HITCON CTF 2020 Lucifer

Windows kernel challenge

Angelboy

angelboy@chroot.org

@scwuaptx

Description

Environment

- Windows 10 Pro 20H2 (update 到最新)
  - Load Lucifer.sys in test mode
  - Normal user account
- Run
  - It will run C:\ctf\cmd.exe as Low integrity
  - You can nc ip port to connect the service
  - You can use curl to download your binary in %TEMP%\Low or c:\ctf\tmp.

Description

Goal

- Read the flag in C:\flag.txt (readable only by SYSTEM)

Description

Lucifer.sys

- Just a storage
  - Create
  - Add
  - Get
  - Release

Description

Lucifer.sys

- Request structure for operation

```
struct request {
    UINT64 idx;
    UINT64 magic;
    UINT64 data;
};
```

Description

Lucifer.sys

- Create
  - Create buffer for storage.

```
if(!Lucifer_secret)
    Lucifer_secret = ExAllocatePoolWithTag(NonPagedPoolNx, sizeof(UINT64) * MAX, (ULONG)0x6963754c);
if (!Lucifer_secret) {
    Status = STATUS_NO_MEMORY ;
}
RtlZeroMemory(Lucifer_secret, sizeof(UINT64) * MAX);
```

Description

Lucifer.sys

- Add
  - Add a value to storage

```
__try {
    if (!Lucifer_secret || r->magic != 0xdadaddaa) {
        Status = STATUS_ACCESS_DENIED;
    }
    Lucifer_secret[r->idx] = r->data;
}
```

Vulnerability

Lucifer.sys

- Get
  - Get a value from storage

```
__try {
    if (!Lucifer_secret || r->magic != 0xdadaddaa) {
        Status = STATUS_ACCESS_DENIED;
    }
    if (r->idx < MAX) {
        r->data = Lucifer_secret[r->idx];
        r->magic = 0xdeadbeef;
    }
}
```

Description

Lucifer.sys

- Release
  - Release the buffer

```
__try {
    if (Lucifer_secret)
        ExFreePoolWithTag(Lucifer_secret, (ULONG)0x6963754c);
    Lucifer_secret = NULL;
}
```

Vulnerability
Out of bound Write

- It does not check index when you add a value to storage

```
__try {
    if (!Lucifer_secret || r->magic != 0xdadaddaa) {
        Status = STATUS_ACCESS_DENIED;
    }
    Lucifer_secret[r->idx] = r->data;
}
```

Exploitation
Challenge

- Because we running the exe in the Low integrity
  - Very hard to leak address in kernel space
- We should find a way to do arbitrary memory read first
  - Not easy to use the vulnerability
    - Hard to locate the buffer
    - Hard to manipulate the memory layout in windows kernel
      - Kernel allocates memory all the time

Exploitation

Find a kernel object to spray

- In NonPagedpoolnjx
  - Alex Ionescu ’s blog
    - Named pipe queue
      - https://alex-ionescu.com/?p=231
      - https://docs.microsoft.com/en-us/windows/win32/api/namedpipeapi/nf-namedpipeapi-createpipe

Exploitation

Find a kernel object to spray

```
    UCHAR payLoad[SIZE - 0x1C + 44];
    res = CreatePipe(&readPipe,
        &writePipe,
        NULL,
        sizeof(payLoad));
    if (res == FALSE) goto Cleanup;
    res = WriteFile(writePipe,
        payLoad,
        sizeof(payLoad),
        &resultLength,
        NULL);
    if (res == FALSE) goto Cleanup;
Cleanup:
    CloseHandle(writePipe);
    CloseHandle(readPipe);
```

ExAllocatePoolWithTag
(NonpagedPoolNx,sizeof(payload)+0x30,tag)

Exploitation

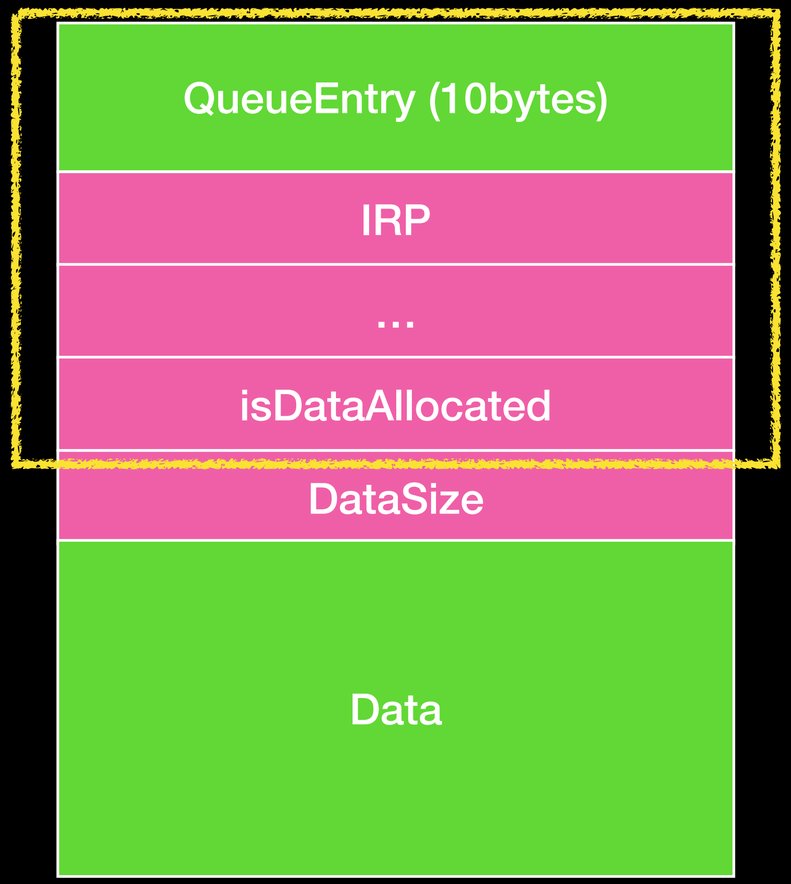
Find a kernel object to spray

- PipeQueueEntry
  - It will be created when we use write data to pipe every time.

```
struct  PipeQueueEntry
{
  LIST_ENTRY list;
  IRP *linkedIRP;
  __int64 SecurityClientContext;
  int isDataAllocated;
  int readbytes;
  int DataSize;
  int field_2C;
  char data[1];
};
```

Exploitation

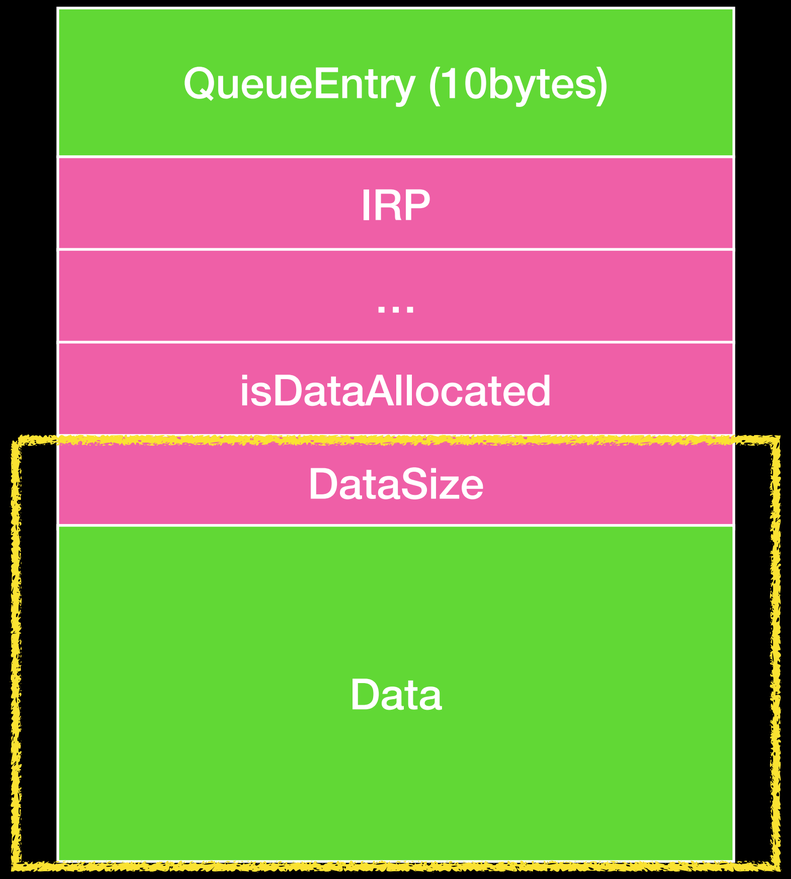
Find a kernel object to spray



- PipeQueueEntry
  - QueueEntry
    - Double linked list of data queue entry
  - isDataAllocated
    - The data is allocated from other buffer.
  - IRP
    - It’s used when isDataAllocated is 1

Exploitation

Find a kernel object to spray



- PipeQueueEntry
  - DataSize
    - Size of data queue (not include metadata)
    - Variable
  - Data
    - Data in queue
      - When we use writefile to queue. It will copy use data to the buffer.

