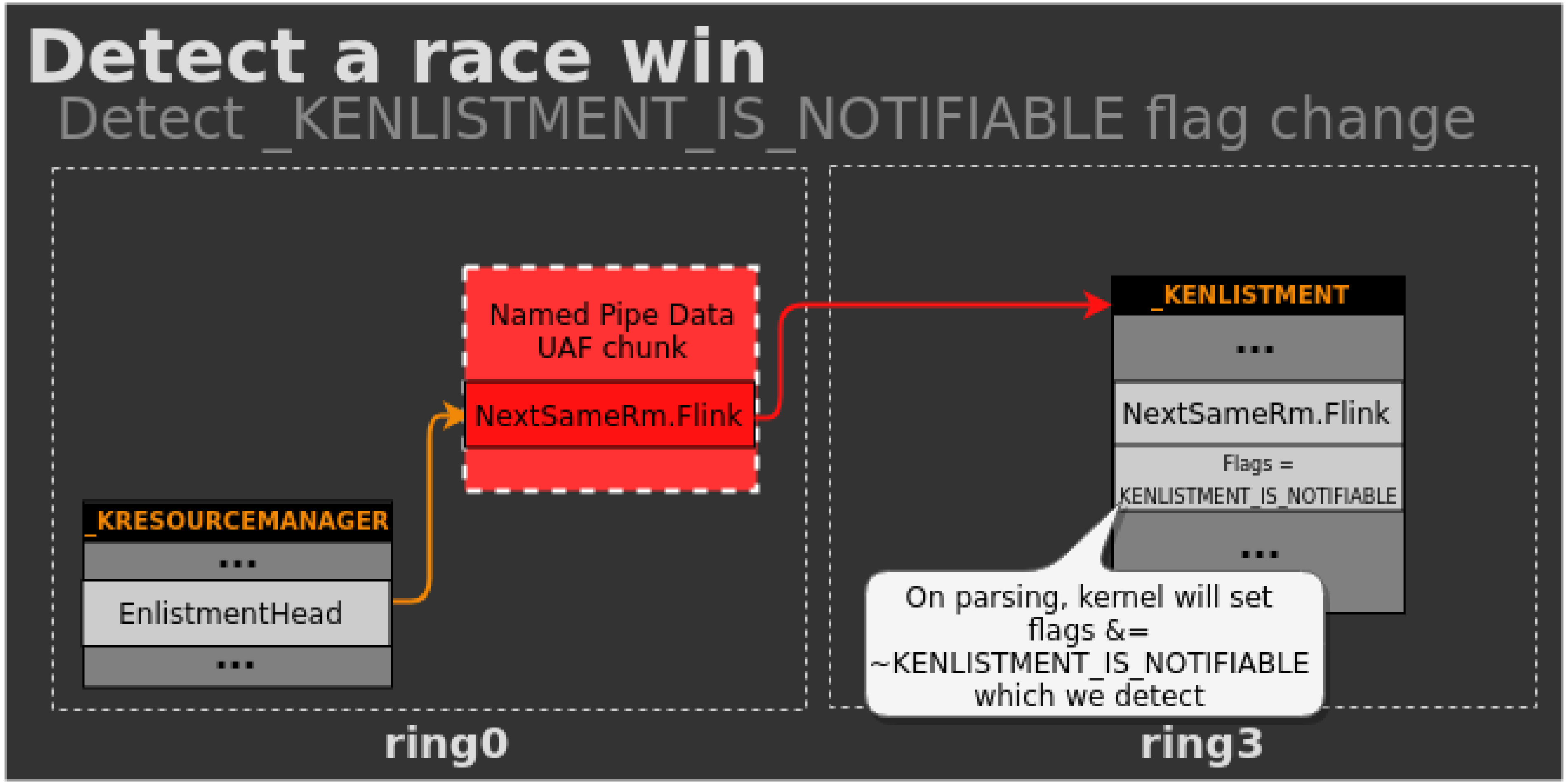


TmRecoverResourceManager() looping
Triggering the vulnerable condition
Failure to detect KENLISTMENT_IS_FINALIZED
invalid
3
Invalid _KENLISTMENT fetched
1
Start at EnlistmentHead
_KENLISTMENT
...
NextSameRm.Flink
...
KENLISTMENT
KENLISTMENT
_KENLISTMENT
NextSameRm.Flink
UAF memory
NextSameRm.Flink
2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

