DeathNote of Microsoft Windows Kernel

windows kernel internals

$whoami

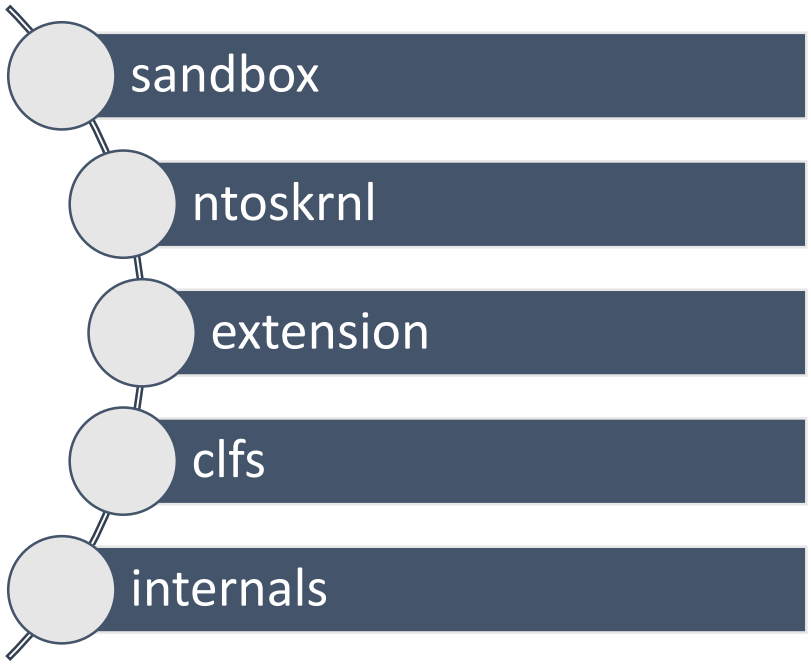
Peter

- @zer0mem
- Windows kernel research at KeenLab, Tencent
- pwn2own winner (2015 / 2016), pwnie nominee (2015)
- fuzzing focus : state
- wushu player

Daniel

- @long123king
- Windows kernel research at KeenLab, Tencent
- pwn2own winner (2016)
- fuzzing focus : data 'format'
- windbg guy

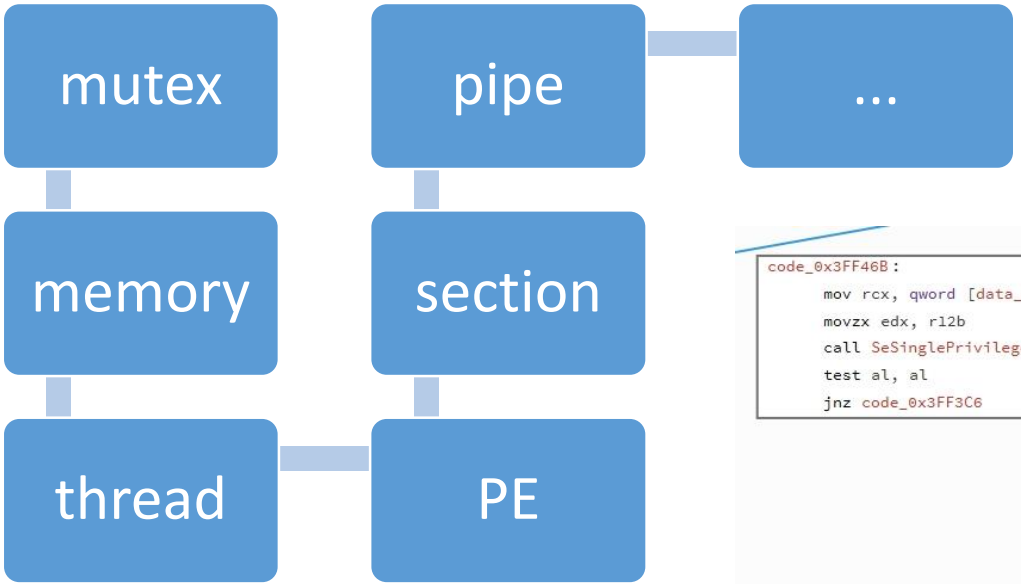
agenda



Sandbox

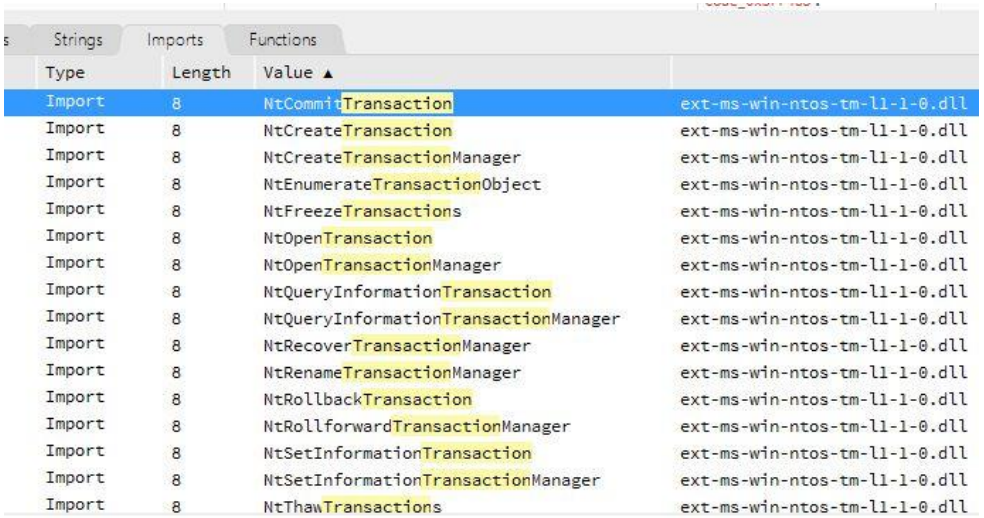
- limiting attack surface
  - potantional landscape for bugs
  - available methods for abusing it
- ACL vs access to various kernel objects
  - non ntos, non w32k drivers
  - various ntos objects
- w32k filtering
  - depends on sandboxed app needs
- w32k lockdown

Sandbox targets



… plus ?

- Nt*Transaction*
- Nt*Enlistment*
- Nt*Manager*



what ?

- Kernel Transaction Manager
  - 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.

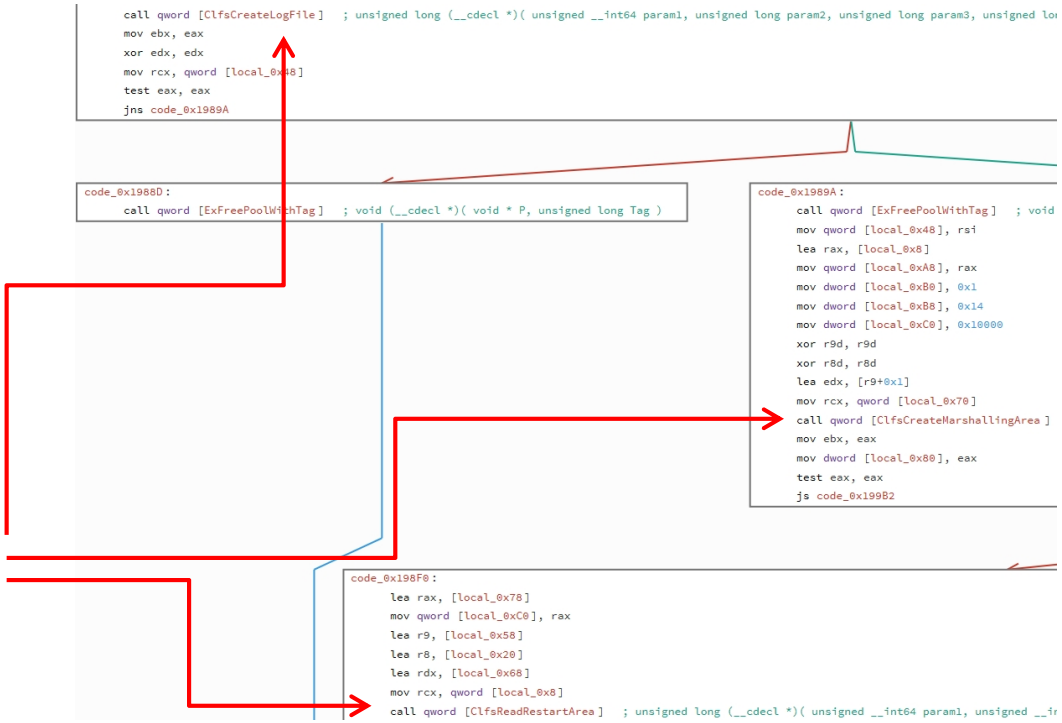
tm.sys

- simple object state
- few syscalls available
- not much code involved
- however interestingly interconnected