Exploitation

Find a kernel object to spray

- PipeQueueEntry
  - We can use it to allocate any size memory in NonPagedPoolNX
  - It will be release when we close the handle of pipe
  - That is, it’s useful for heap spray.

Exploitation

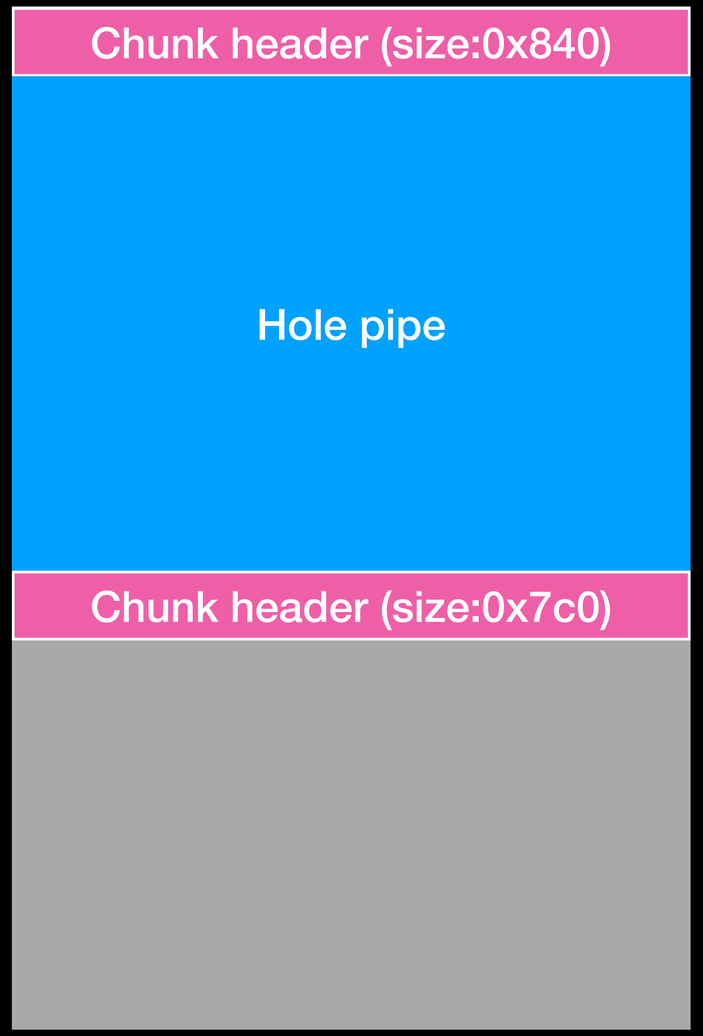
Use heap spray to prepare hole for storage

- The size of buffer of storage is 0x400
  - Variable Size Allocation
- Prepare four different pipes
  - hole_pipes : prepare_pipe(0x800-0x40, 0x10000)
  - fill_pipes : prepare_pipe(0x7d0 - 0x40, 0x10000)
  - evil_pipes : prepare_pipe(0x400, 0x10000)
  - target_pipes : prepare_pipe(0x3c0-0x40, 0x10000)

Exploitation

Use heap spray to prepare hole for storage

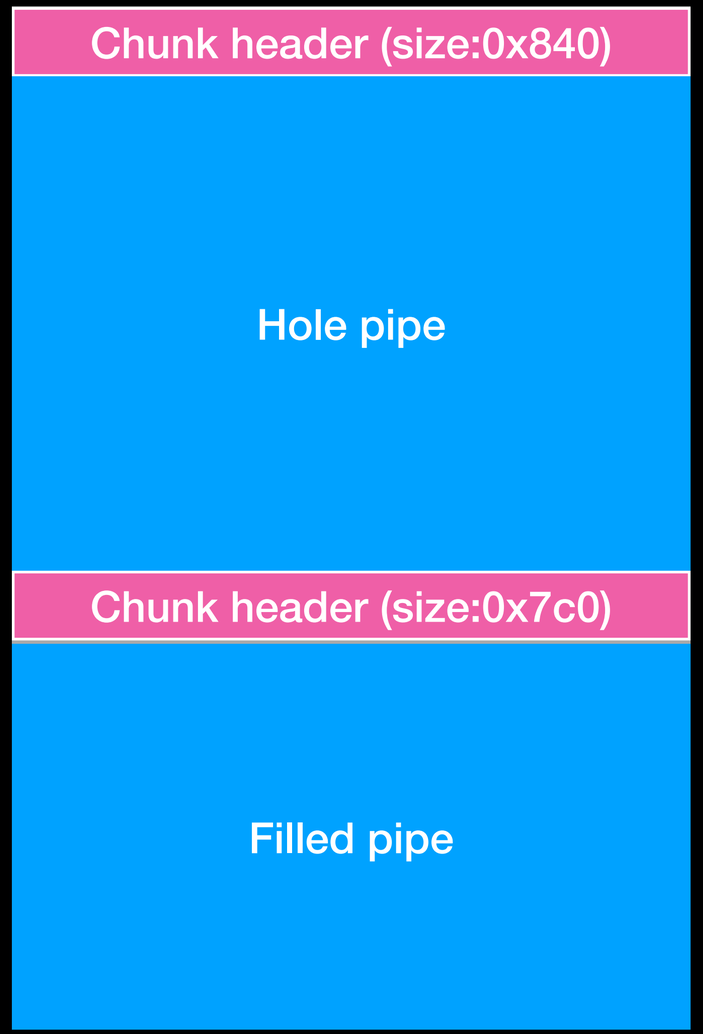
- Spray
  - Spary hole pipe
    - It will allocate 0x840