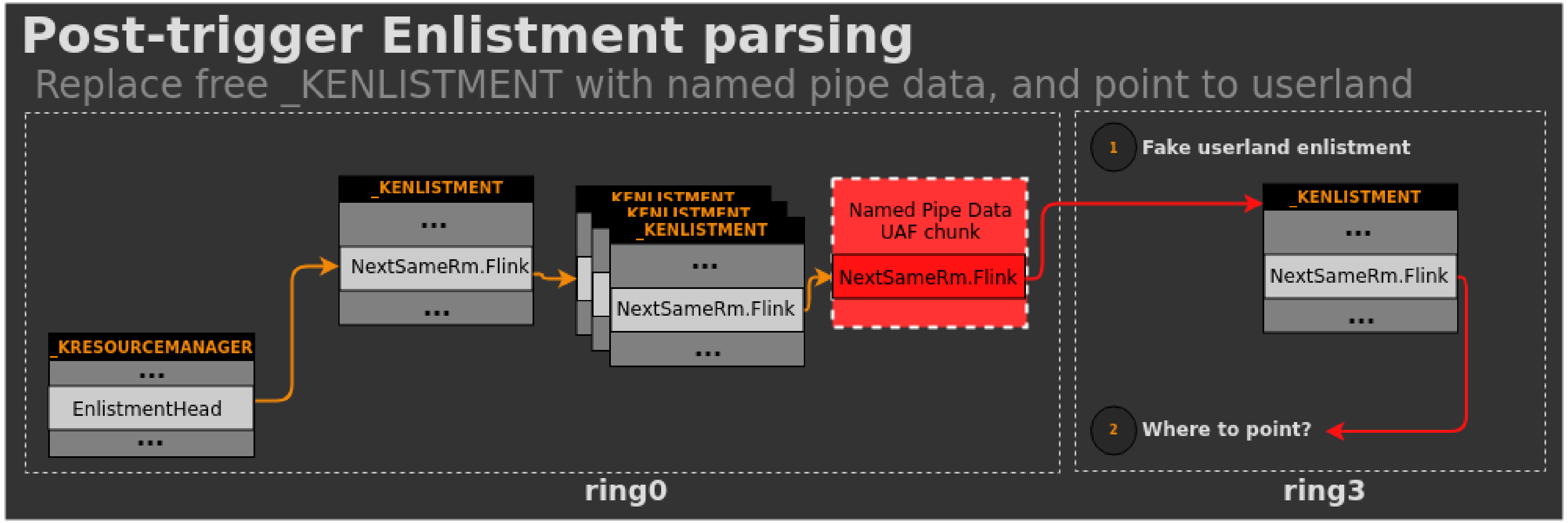
Detecting a race win

- How seize control of loop?
- 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 notifiable flag on userland enlistment
    - This modification in userland tells us we won!