- Results :
  - 1 nullptr deref
  - 1 exploitable vulnerability

tm indirection

- tm.sys simple
  purpose driver
- but interesting
  module involved
  at backend
- CLFS.sys



CLFS.sys

- Purpose
  - The Common Log File System (CLFS) API provides a high-performance, general-purpose log file subsystem that dedicated client applications can use and multiple clients can share to optimize log access.
  - Any user-mode application that needs logging or recovery support can use CLFS.
- Where applicable
  - You can use CLFS for data and event management and to develop server and enterprise applications.
  - For data management, you can use CLFS with the following:
    - Database systems
    - Messaging, such as store-and-forward systems
    - Online transactional processing (OLTP) systems
    - Other kinds of transactional systems

CLFS.sys

- well integrated to transactions and more!
- c++ code base
- serve fair attack surface

- ... but not at appcontainer or untrusted level ...

- or does it ?

NtCreateTransactionManager

- depends on CLFS
- use CLFS for its checkpoints

- therefore implies :
  - Opens CLFS
  - *PARSE* CLFS
  - interact with CLFS

- lets try it out!

CLFS - data fuzzing I.

- i am not fan of data fuzzing in kernel
  - as i am strongly against data parsing at kernel at all :)
- lets do quick probe, that i am ok with :
  - mutate randomly file
  - results = 0
  - cool for me, i am not much interested anyway
  - get back to original idea!

CLFS - state fuzzing

- approach 1.
  - RE clfs.sys
  - go to ioctl
  - .. ah too lazy to do that from scratch ...
- approach 2.
  - go trough msdn docs
  - understand how those api works
    - callstack necessary to suceed to call one or another api
  - implement that logic to Qilin (our internal fuzzer)
  - mess with logic in Qilin little bit

bugz++

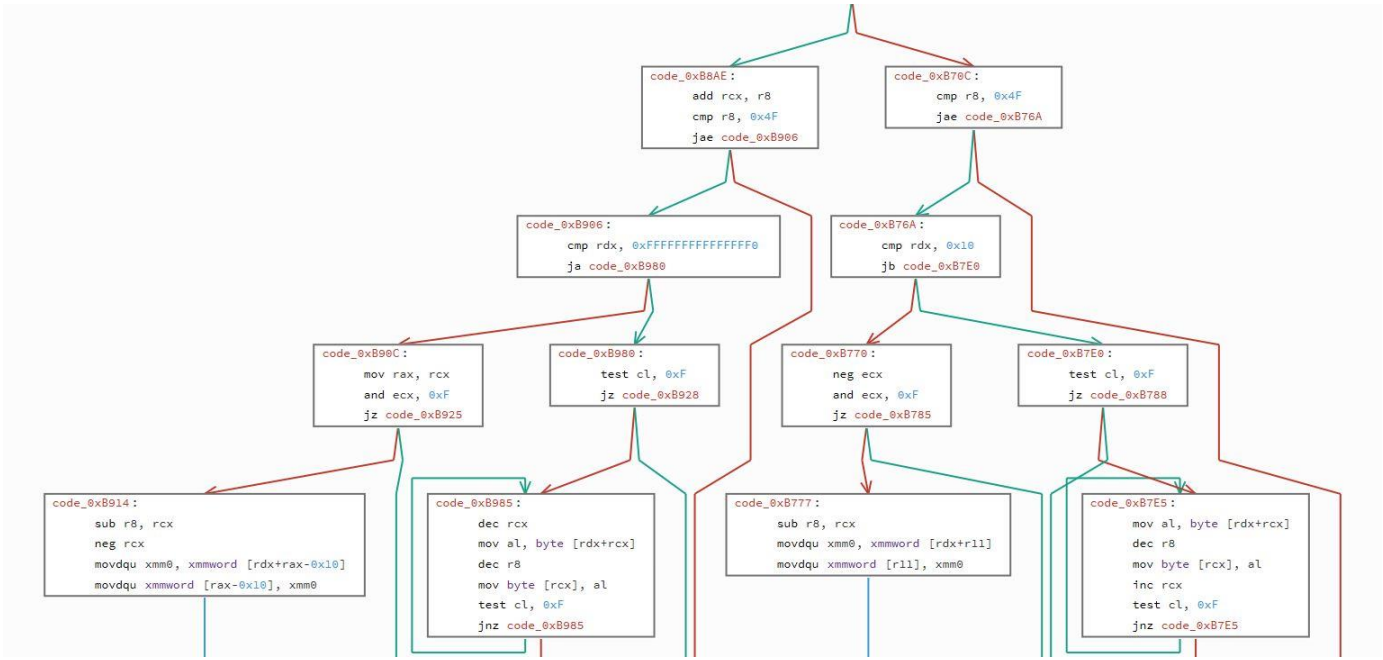
- after first dry run of fuzzer in 15 min first crashes
  - ... wtf
- but ddos only
- eliminate that
- another bugz apear
- now time to rethink .. data fuzzing really so bad idea afterall ?

CLFS - data fuzzing II.

- RE where & how are data parsed
- EntryPoint : ClfsCreateLogFile
- ouch ... magic .. dummy fuzz proof
  - I. crc
  - II. rellocation required

CLFS - lets fuzz more seriously

- too lazy to re-implement existing code, but is it even necesary ?



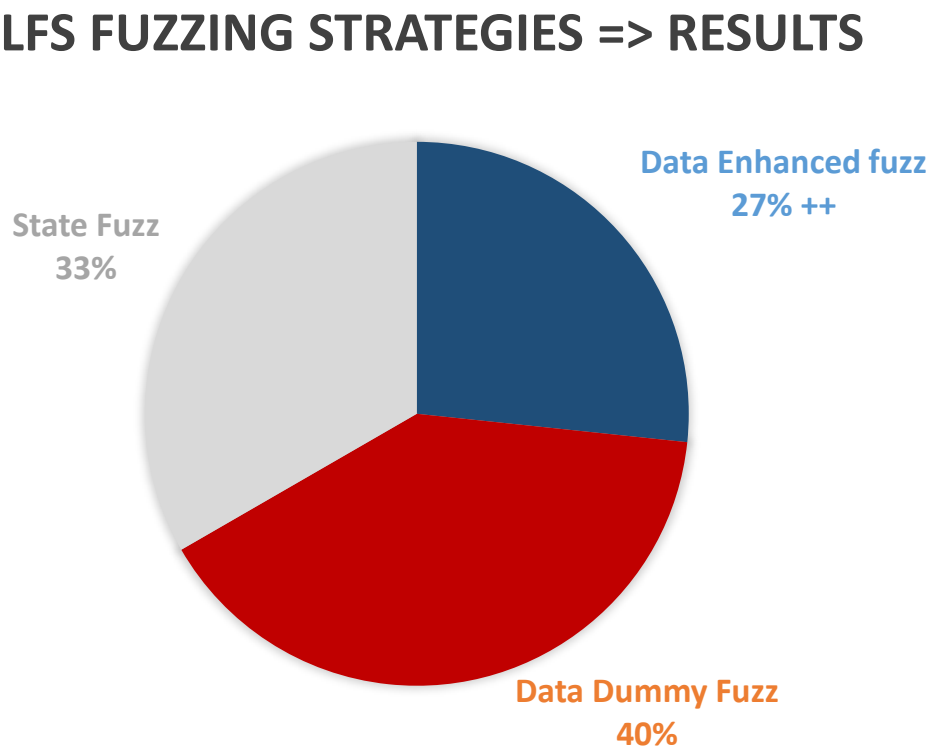
CLFS - lets fuzz more seriously

- too lazy to implement crc & rellocations

```
__checkReturn
NTSTATUS
ClfsEncodeBlock_fn(
    void* block,
    size_t size,
    size_t type,
    size_t gran = 0x10,
    bool crc32 = true
    );
decltype(&ClfsEncodeBlock_fn) ClfsEncodeBlock =

void
ClfsDecodeBlock_fn(
    void* block,
    size_t size,
    size_t type,
    size_t gran,
    size_t* res
    );
decltype(&ClfsDecodeBlock_fn) ClfsDecodeBlock =
```

```
void
DoPoC()
{
    //...

    CMemFromFileResource clfs("c:/windows/system32/drivers/clfs.sys");
    CMemFromFileResource crc("m_rgCrcTable");

    printf("\n table : %i", crc->size());

    memcpy(clfs->get<char>() + 0x1D480, crc->get<char>(), crc->size());

    ClfsEncodeBlock = reinterpret_cast<decltype(&ClfsEncodeBlock_fn)>(clfs->get<char>() + 0x9d88);
    ClfsDecodeBlock = reinterpret_cast<decltype(&ClfsDecodeBlock_fn)>(clfs->get<char>() + 0x9d88 - 0x0

    DWORD protect;
    if (FALSE == VirtualProtect(clfs->get<char>(), clfs->size(), PAGE_EXECUTE_READWRITE, &protect))
        return;
```

