Advances in Win32 ASLR Evasion

Justin Ferguson, May 2011
Distinguished IOActive Associate

Summary of Points 1/4

- Large emphasis has been placed on randomizing pointers directly
  - Lesser emphasis placed to ensure pointers that point to them are non-deterministic
  - This is dangerous in some conditions
    - 32-bit Windows system calls accessed through static pointers
    - More generically take advantage of instances where * is deterministic and ** is special
  - *(special) allows meta-programming sorta

Summary of Points 2/4

- Process ASLR efficiency is **not constant**
  - Some actions have deterministic effects
  - Threads create a new stack, mapped files occupy space
    - Directly or indirectly controllable
    - Leads to potential "side-channel-esque" attacks
  - Programmers inherently inject order to memory layout
    - When did you last select random indexing to operate on an object array?
    - When did you last over-allocate and start from a randomized base?

Summary of Points 3/4

- Secure systems must take an increasing number of steps to prevent this order from occurring
  - When you plot the pointers returned from the malloc (without free) in OpenBSD, you find...
    - A pattern of 3-4 base addresses intermixed between allocations
    - No accurate relationship between knowledge of order and how many subsequent allocations can occur
  - Windows returns deterministic incrementing pointers

IOActive™
COMPREHENSIVE COMPUTER SECURITY SERVICES

Summary of Points 4/4

- The resultant environment's complexity favors the attacker
  - Taming the chaos becomes attackers goal
  - 32-bit Windows 7 and WoW64 albeit differently
  - Current protections include
    - NX, ASLR, DEP, compilers that bitch at you for everything, SafeSEH, SEHOP, the SDL and Mike Howard comedy hour with cameos from David the Grouch, banned APIs, mandatory static analysis at check-in, variable reordering... THE SDL TRAINING CARD GAME!
  - Current protections not offered include

Lineage

- Personally, 2007-2008 timeframe
  - Murmurs with credible evidence that establish active usage pre-dating this time
- Updated because I didn't google
  - Chinese forums
  - HDM in Metasploit, albeit not for this reason
- XCon 2010
  - "Defeat Windows 7 browser memory protection" Chen Xiabo & Xie Jun
  - http://ivanlef0u.fr/repo/expl0it/XCon2010_win7.pdf

Meta

- Current efficiency (syscalls)
  - 32-bit Windows is ideal
  - WoW64 is not covered here
    - Bit of a moving target
    - Various non-system call function pointers to be repurposed (22-May-2011)
  - Win x64 is not covered here
    - Bit of a moving target
    - Fundamental aspect of technique no longer applies
    - General concepts apply, but broadly

Scene

- 32-bit Windows 7 patched as of 22-May-2011
  - DEP, /GS, ASLR, SafeSEH
  - Absence of non-randomized DLLs
- Specific focus on instances where stack pointer is controlled
  - Most obvious: stack overflows
  - Less obvious: contextual situations, mov esp, lol
- One threat per network connection
  - Not necessary, but not uncommon either
  - Represents one aspect of an ideal attack state—gives “safe from everyone else” memory
- Execution transfer requires **, can’t just use ret
  - Everything’s C++ anyway—mov ecx, [ptr]—call [ecx+x]

AKA

```
typedef struct _X_t { void (*ptr)(void); […] } X_t;
T func(...) {
    X_t     xInstance;
    char    buf[SIZE];
    [….]
    read_one_from_network_until_crlf(&buf);
    retval =  xInstance.ptr();
    if (0 > retval)
        errExitMsgToNetwork(“DANGER WILL ROBINSON: 0x%x\n”,
    GetLastError());
    […]
    }

int main(…)
{
       […]
       while (1) {
                     fd = accept(…);
                     CreateThread(…, &func, &fd, …);
```

Win32 Syscalls

- Call into kernel32, say VirtualProtect()
  - Kernel32!VirtualProtect redirects into layered dlls
    - API-MS-Win-Core-Memory-L1-1-0.dll
    - Gets fixed up at runtime
    - calls KERNELBASE!VirtualProtectEx
  - KERNELBASE!VirtualProtectEx calls ntdll!ZwProtectVirtualMemory

```
0:003> u ntdll!ZwProtectVirtualMemory
ntdll!ZwProtectVirtualMemory:
    77a15360 b8d7000000      mov     eax,0D7h
    77a15365 ba0003fe7f      mov     edx,offset SharedUserData!
    SystemCallStub (7ffe0300)
    77a1536a ff12            call    dword ptr [edx]
```

dt ntdll!_KUSER_SHARED_DATA 0x7ffe0000

- _KUSER_SHARED_DATA exists here in all versions of windows since XP
  - x64 dt ntdll!_KUSER_SHARED_DATA 0x00000000`07ffe000
- Big structure, various aspects contextually
  - +0x300 SystemCall
  - +0x304 SystemCallReturn
- Pointers are NULL in WoW64 & x64
- 0x7ffe0000 mapping
  - Read-only, 4096 bytes
  - Previously famous for its executable code

poi(0x7ffe0300)