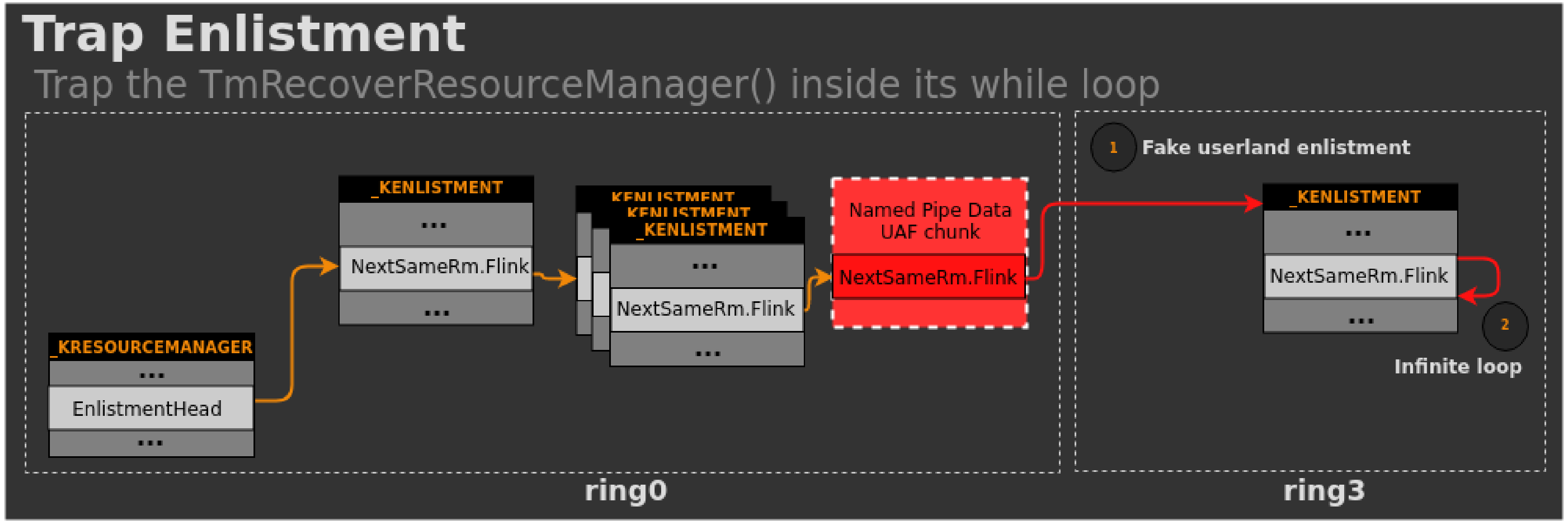
Detecting a race win


Detect a race win
Detect _KENLISTMENT_IS_NOTIFIABLE flag change
Named Pipe Data UAF chunk
NextSameRm.Flink
_KRESOURCEMANAGER
...
EnlistmentHead
...
_KENLISTMENT
...
NextSameRm.Flink
Flags = KENLISTMENT_IS_NOTIFIABLE
...
On parsing, kernel will set flags &= ~KENLISTMENT_IS_NOTIFIABLE which we detect
ring0
ring3

Now what?