Exploitation

Use heap spray to prepare hole for storage

- Spray
  - Spary filled pipes
    - It will allocate 0x7c0



Exploitation

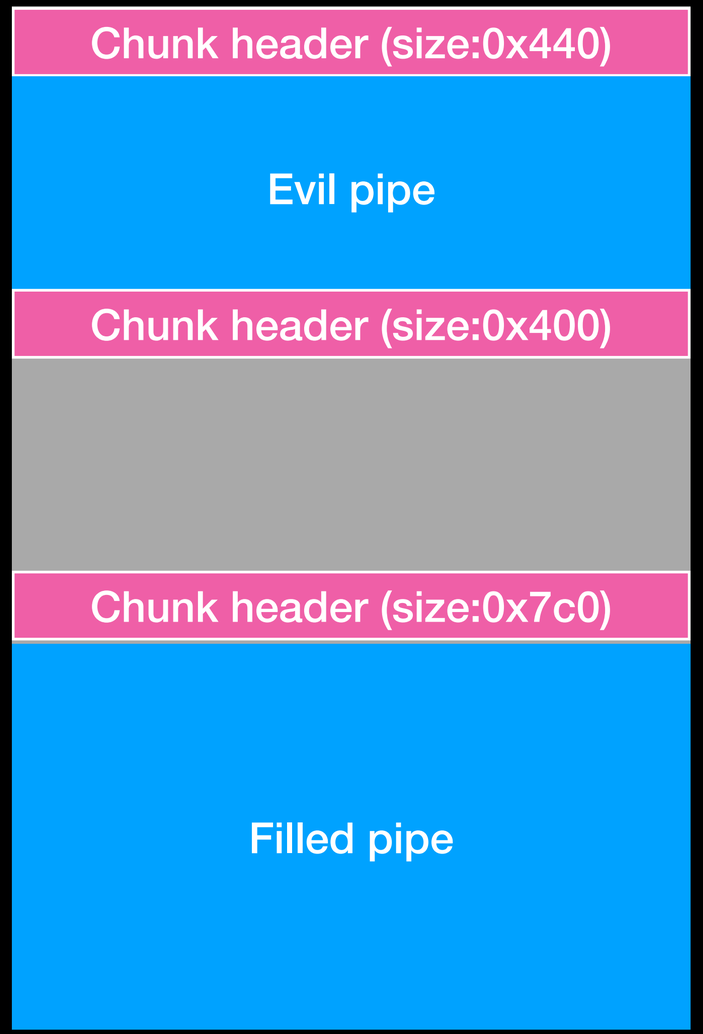
Use heap spray to prepare hole for storage

- Spray
  - Release hole pipe

Exploitation

Use heap spray to prepare hole for storage

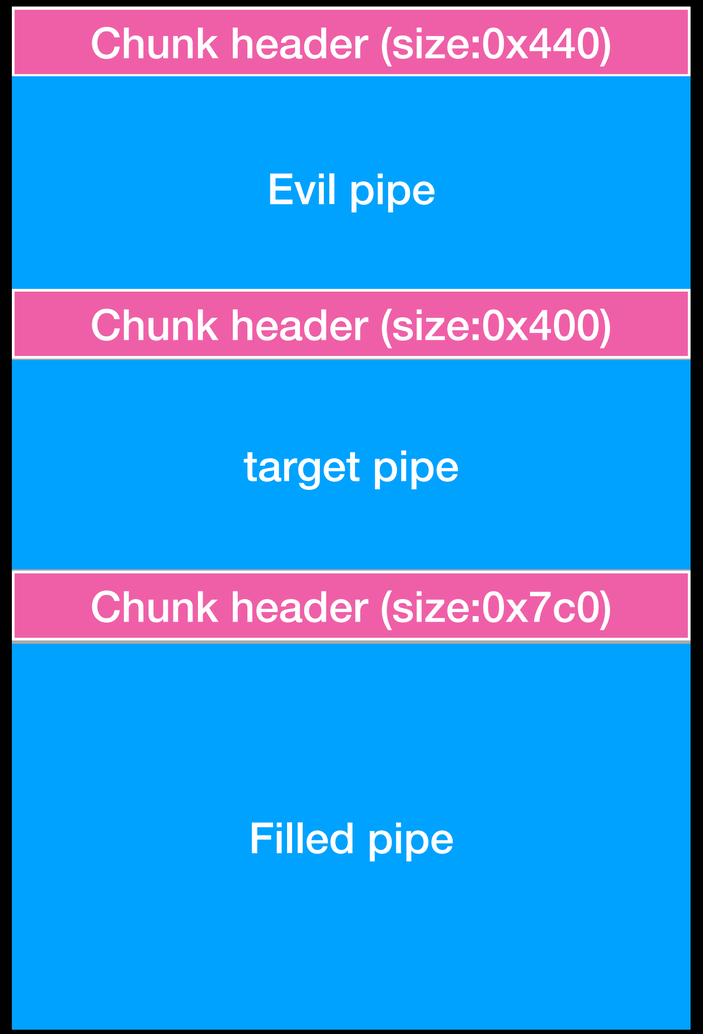
- Spray
  - Spary filled pipes
    - It will allocate 0x440
      - We use it to create hole for storage



Exploitation

Use heap spray to prepare hole for storage

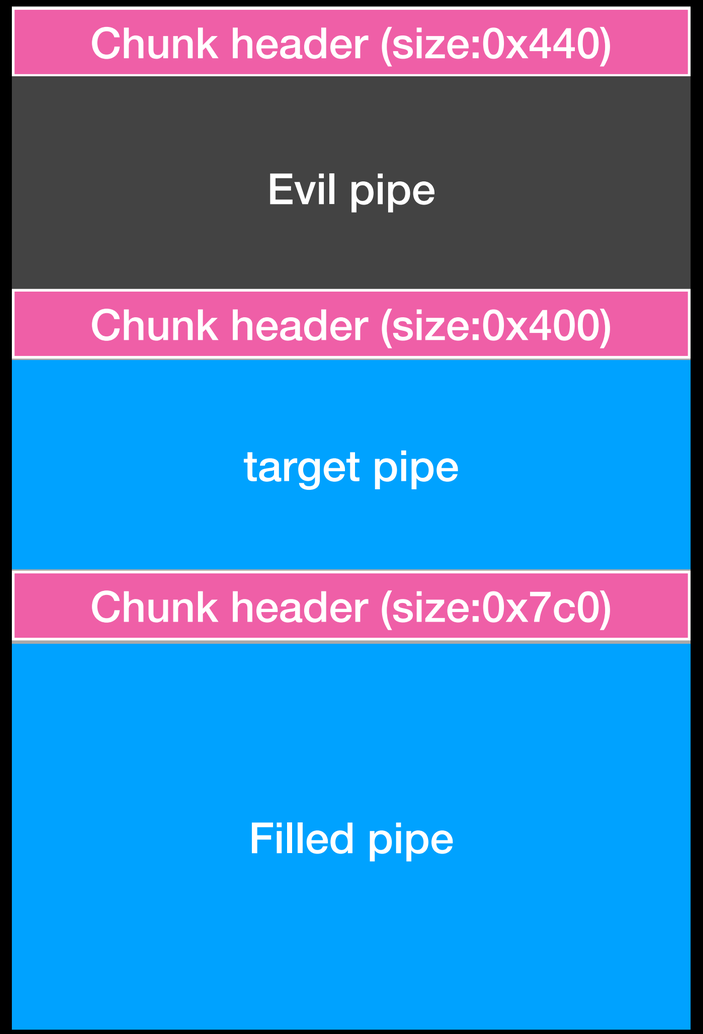
- Spray
  - Spary target pipes
    - It will allocate 0x400
      - We will write something to it



Exploitation

Use heap spray to prepare hole for storage

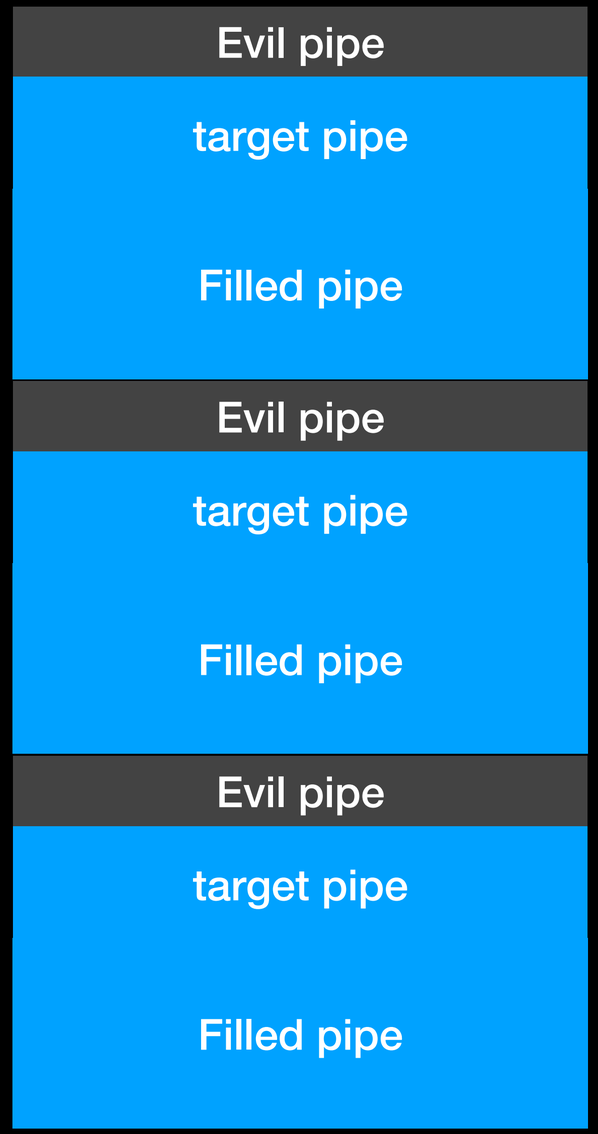
- Create hole for buffer of storage
  - Release many evil pipe