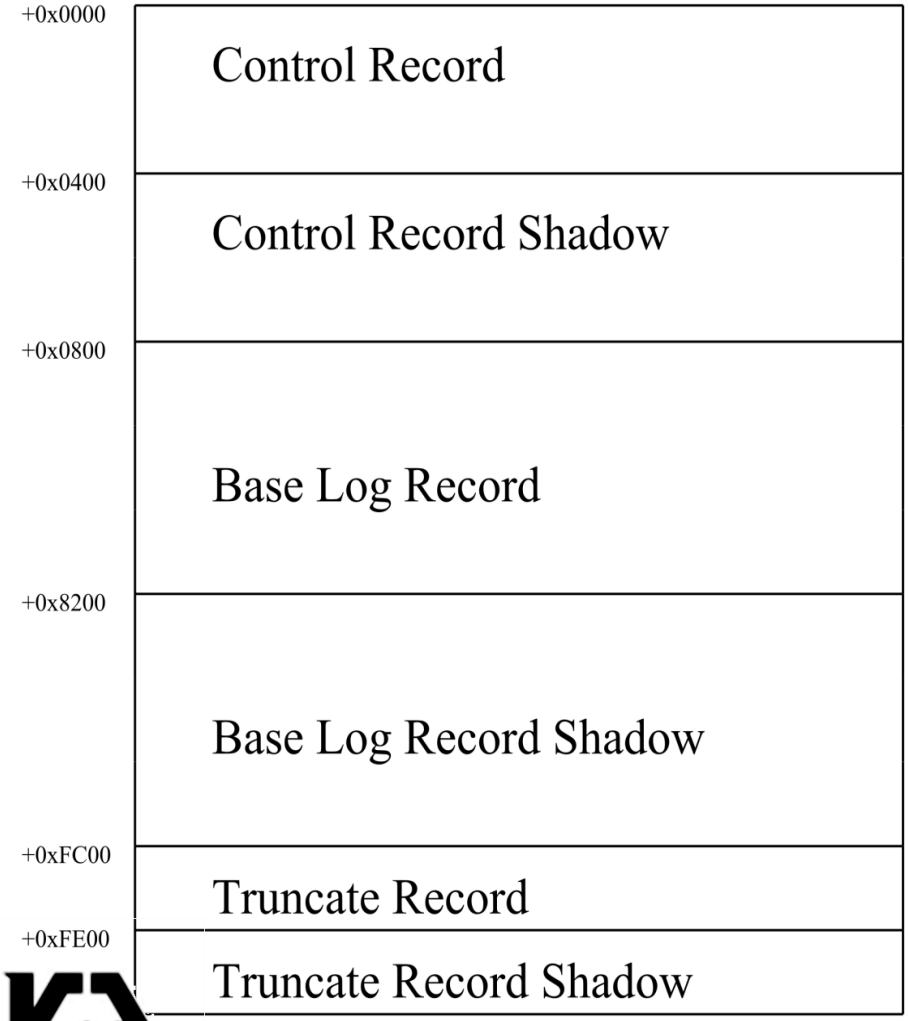
CLFS { state, dummy, enhanced }



CLFS internals

... under the hood ...

BLF (Base Log File) Format



[2 Sectors], contains layout, extend and truncate information.

[2 Sectors], shadow copy of control record

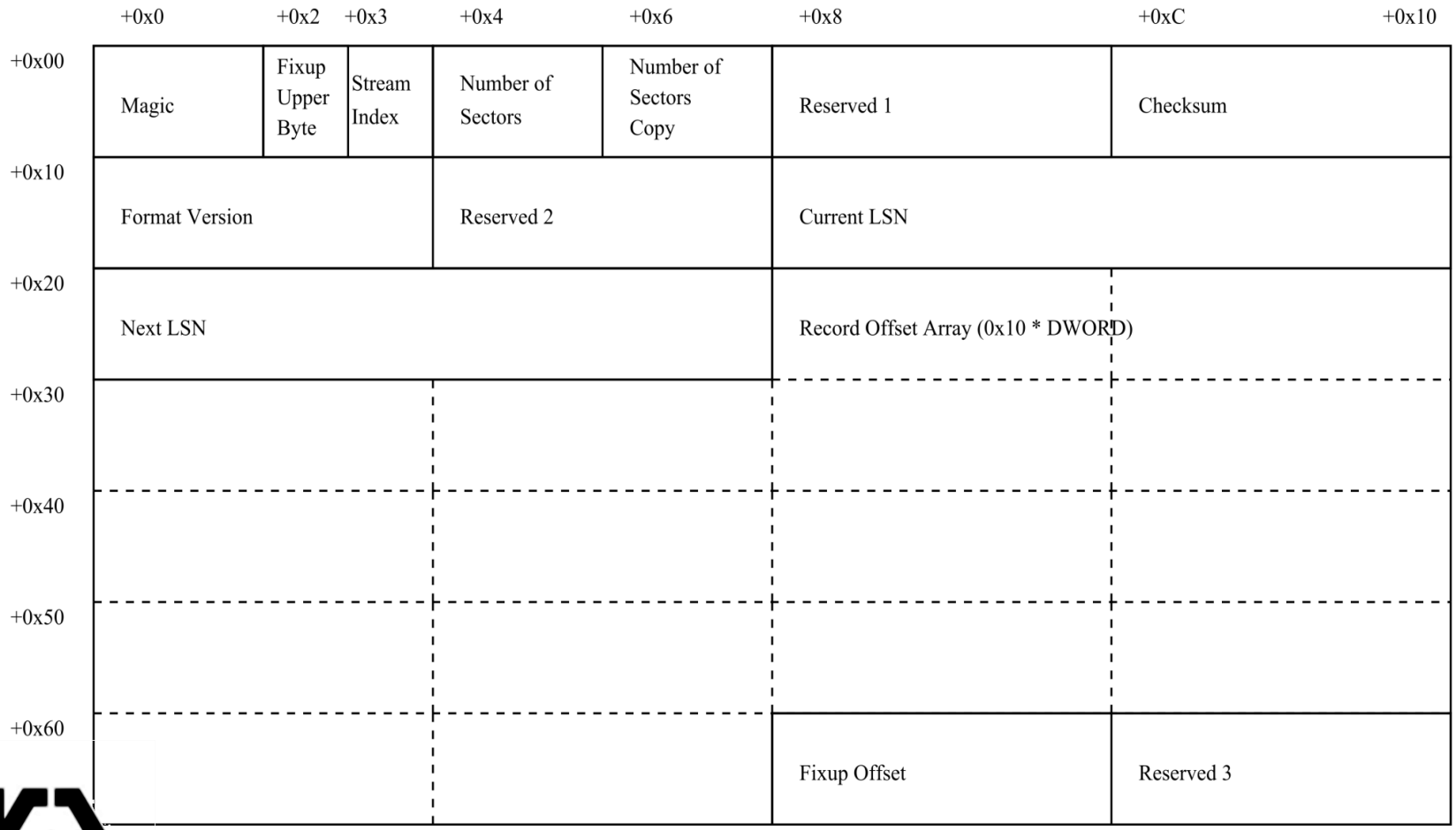
[0x3D Sectors], contains client and container information.

[0x3D Sectors], shadow copy of base log record.