- 0:003> u poi(0x7ffe0300)

  ntdll!KiFastSystemCall:

  77a164f0 8bd4 mov edx,esp

  77a164f2 0f34 sysenter

  ntdll!KiFastSystemCallRet:

  77a164f4 c3 ret

- Constraints for system call with parameters
  - Control, at least partially eax
  - esp points to our parameters
- Can be used without esp, just no parameters
  - Xiabo’s example, MS08-078-
    - heap spray 0x0a0a11c8 into eax
    - 0x0a0a11c8 == 0x7ffe0300 – vptr offset used in call

Why so Serious

- Tradition dictates that we don't use system calls for windows exploits
  - System call numbers are not set in stone like the unices
  - Not really considered an external interface
  - Many are undocumented, all subject to change
  - Effective use is contextual & moderately non-trivial
- We already have to do per service-pack, per-language per-version et cetera exploits
  - Syscall numbering relatively stable (compared to say stack offsets between two SPs)

Go West

- What to do with a free system call?
  - Ring 3 to Ring 0 Exploits
    - Don’t forget all those win32k system calls!
    - And that applications can have their own
    - Win32k return values are variable, not NTSTATUS
      - Many kernel memory leaks via eax
  - Scavenge from existing deterministic data
    - Surprising volume of data fit for use
    - TIB is near-deterministic
      - Awesomely contains pointers to the threads base, end and current SEH record
      - Awesomely is writeable
    - NtProtectVirtualMemory(0xFFFFFFFF, 0x7ffdf008, 0x7ffdf010, 0x00000040)
      - *Probably* makes a large-portion of the main thread’s stack executable
      - SEHOP/SafeSEH + SEH pointer re-ordering save the day here; otherwise, we’d cause an exception and be done,

Near Determinism

- The TIB is for thread x is at…
  - TIBs are laid out near-sequentially from 0x7ffdf000
    - Skips over conflicting pre-existing mappings
    - 4-bits of entropy, group into sets of 16
  - In practice there are generally only 3 mappings that conflict
    - AnsiCodePageData at 0x7ffb0000
    - ReadOnlySharedMemoryBase at 0x7f6f0000
    - PEB who will show up at a range from 0x7ffb0000 and 0x7ffdf0000
    - Large volumes of threads will cause conflicts at other mappings
- Sans those exceptions, the first thread sets @ 0x7ffdf000-0x7ffdf000+16*sizeof(TIB), the second at 0x7ffdf000+17*sizeof(TIB) – …
  - Past 0x7f6f0000 & until 0x7C100000, we can know the TIB for sets of threads [x-x+16] at 100%

Thread Stack Layouts

- Each thread has it's own stack & stack's start towards low memory
  - New stacks exist at a positive offset from the last (generally)
  - Pre-thread stack allocation handled via NtSetInformationFile() opcode 0x29
    - Walks user-address space returning first address free that is large enough
    - Calls function that finds free memory a random number of times- derived from system time
- The lower end of memory tends to also have
    - Non-DLL based file mappings
    - Executable image being executed
- As thread numbers increase…
  - Layouts become more predictable
  - Stack layouts become near-deterministic
    - Trend becomes slightly more profound when thread stack's grow beyond DLL mappings
      - Seems to have had recent changes- used to reduce entropy on average to 3-4 bits, obscenely high-numbers of threads seem to still come close

Thread Spraying

- Per MSDN, max stack for all threads is 32M
  - Large chunks conditionally under attacker control
- For a given set of 16 threads..
  - Their stack's will fall loosely in the same range
    - Id est 0x020XX000 through 0x02FXX000
      - Except for those that don't …
        - 3 threads on average fall in 0x01XXXXXX or 0x03XXXXXX
- Thus a CreateThread() is in our favor
  - Create groups of mirror threads created in sequential order
  - Grouping allows us to create sections of semi-contiguous memory
    - Still contains gaps (end of guard to next stack ~1.5M)
    - Contains guard pages between stacks (accesses = extension)
    - Changes make address prediction less likely- borks our heap technique

CreateThread() as malloc()