Exploitation

Use heap spray to prepare hole for storage

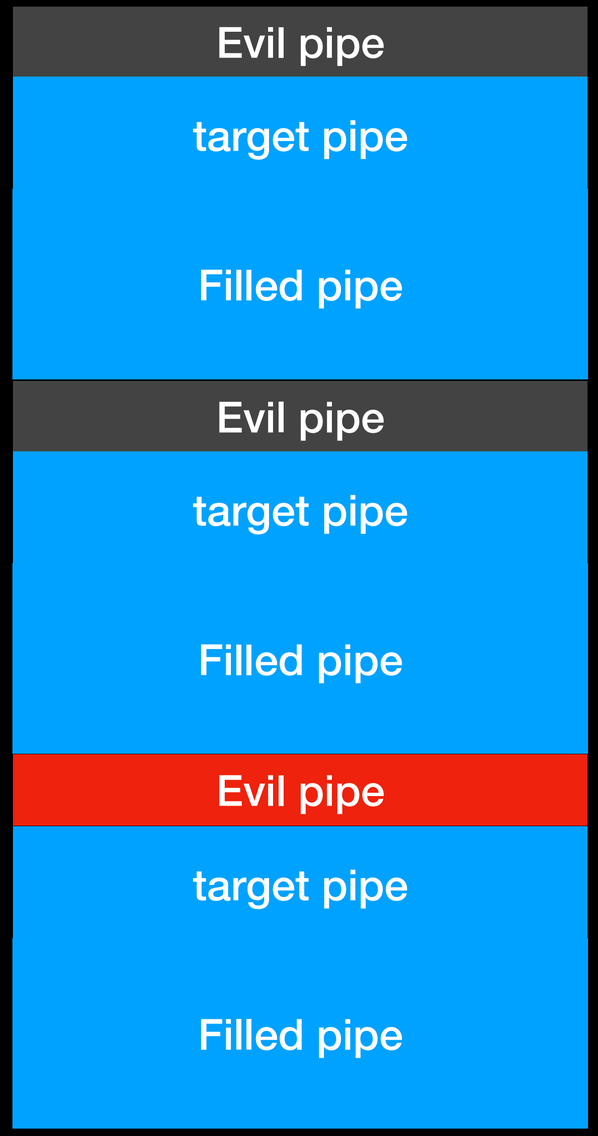
- We have many hole in NonPagedPool Now



Exploitation

Use heap spray to prepare hole for storage

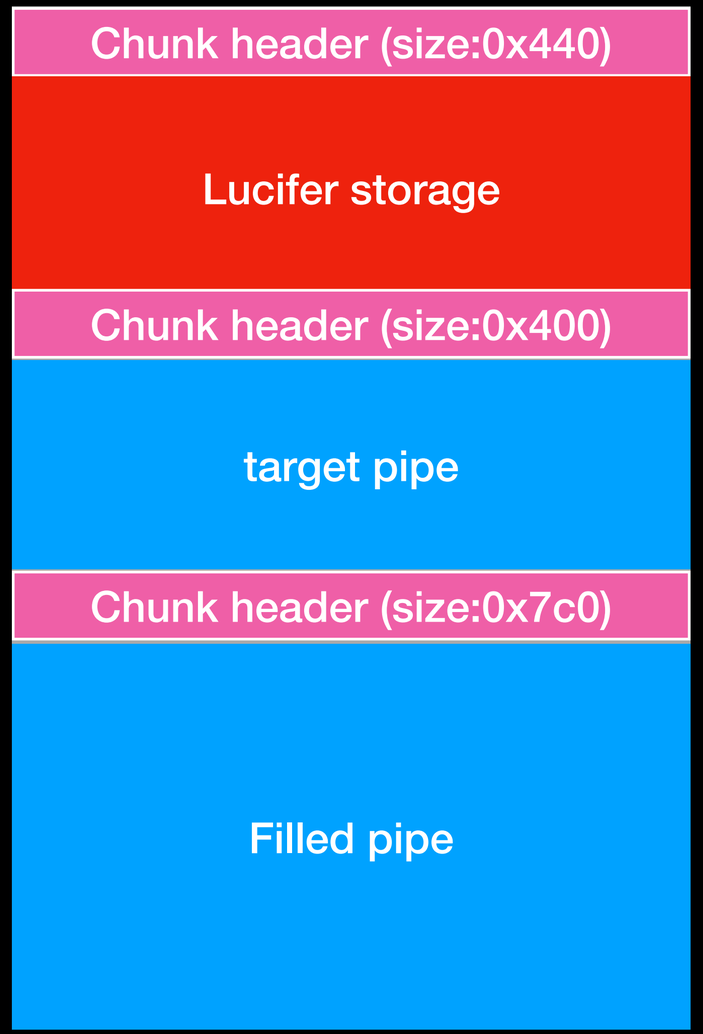
- We have many hole in NonPagedPool Now
  - Create buffer for storage



Exploitation

Use heap spray to prepare hole for storage

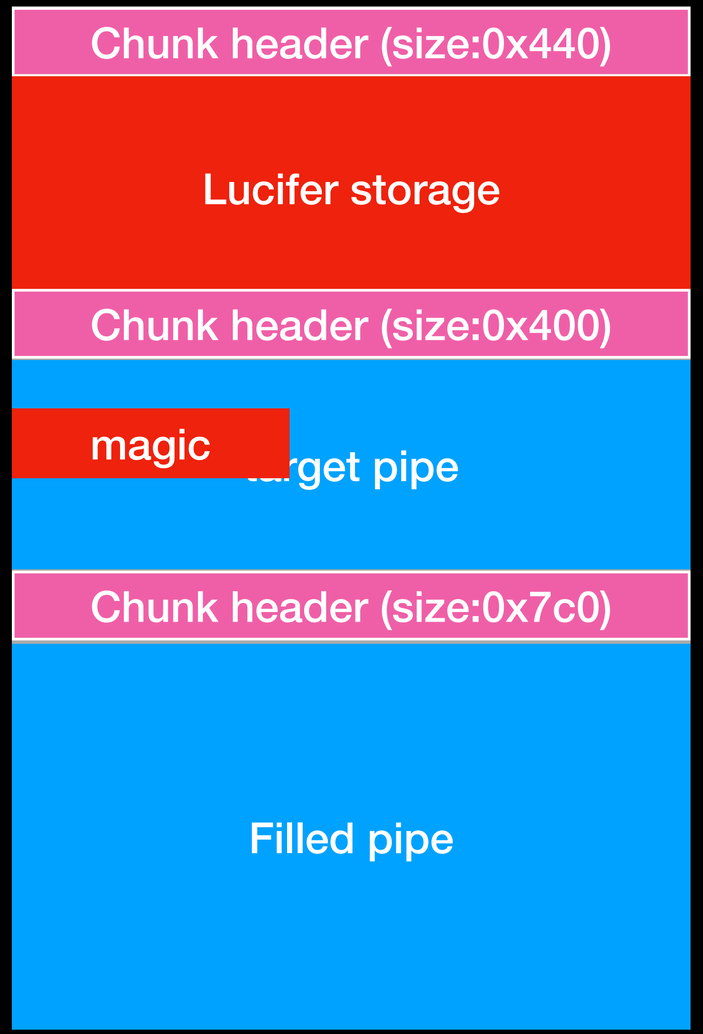
- We have many hole in NonPagedPool Now
  - Create buffer for storage