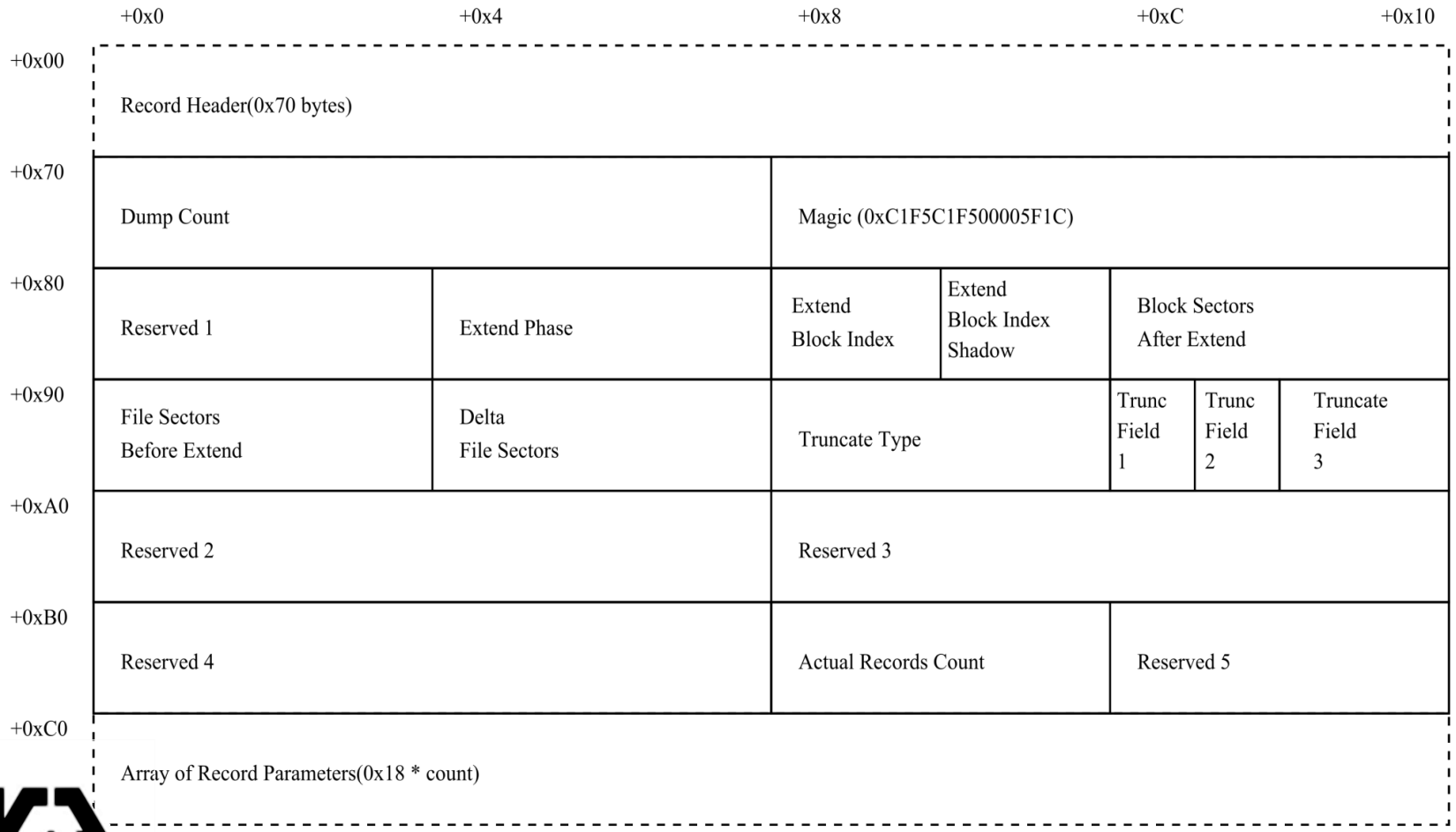
[1 Sector], contains truncate information.

[1 Sector], shadow copy of truncate record.