- Lots of threads in transient states is bad
  - What if thread exits?
  - What is it's current state?
  - What about the other XX threads you created?
  - Techniques with timing requirements are inherently unstable (generally)
- That's okay!
  - NtSuspendThread() – takes all static parameters
  - NtWaitUserMessage() – takes no parameters
    - May not be synonymous with NtSuspendThread()
    - If there's no WM_*'s to be delivered, it blocks
    - No parameters makes it usable from the heap
  - Iterate across thread groups TEBs calling NtVirtualProtect()
  - CreateThread() -> NtSuspendThread()
    - Allocates a stack whose location we can almost guess
    - Creates a TIB record with a ** to the stack at an address we can guess
    - Fixes it in place, thread X will never exist again, only thread X+y
    - Can serve as a the basis for a malloc()/VirtualProtect() primitive

Guessing your thread number

- All fine and well, but..
  - We have no way of knowing how many threads the application currently has
  - Or do we?
  - Microsoft was kind enough to never randomize IP IDs
  - We can tell how many connections have occurred since our last
  - Does not tell us what was connected to
  - Serves as a boolean to know if our threads were grouped next to each other
- Slower servers obviously more advantageous
  - Patience !
  - Each probe can be a thread spray'n'lock
  - Each probe can be anything that retrieves an IP packet
  - Not strictly necessary, but improves first-guess accuracy
  - Also, way more cool

Guessing your thread number

- Must model application's thread usage during idle times
  - Use as offset in calculations
  - I have yet to encounter 'random idle thread counts'
- Experience has shown that over-estimating number of pre-existing threads is helpful
  - More threads in larger groups allows us to divide in half and estimate our middle
- Favors server side attacks
  - Things like Chrome's sandbox help actually
  - It's okay, not all targets are sitting aggregating porn blogs & viagra emails
  - Targeted client-side attacks are effective against organizations
  - Generally not effective against individual targets

Finding your target's TIB

- Now that we have groups of threads locked into place
  - We need to take advantage of it
  - We want to be able to accurately target VirtualProtect() to a given thread
- Let…
  - S = sizeof(TIB)
  - C = number of estimated threads
  - N = the number of threads grouped together under your control
- Target thread sets TIB address = 0x7ffdf000 – 16*S – C*S – (N/2)*S
  - Requisite that there are enough threads to cross the boundary at which the conflicting PEB et al mappings effect no longer exists, thus 16*S
- Surprisingly stable & accurate

Egg hunting

- **System calls are nice**
  - Tend to return 0xC0000005 and not crash
  - In any case where the return value can be discerned remotely, the attacker can probe memory
- Can be used in two manners—
  - Ensure that guessed pointer is valid memory- dereference check
  - Ensure that guessed pointer points to data we want- valid data check
- Many system calls are not usable for data checks- require pointers
  - OBJECT_ATTRIBUTES & UNICODE_STRING are evil- contains pointer in structure
  - ‘1 pointer rule’
- Permissions matter- some system calls require LOCAL_SYSTEM for success
  - NtAddAtom() , NtFindAtom() combinations
- NtSetInformationFile() opcode 0x29 – minor data checks
- NtQueryVirtualMemory(), NtAllocateVirtualMemory() et cetera
- Several win32k system calls- less standardized, more usuable system calls

IOActive™
COMPREHENSIVE COMPUTER SECURITY SERVICES

All together now!

- Use IP ID to determine how busy server is
  - Group as many sequential threads together as reasonable / possible
- Load threads with both shellcode and egg hunting tag(s) (id est NtAddAtom() / NtFindAtom())
  - All of these threads are to be suspended
  - Determine TEB address, make stack executable for these threads
- Use system calls to fine tune stack address guesses
  - Which system calls to make are contextual– best ones require LOCAL_SYSTEM
    - Others work, but higher probability that data matched
- Return into shellcode on stack

Other techniques & work in

- PEB is highly guessable – approximately 1 in 12 chance give or take
  - PEB contains, among others, pointer to .text segment
  - For contexts akin to:

    lea reg32, [user+user]

    call [reg32]
  - Use this if you can! Far less complex!
- We can use system calls to probe memory
  - 0x7ffdf000, NtProtectVirtualMemory()
    - Success? Not PEB, subtract sizeof(TIB), repeat
    - Fail? PEB!
  - Return into ROP
- Not 32-bit specific!
  - If we can find a way to further reduce this, we broke x64 too!

Final thoughts

- ASLR effectiveness is non-constant!
  - Sometimes can be manipulated by attackers to their advantage
- Many possibilities for side-channel attacks
  - Correlations can sometimes be made between non-address space related data
- Deterministic pointers to randomized data is potentially dangerous
  - Akin to not randomizing dyld
- More of this type of stuff is there to be found!
  - x64 is the big target
  - Win32 and WoW64 more or less broken
    - Albeit contextually

- Questions?