Exploitation

Find the target data queue behind storage

- We can use the out of bound write to write a magic to data queue of target pipe
  - lucifer_add(144, magic)
- Then traverse all of target pipe
  - Read 8 bytes from every target pipe
    - Check whether the value is same as magic



Exploitation

Arbitrary memory reading

- Let’s back to PipeQueueEntry.
  - When we use readfile to read data from queue, it will data from queue->data
  - But if queue->isDataAllocated is 1, it will read from IRP->systembuffer

```
if ( queue->isDataAllocated == 1 )
  databuffer = (char *)queue->linkedIRP->AssociatedIrp.SystemBuffer;
else
  databuffer = queue->data;
```

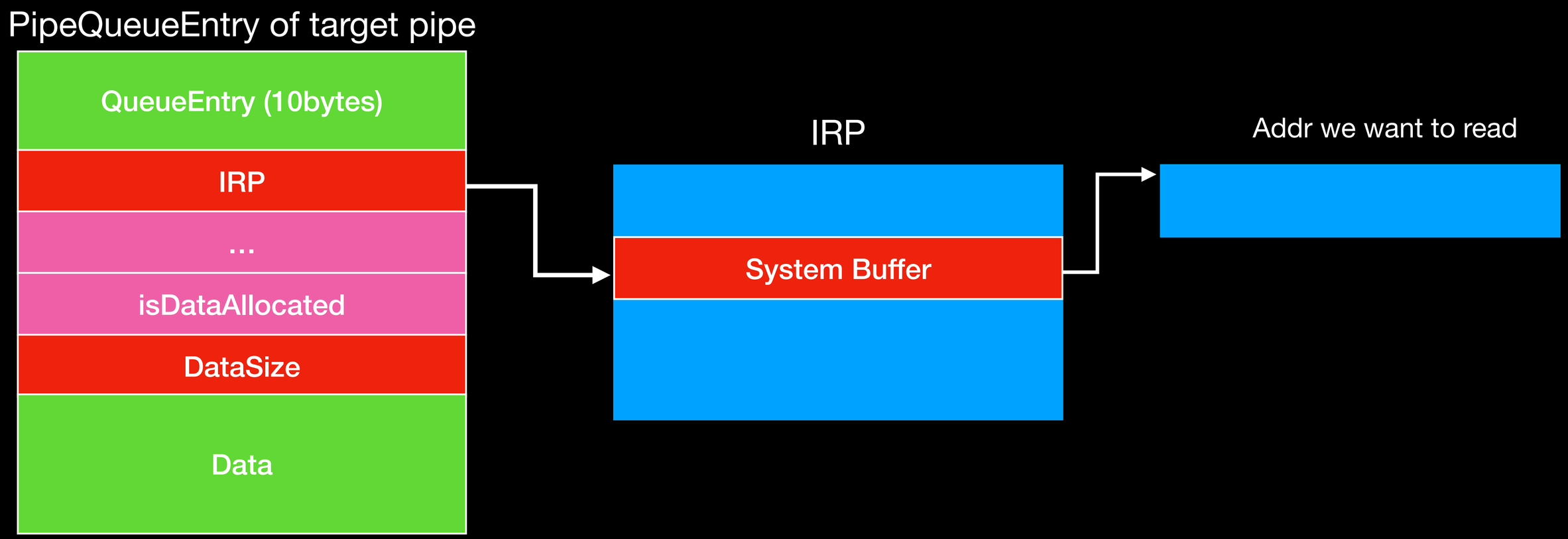
Exploitation
Arbitrary memory reading

- Let’s back to PipeQueueEntry.
  - Because we can overwrite the target pipe
    we also can over write PipeQueueEntry of target pipe

```
struct  PipeQueueEntry
{
  LIST_ENTRY list;
  IRP *linkedIRP;
  __int64 SecurityClientContext;
  int isDataAllocated;
  int readbytes;
  int DataSize;
  int field_2C;
  char data[1];
};
```

Exploitation
Arbitrary memory reading

- We can forge IRP in userspace, and modify PipeQueueEntry of target pipe.



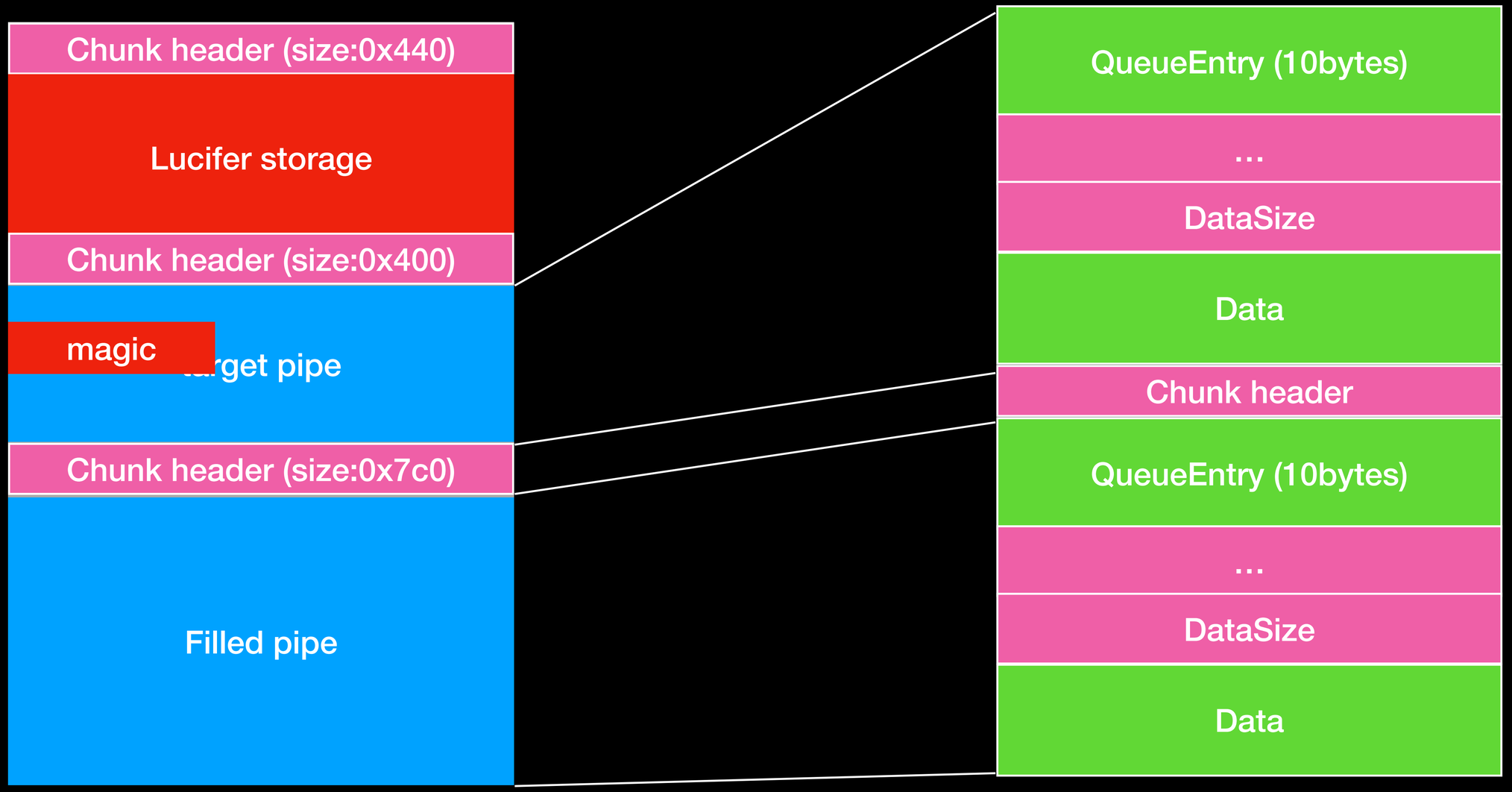
Exploitation

Arbitrary memory reading

- We have arbitrary memory reading !
  - But there are some problem
    - In normal case, we can use NtQuerySystemInformation to get address of nt. But we run exe under Low integrity. We can not do it.
    - We don’t have any kernel address to read.