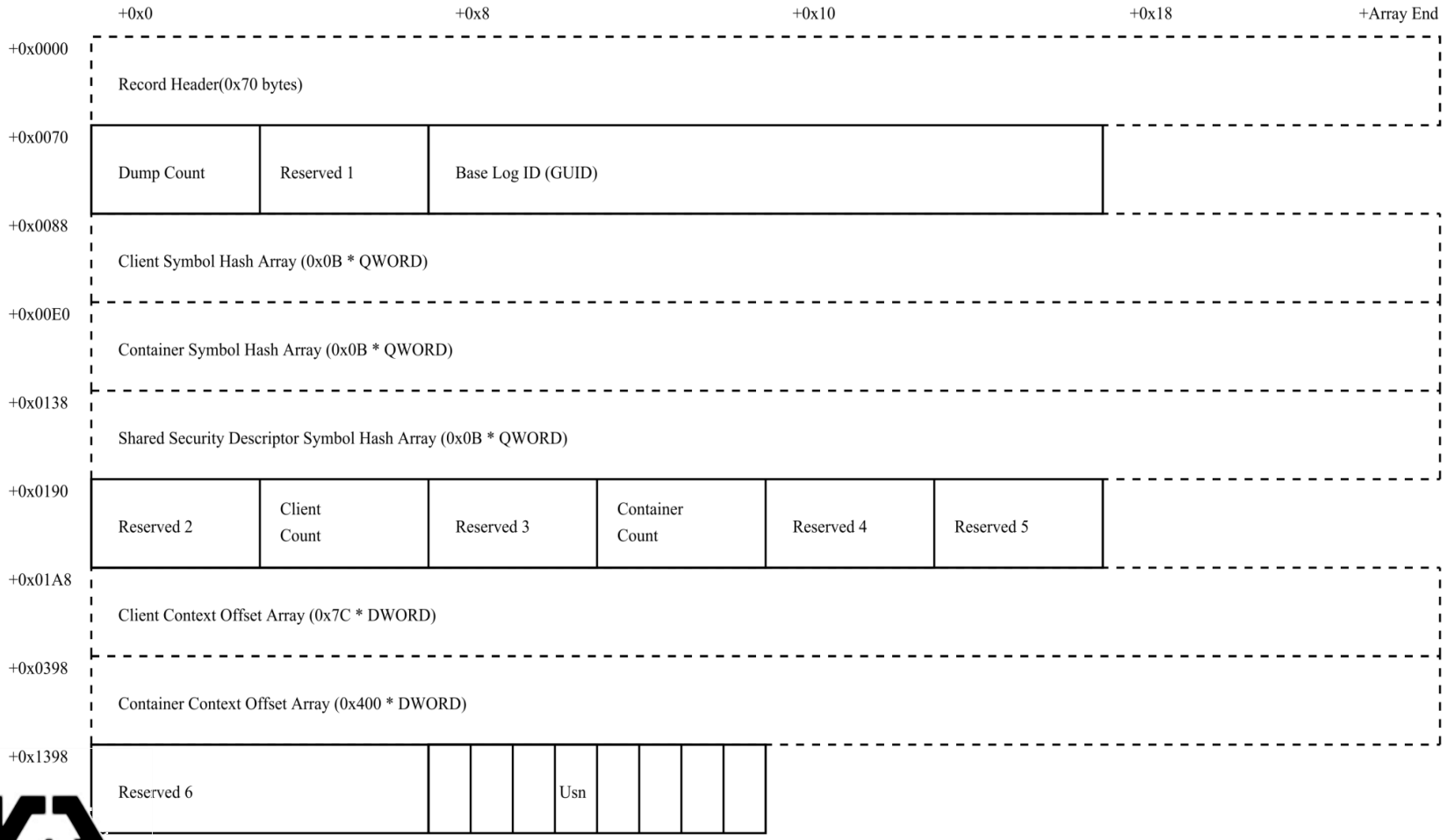
Record Header



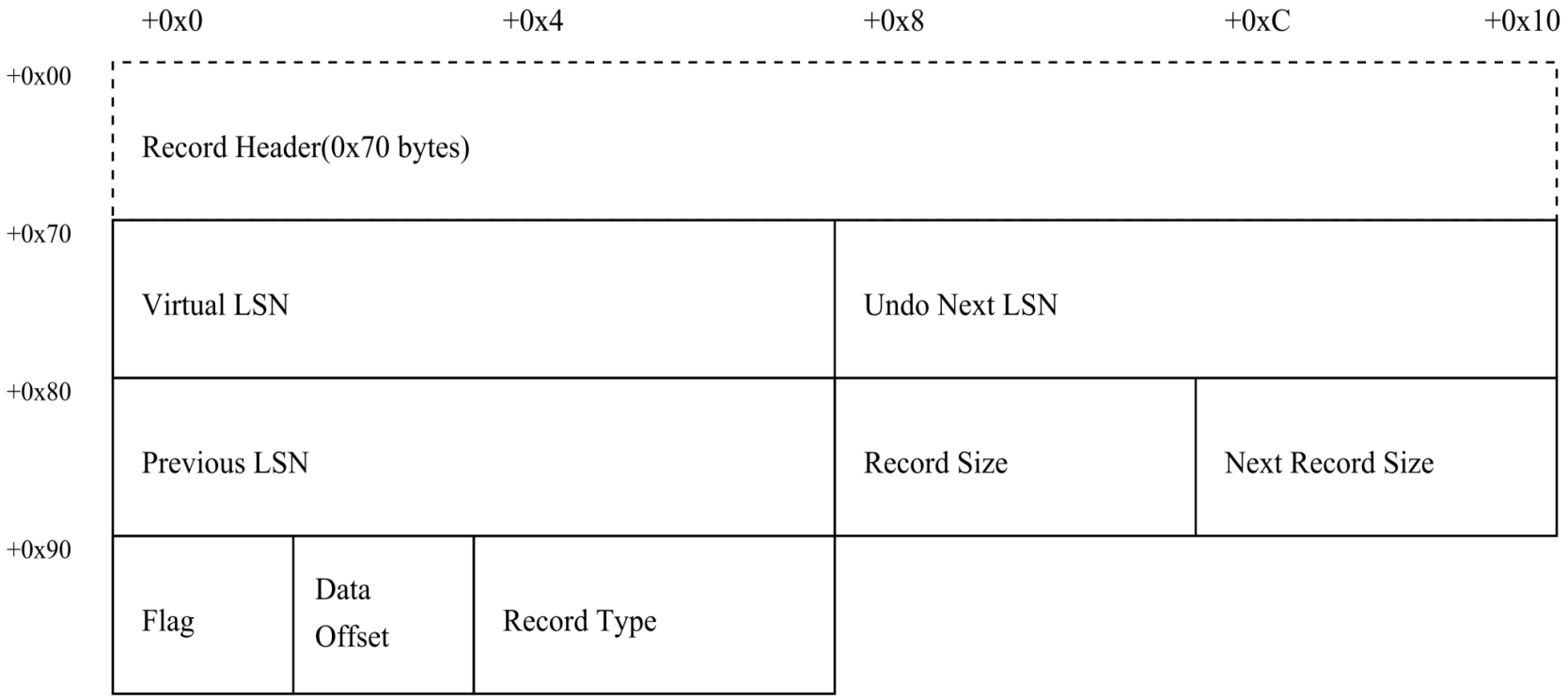
Control Record



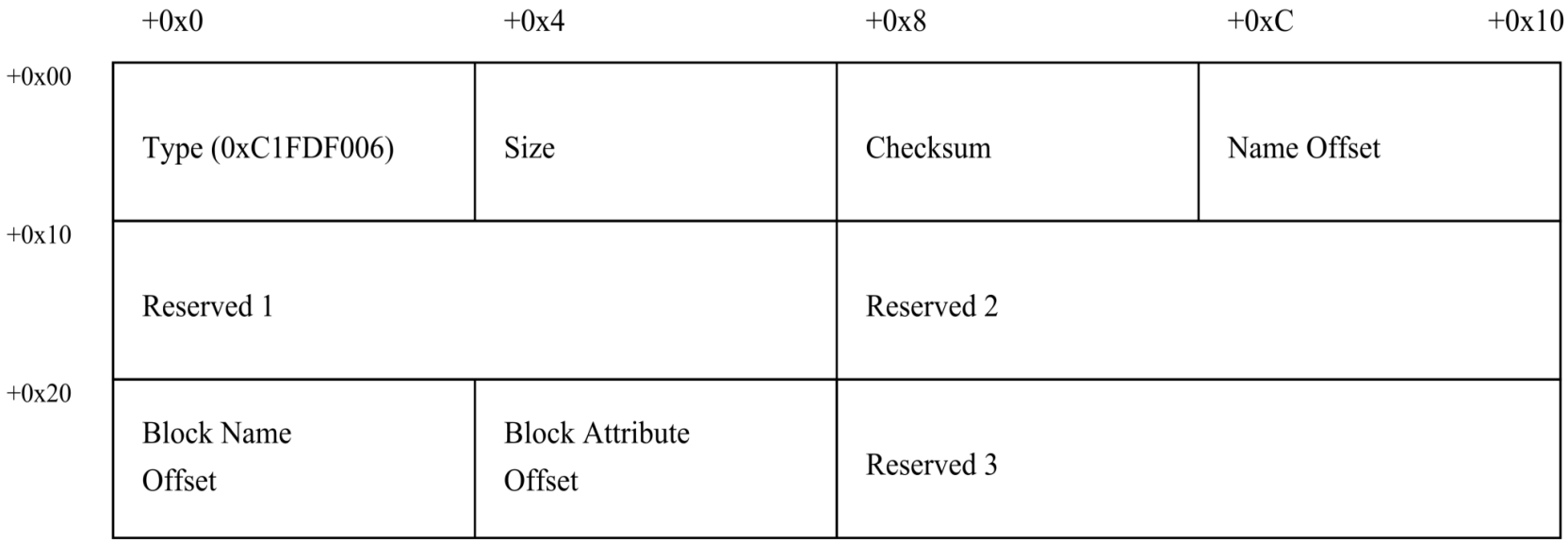
Base Log Record



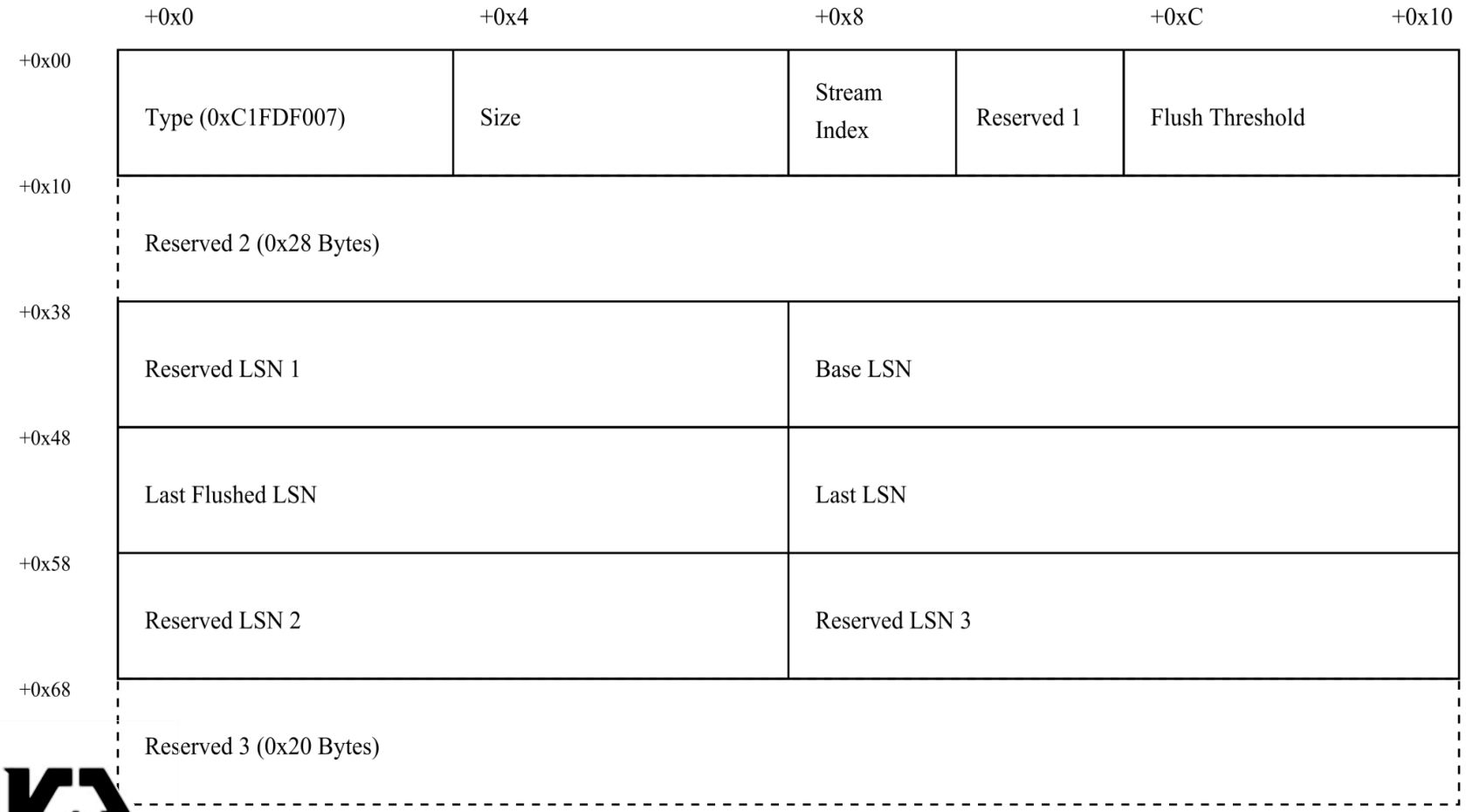
Container Record



Symbol Header



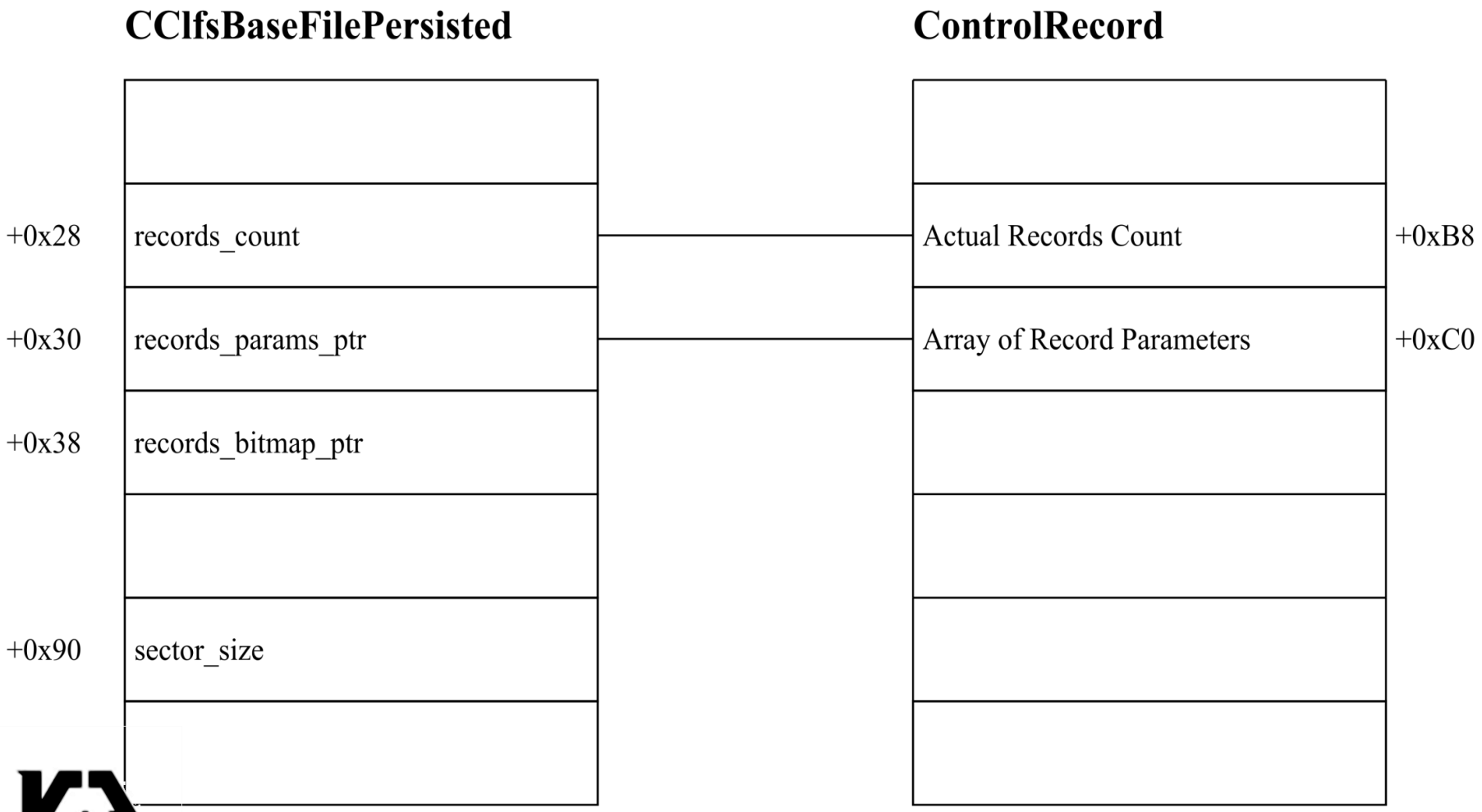