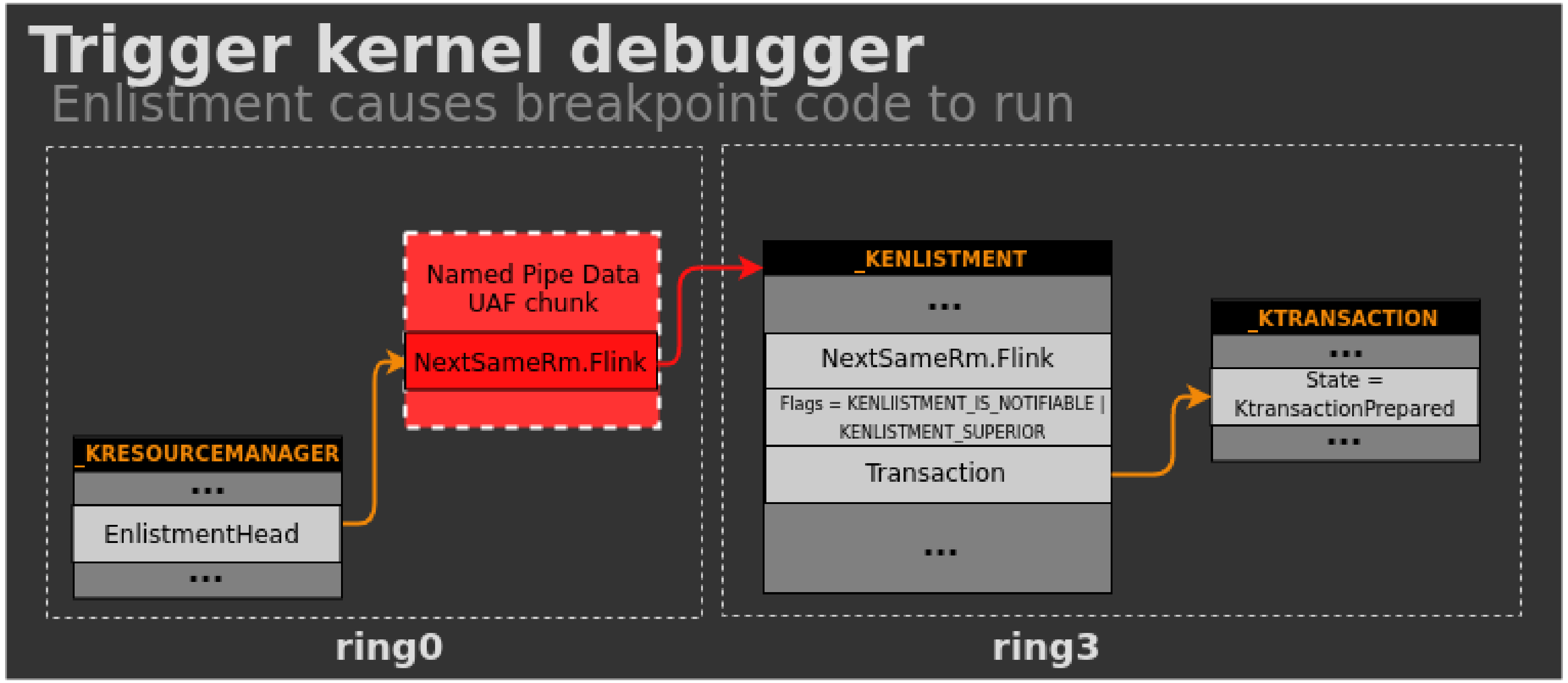

Post-trigger Enlistment parsing
Replace free _KENLISTMENT with named pipe data, and point to userland
_KRESOURCEMANAGER
...
EnlistmentHead
...
_KENLISTMENT
...
NextSameRm.Flink
...
_KENLISTMENT
_KENLISTMENT
_KENLISTMENT
...
NextSameRm.Flink
...
Named Pipe Data UAF chunk
NextSameRm.Flink
1 Fake userland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