Exploitation

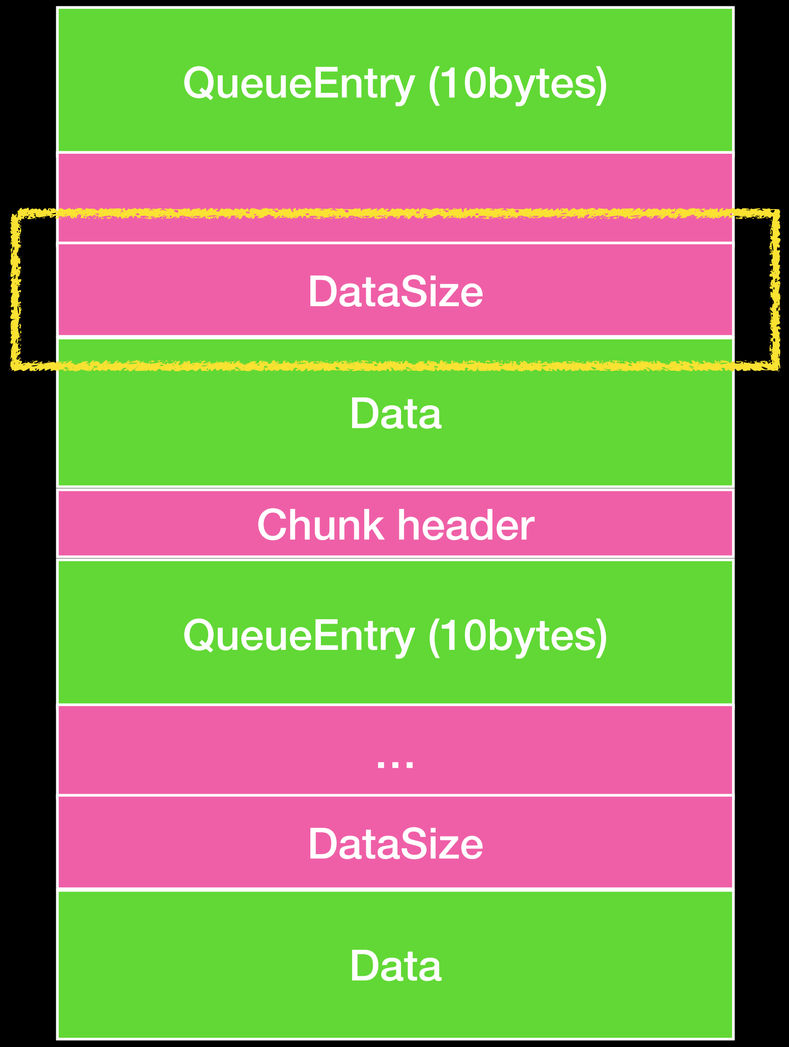
Get address of PipeQueueEntry of target pipe



Exploitation

Get address of PipeQueueEntry of target pipe

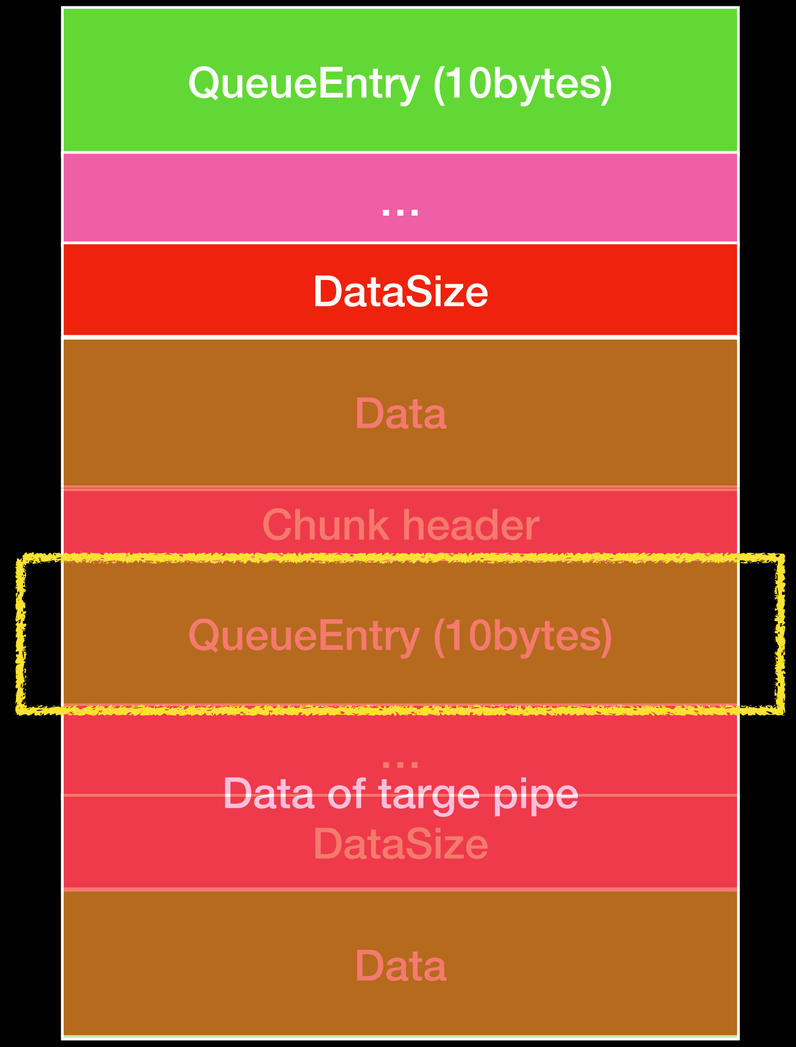
- We can use the vulnerability to overwrite the size of target pipe.



Exploitation

Get address of PipeQueueEntry of target pipe

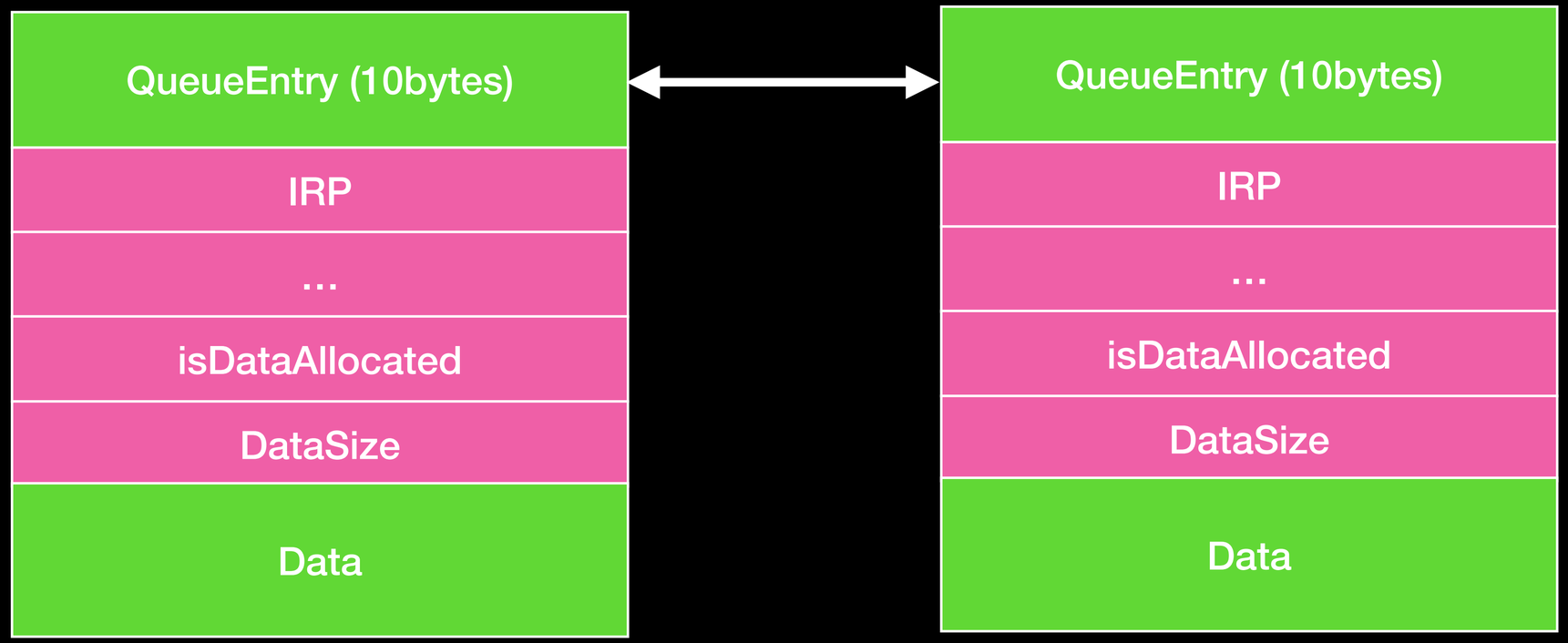
- We can use the vulnerability to overwrite the size of target pipe.
- We can read more data from target pipe
  - That is, we can leak the metadata of filled pipe
    - We can leak QueneEntry of filled pipe