Client Context



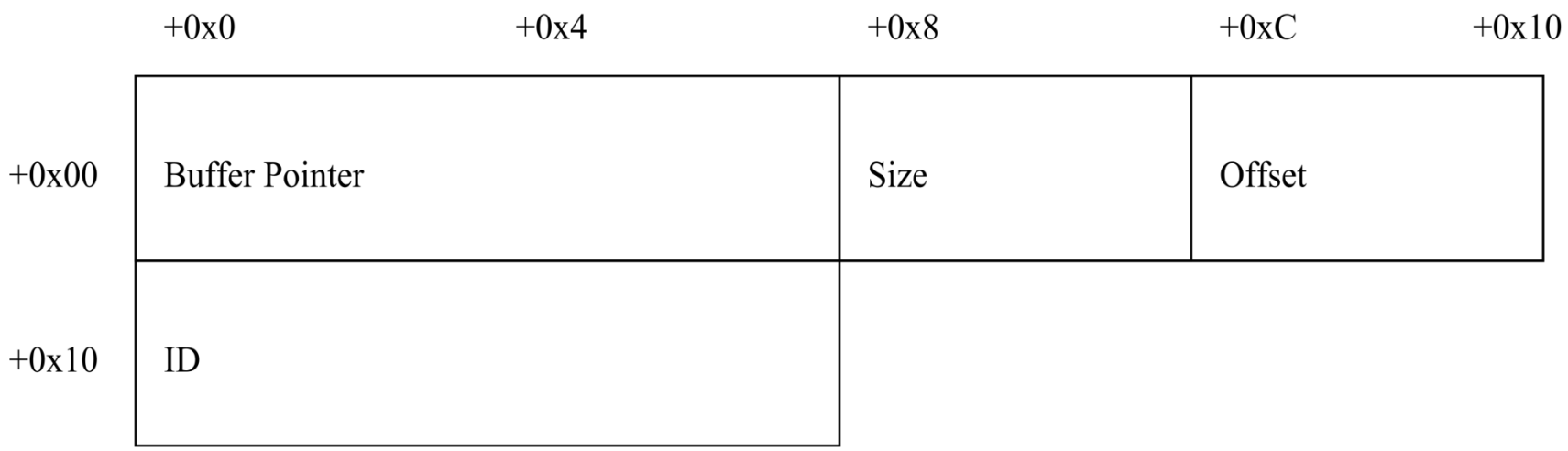
Container Context

	+0x0	+0x4	+0x8	+0xC	+0x10
+0x00	Type (0xC1FDF008)	Size	Container File Size	Reserved 1	
+0x10	Physical Container Index	Logical Container Index	Reserved LSN 1		
+0x20	Stream Count	Flag	Reserved 2		