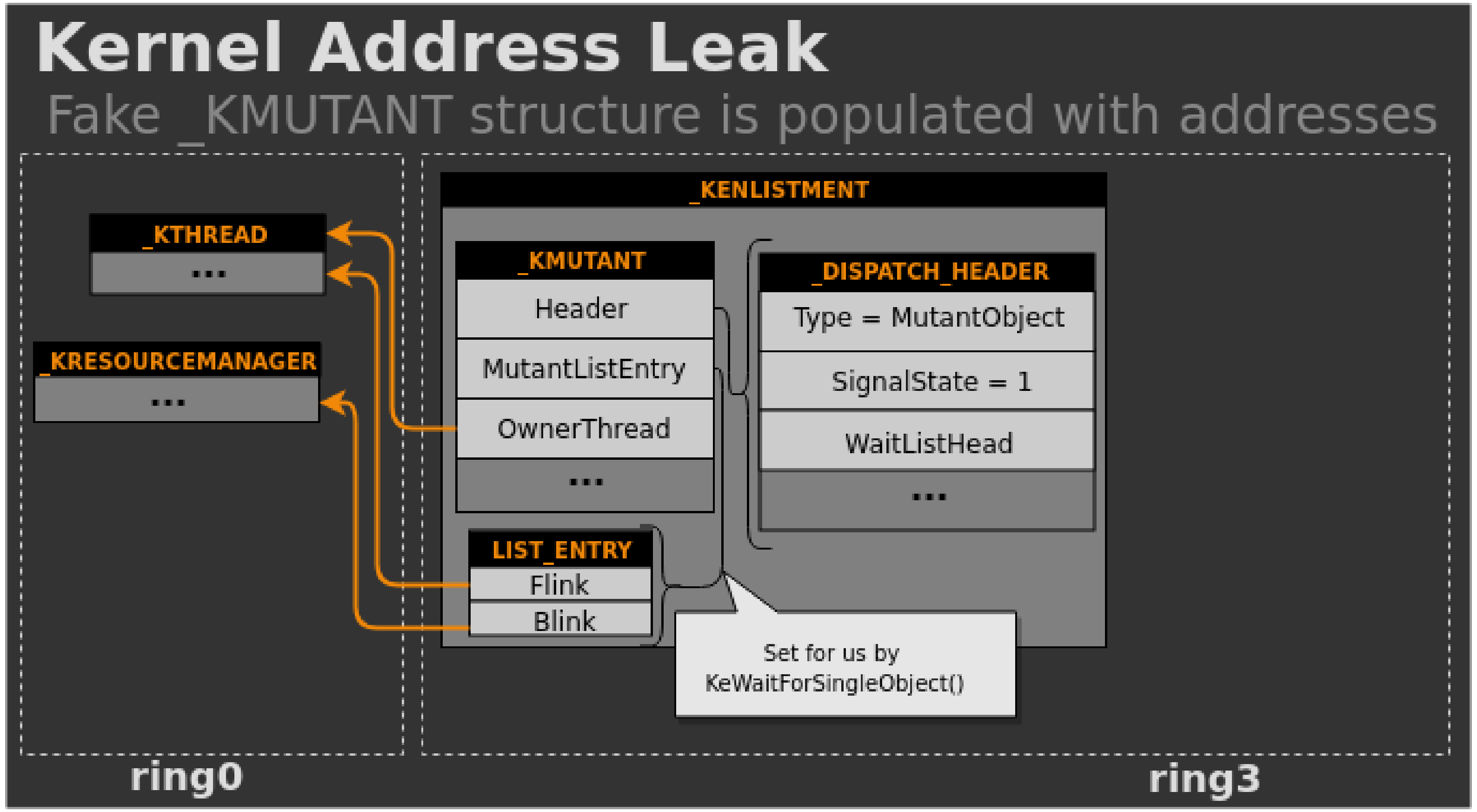
Debugging a race win?