Exploitation

Get address of PipeQueueEntry of target pipe

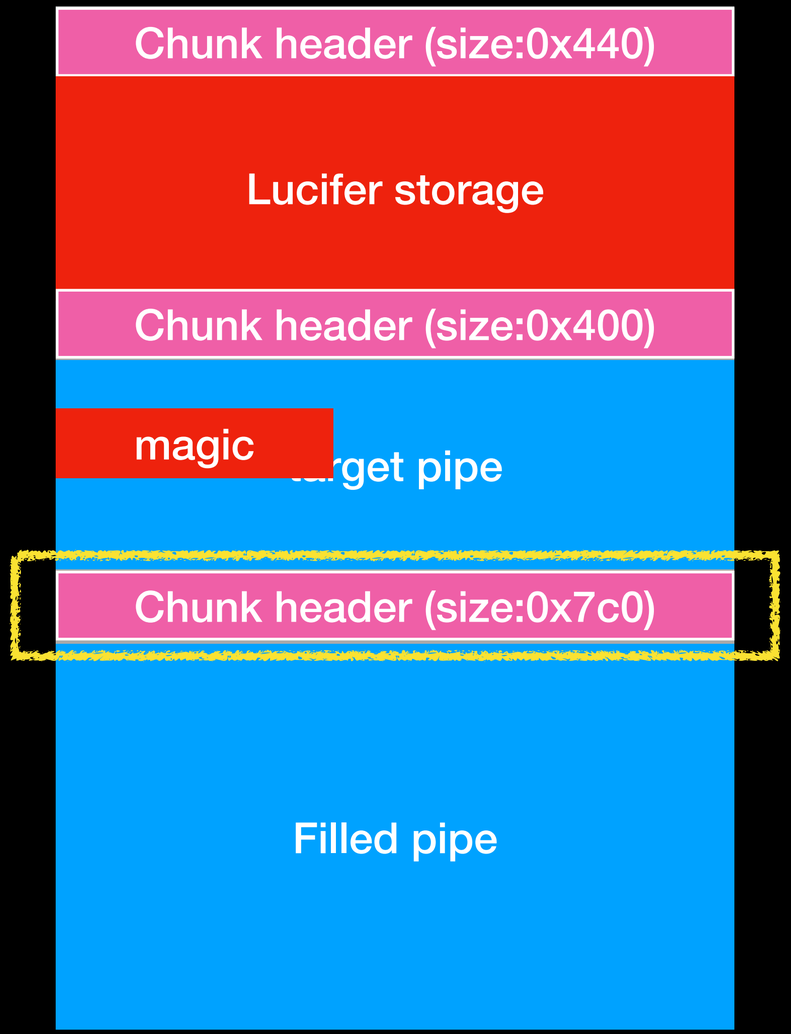
- Because PipeQueueEntry is a double linked list, we can use arbitrary memory reading to read address of filled pipe queue and get address of target pipe.
- Now we can read every thing around PipeQueueEntry of target pipe



Exploitation

Leak address of nt

- We can leak the chunk header of filled pipe.
  - Because we know the metadata and address of chunk, we can get RtlpHpHeapGlobals.HeapKey
  - RtlpHpHeapGlobals.HeapKey = metadata^chunk address ^ encode header



Exploitation

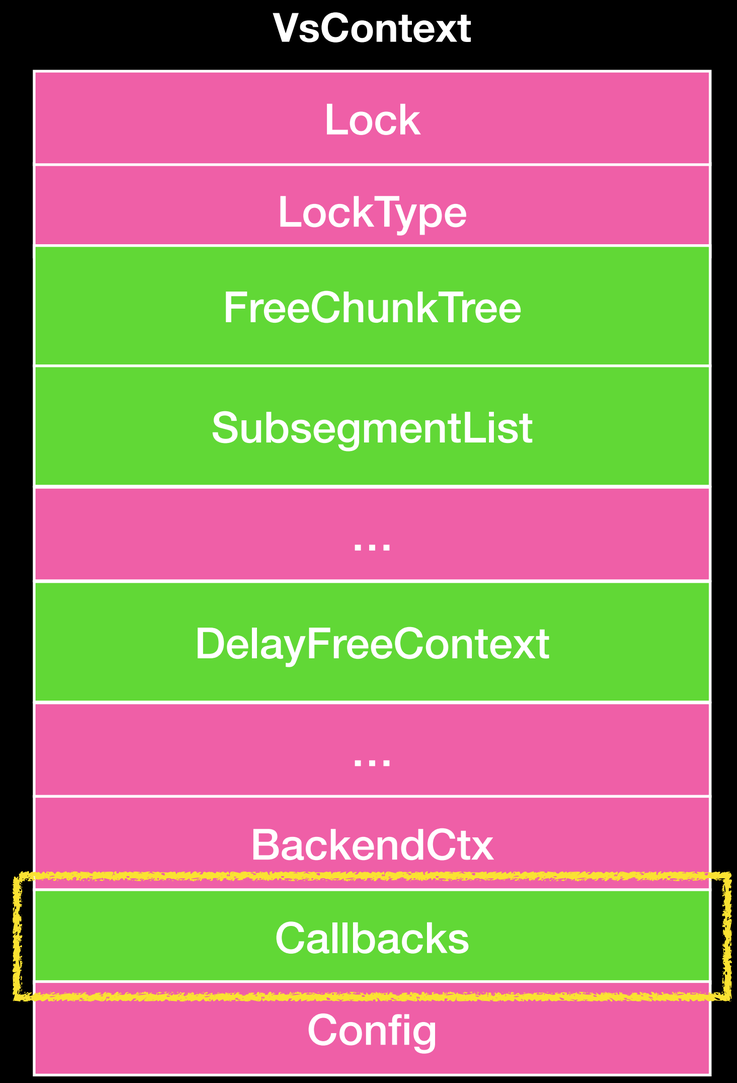
Leak address of nt

- We can calculate many address from address of PipeQueueEntry
  - Page segment = PipeQueueEntry & 0xfffffffffff00000 (Segment mask)
  - Signature of Page segment = readmem(Page segment + 0x10)
  - Segcontext = (Page segment)^ (Signature of Page segment) ^ RtlpHpHeapGlobals.HeapKey ^ 0xA2E64EADA2E64EAD
  - Segment heap = Segcontext - 0x100

Exploitation

Leak address of nt

- After we get the address of segment heap
  - We can leak callbacks function from
    - Segment heap->VsContext
  - But it’s be encoded, we need to recover it
    - Call back function = RtlpHpHeapGlobals.HeapKey^ encode data ^ VsContext address
  - We can use callback function to get address of nt



Exploitation

Arbitrary memory writing

- Because we have out of bound write and know the address of buffer
  - We can do arbitrary memory writing !