CClfsBaseFilePersisted::ReadImage



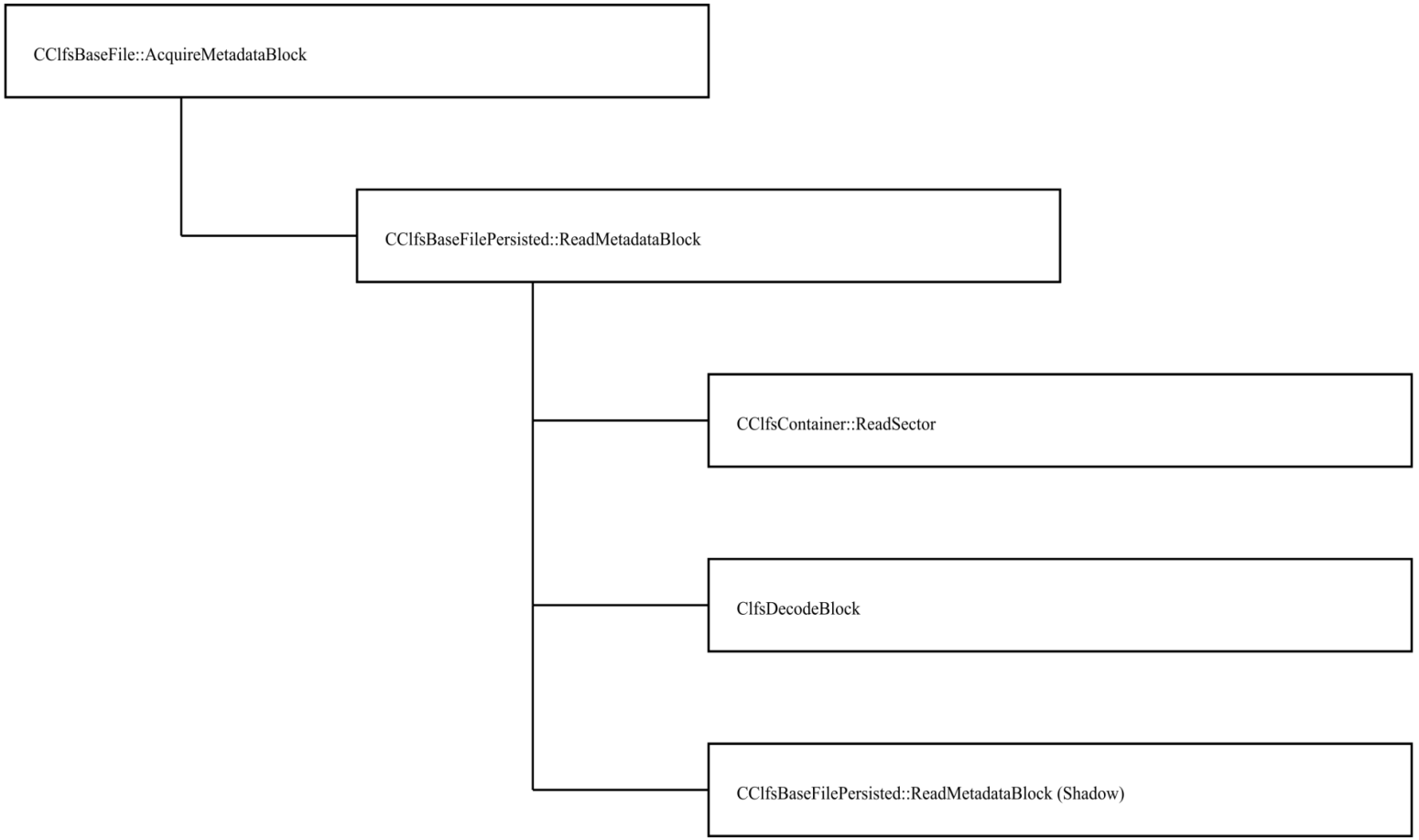
Record Parameter



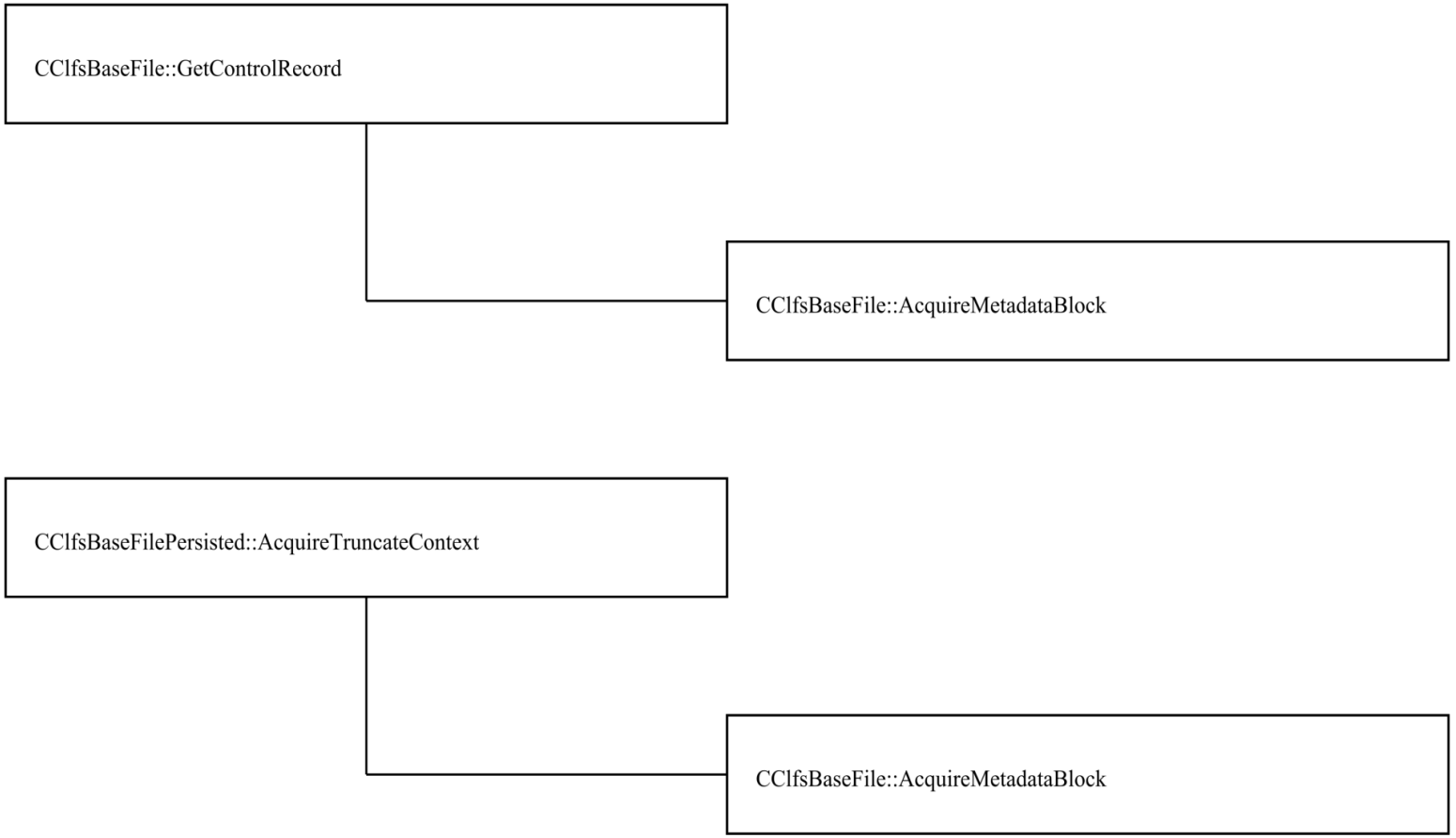
CClfsBaseFile::GetBaseLogRecord

```
    CClfsBaseFile::GetBaseLogRecord(CClfsBaseFile* this)
                    xor     eax, eax
                    cmp     ax, [rcx+28h]
                    jz      short locret_1C00335DB
                    mov     rcx, [rcx+30h]
                    mov     rcx, [rcx+30h]
                    test    rcx, rcx
                    jz      short locret_1C00335DB
                    mov     eax, [rcx+28h]
                    add     rax, rcx

    locret_1C00335DB:
                    retn
```