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

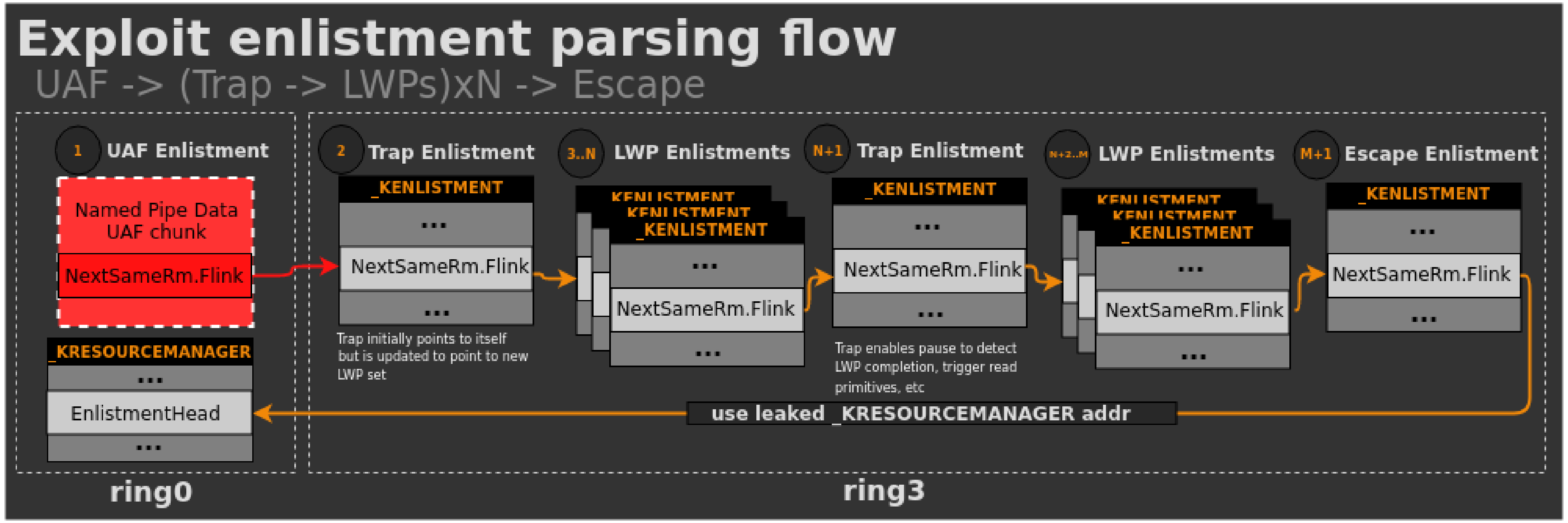


- Thank you `KeWaitForSingleObject()`

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()` seems 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..