Exploitation
Get system token

- Because we have address of nt
  - We can get address of _EPROCESS of system process (pid=4)
    - Use arbitrary memory read to read nt!PsInitialSystemProcess
    - And read _EPROCESS->token

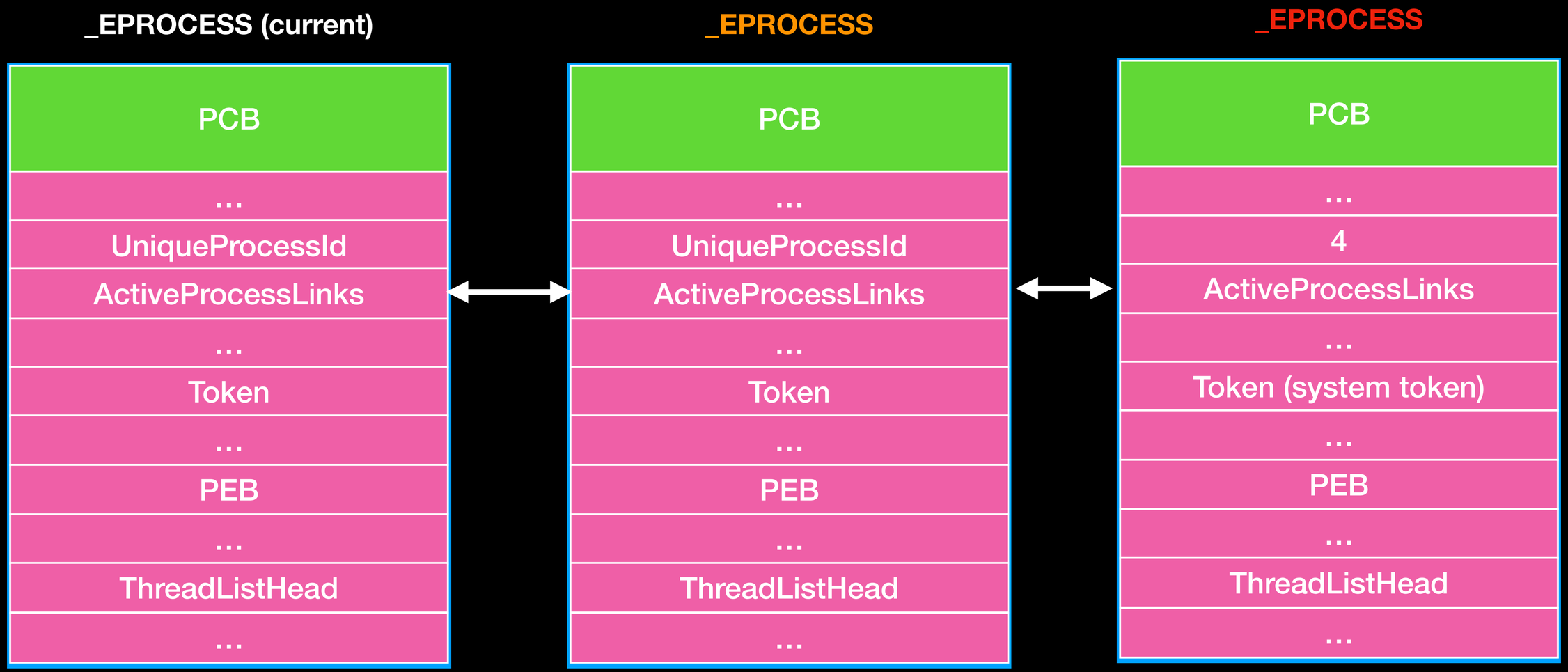
Exploitation

Find _EPROCESS of current process

- Because every _EPROCESS will be stored in a double linked list
  - We can traverse the linked list to find _EPROCESS of current process.
  - After find _EPROCESS of current of process, we can replace the token with token of system process
- Note :
  - _EPROCESS structure has been greatly changed at 20H1, you should make sure the offset is correct

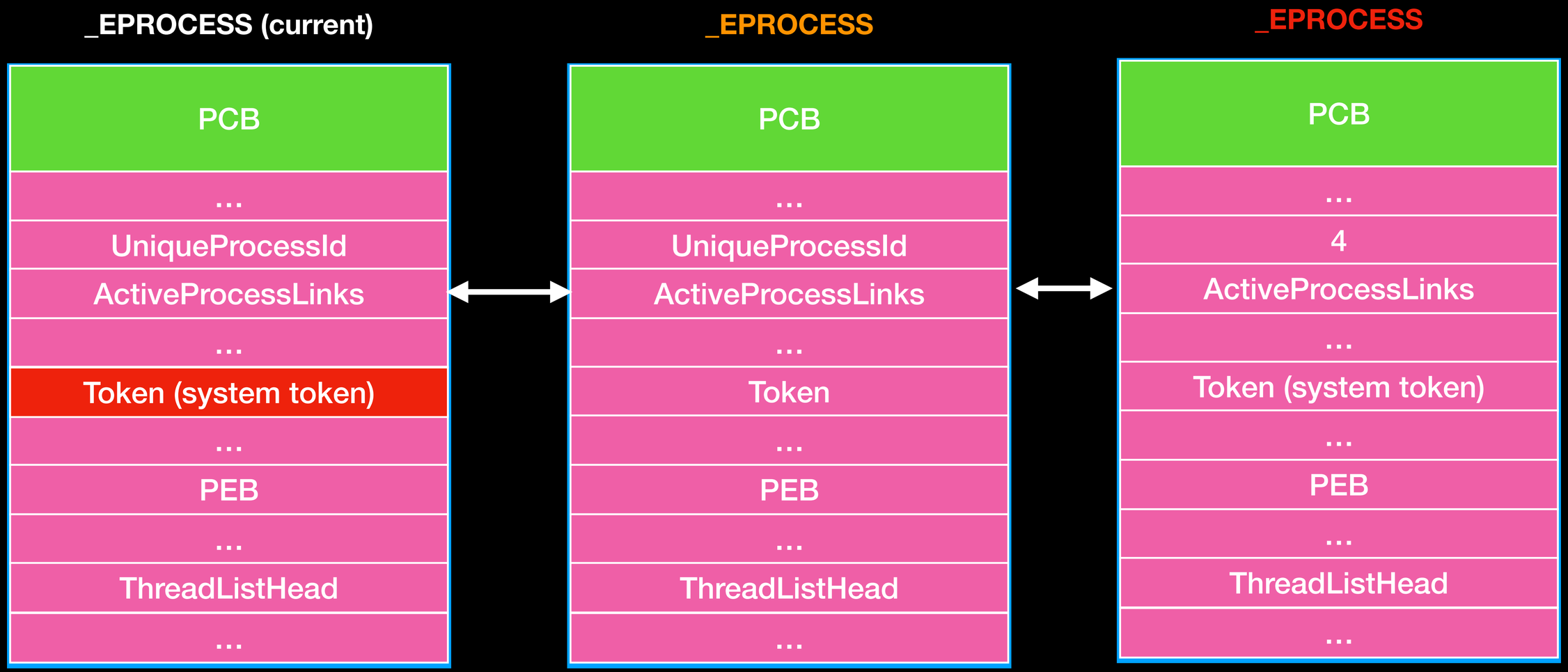
Exploitation

Find _EPROCESS of current process



Exploitation

Find _EPROCESS of current process



Exploitation

Get system shell

- After we replace the token
  - We can invoke system to get system shell !

Reference

- https://www.blackhat.com/docs/us-16/materials/us-16-Yason-Windows-10-Segment-Heap-Internals.pdf

- https://www.sstic.org/media/SSTIC2020/SSTIC-actes/pool_overflow_exploitation_since_windows_10_19h1/SSTIC2020-Article-pool_overflow_exploitation_since_windows_10_19h1-bayet_fariello.pdf