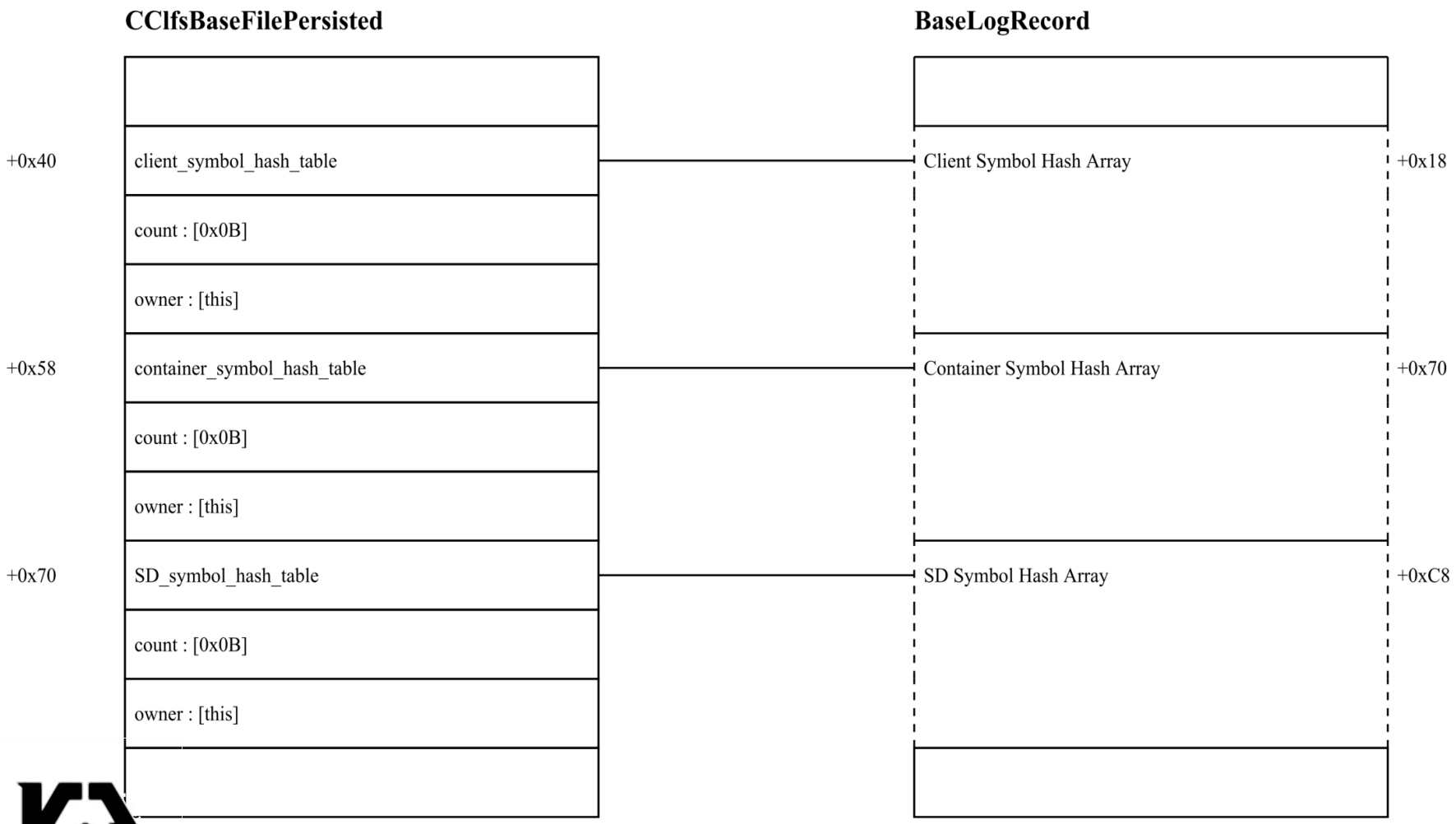
CClfsBaseFile::AcquireMetadataBlock



Use of AcquireMetadataBlock



CClfsBaseFilePersisted::OpenImage



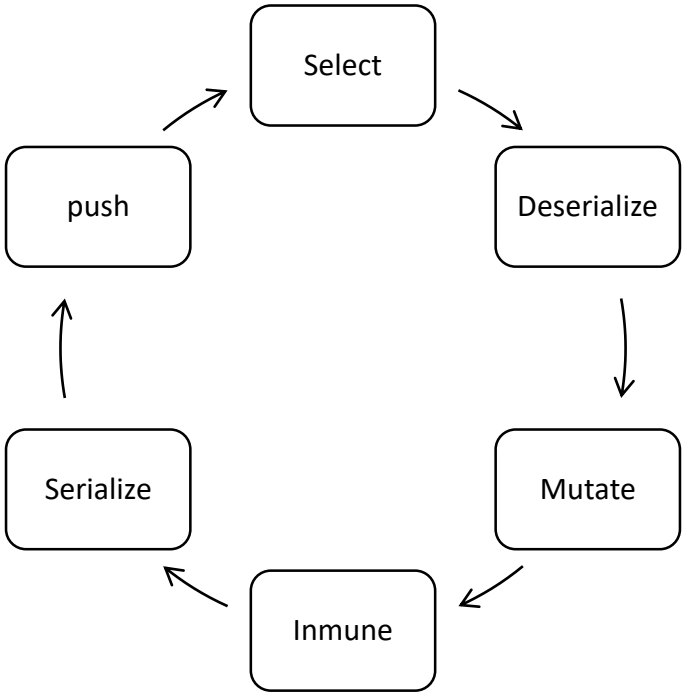
Symbol Hash Function

```
__int64 ClfsHashPJW(const struct _UNICODE_STRING *a1)
{
    unsigned int v1 = 0, v4 = 0, v6;
    PWSTR wchar_buffer = a1->Buffer;
    const struct _UNICODE_STRING *v3 = a1;
    if ( a1->Length & 0xFFFE ){
        do{
            int v5 = 0x10 * v1 + RtlUpcaseUnicodeChar(*wchar_buffer);
            v6 = v5 & 0xF0000000;
            if ( v5 & 0xF0000000 )
                v5 ^= v6 >> 0x18;
            v1 = v6 ^ v5;
            ++wchar_buffer;
            ++v4;
        }
        while ( v4 < (unsigned int)v3->Length >> 1 );
    }
    return v1;
}
```

Enhanced CLFS format fuzzing

- If you know the target well enough, you can fuzz it well.
- Since now, we know:
  - BLF file format
    - Control Record
    - Base Log Record
      - Symbol Header
      - Client Context
      - Container Context
  - Container Record
- Clfs.sys has its own logic to parse these formats, is it robust enough?

Enhanced CLFS format fuzzing



Enhanced CLFS format fuzzing

```
class CControlRecord : public CFormatBase<CControlRecord>
{
    ……
    virtual bool serialize(ostream & out) const override;
    virtual bool deserialize(istream & in) override;
    virtual bool mutate() override;
    ……
};

class CBaseLogRecord : public CFormatBase<CBaseLogRecord>
{
    ……
    virtual bool serialize(ostream & out) const override;
    virtual bool deserialize(istream & in) override;
    virtual bool mutate() override;
    ……
};
……
```

Enhanced CLFS format fuzzing

```
bool CCLFSFormat::deserialize(istream & in)
{
    ……
    m_controlRecord.deserialize(in);
    m_controlRecordShadow.deserialize(in);
    m_baseLogRecord.deserialize(in);
    m_baseLogRecordShadow.deserialize(in);
    m_truncateRecord.deserialize(in);
    m_truncateRecordShadow.deserialize(in);
    ……
}

bool CCLFSFormat::mutate(istream & in)
{    ……   }

bool CCLFSFormat::serialize(istream & in)
{    ……   }
```

Enhanced CLFS format fuzzing

```
CCLFSDocument::CCLFSDocument(const string filename)
    :m_template_filename(filename)
    ,m_template_stream(filename, ios::in | ios::binary)
{
    /* number: 0 */m_engine.registerFilter(make_unique<CCommonErrorBypass>());
    /* number: 1 */m_engine.registerFilter(make_unique<CPOC_XXX_1>());
    /* number: 2 */m_engine.registerFilter(make_unique<CPOC_XXX_2>());
    /* number: 3 */m_engine.registerFilter(make_unique<CPOC_XXX_3>());
    /* number: 4 */m_engine.registerFilter(make_unique<CPOC_XXX_4>());
    /* number: 5 */m_engine.registerFilter(make_unique<CPOC_XXX_5>());
    ……
}

void CCLFSDocument::mutate()
{
    m_clfs_format.mutate();
    m_engine.triggerFilter(3, m_orginal_clfs_format, m_clfs_format);
}
```

Enhanced CLFS format fuzzing

```
bool CPOCFilterEngine::triggerFilter(size_t filterIndex,
CCLFSFormat& originalFormat, CCLFSFormat& format)
{
    bool b_triggered = false;
    for (size_t i = 0; i < m_filters.size(); i++)
    {
        if (i == filterIndex)
        {
            m_filters[i]->infect(originalFormat, format);
            b_triggered = true;
        }
        else
            m_filters[i]->immune(originalFormat, format);
    }

    return b_triggered;
}
```

Q & A

Thank you!

科恩实验室
KEEN
security lab