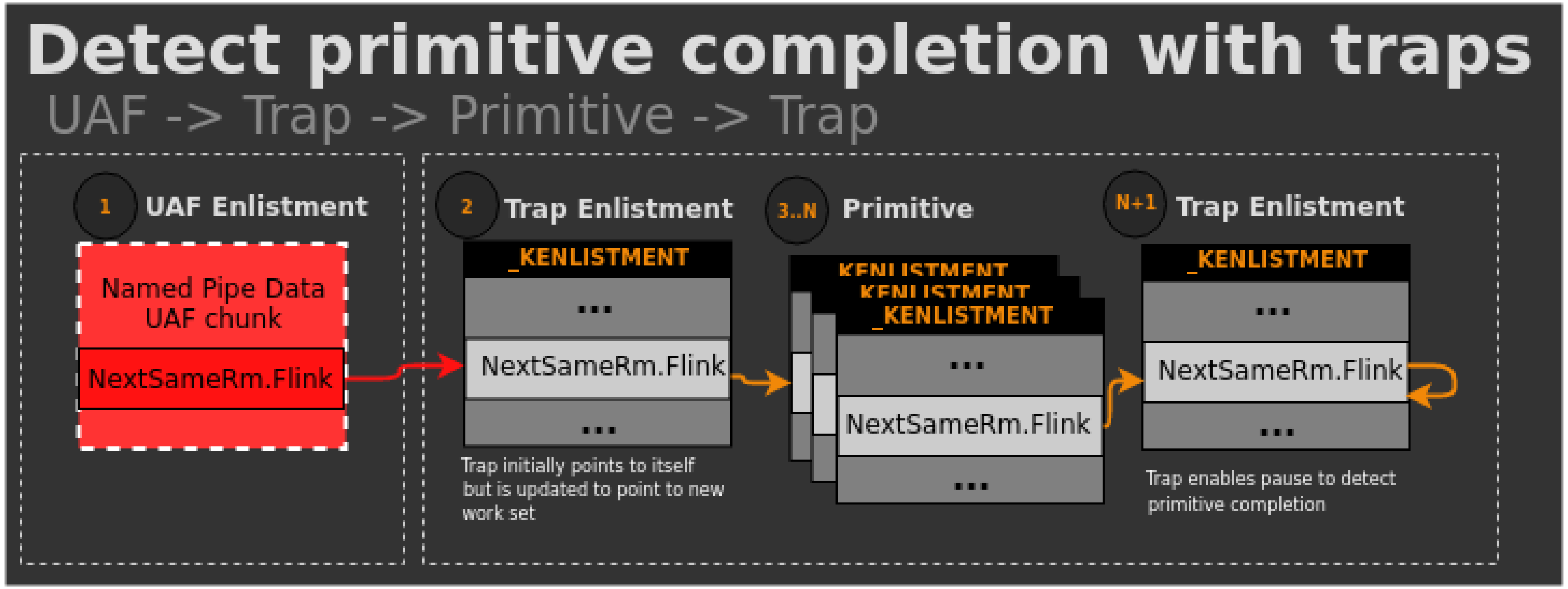
Arbitrary increment primitive

- `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

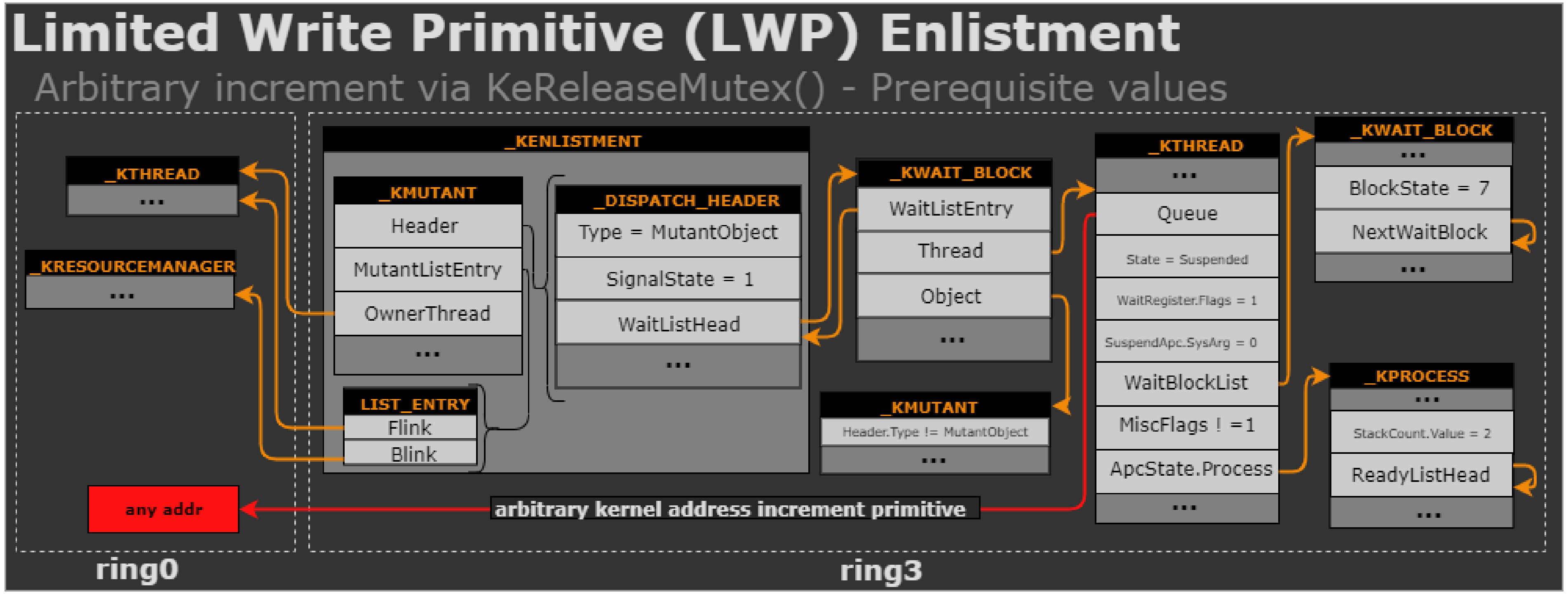
Repeatable arbitrary address increment

- 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


Detect primitive completion with traps
UAF -> Trap -> Primitive -> Trap
1 UAF Enlistment
Named Pipe Data UAF chunk
NextSameRm.Flink
2 Trap Enlistment
_KENLISTMENT
...
NextSameRm.Flink
...
Trap initially points to itself but is updated to point to new work set
3..N Primitive
_KENLISTMENT
_KENLISTMENT
_KENLISTMENT
...
NextSameRm.Flink
...
N+1 Trap Enlistment
_KENLISTMENT
...
NextSameRm.Flink
...
Trap enables pause to detect primitive completion

What does our increment primitive look like?



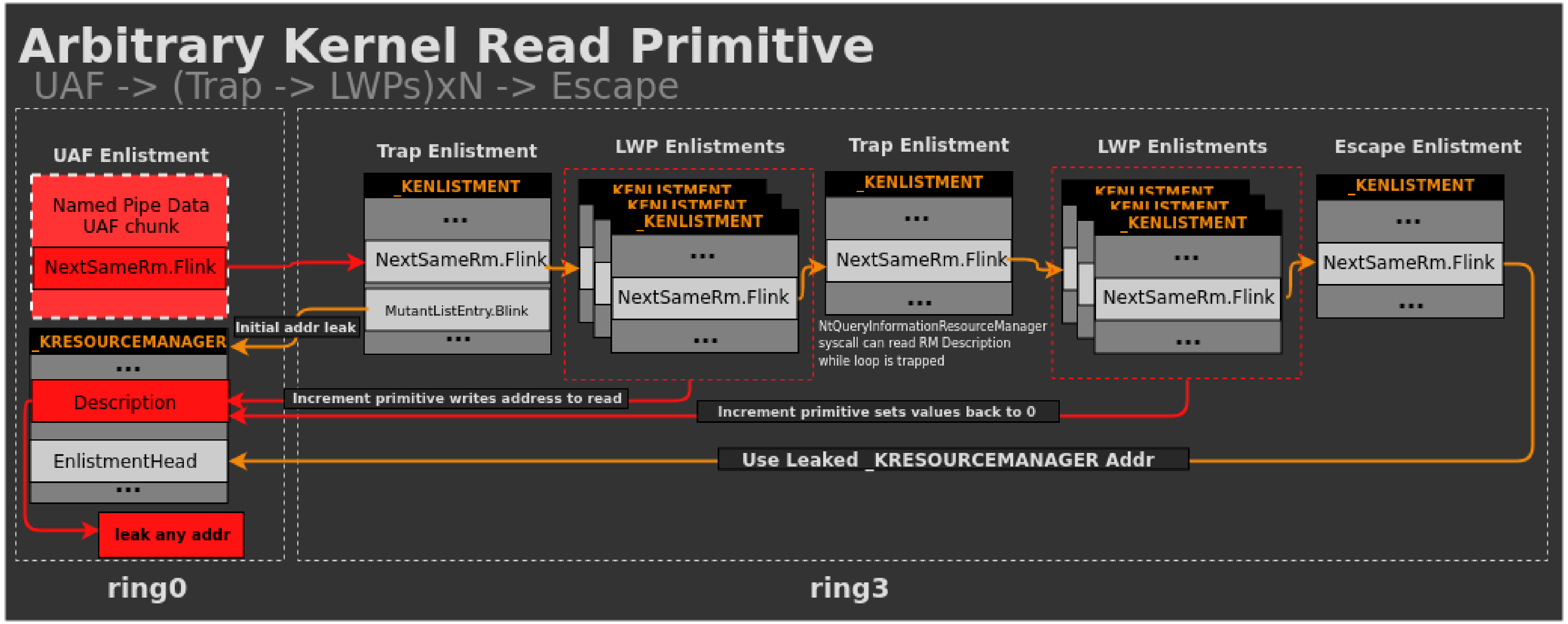
- Lots of constraints
- Some requirements change across OS versions

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 not set a `Description` field
- Means we know `_UNICODE_STRING` `Length` and `Name`
- Point anywhere we want
- Call `NtQueryResourceManager` syscall to get description
- Rinse and repeat

What does our read primitive look like?



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 x86/x64

- Use read primitive to find SYSTEM process token
- Patch process _KPROCESS struct
- Bypassing kernel CFG wasn't investigated
    - 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
  - 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
  - Actually quite happy in the end we never saw it!
- Most interesting highlight
  - 0day exploit used multiple different approaches from us

Bonus - Race winning

- `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
  - Negatives
    - Doesn't actually work on all OS versions (Vista x64, Vista/7 x86)
    - 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
    - Possibly first in-the-wild use?
  - Negatives
    - Doesn't really work on x86 (we will explain why in blog series)

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 soon after POC2019

Questions?

- Aaron Adams - @fidgetingbits, aaron.adams@nccgroup.com