Automated vulnerability analysis of zero sized heap allocations

April 2010

Julien Vanegue (jvanegue@microsoft.com)

Microsoft Security Engineering Center (MSEC)
Penetration testing team
Zero alloc: what is it?

Scenario 1:

UINT size = readfromuntrusted();
PCHAR ptr = malloc(size);

if (ptr == NULL) return -ERR;
memcpy(ptr, data, size-1);

Test for NULL does not replace a 0-size check!
Scenario 2:

PSTRUCT data = readfromuntrusted();
if (data->nbr > MAXSHORT) return -ERR;
UINT size = data->nbr * sizeof(TYPE);

PSTRUCT2 ptr = Alloc(size);
if (ptr == NULL) return -ERR;
ptr->field = data->field;

Failure to test lower bound of allocated size
Scenario 3:

UINT size;
UCHAR *ptr;

size = readuntrusted() + sizeof(TYPE);
ptr = Alloc(size);
if (ptr == NULL) return -ERR;
memcpy(ptr, (TYPE*)buf, sizeof(TYPE));

Fail to test upper bound of modulo arithmetic
Overview

• ISO C99: “If the size of the space requested is zero, the behavior is implementation defined: either a null pointer is returned, or the behavior is as if the size were some nonzero value, except that the returned pointer shall not be used to access an object.”

• Similar characterization are given in ANSI and earlier ISO C standards: the allocator can choose to return NULL or a “valid” pointer. Most allocators return the address of a valid heap chunk.

• This is not a class of vulnerability, more of an analysis technique to find new buffer overflows when the size of a heap memory chunk fails to follow certain constraints.

• We have an internal tool (Havoc/0-Alloc) that finds all such code occurrences automatically in large depots of C source code.
Presentation outline

1. Vulnerability assessment
   - Zero allocations tests
   - Ex1: A first historical zero allocation kernel bug
   - Ex2: Local escalation of privilege kernel bug due to a zero allocation

2. Static analysis of zero allocations
   - Automated deduction tools and techniques
   - HAVOC: a heap-aware verifier for C programs
   - Ex3: finding complex bugs with HAVOC

3. Generalization: near-zero allocation analysis
   - Ex4: Remote user-land heap overflow vulnerability
Part I

Vulnerability assessment
A. Testing: is your environment exposed?
Dummy kernel driver

DriverEntry()
{
  (...)
  for (i = 0; i < 5; i++)
  {
    //Main kernel memory allocator
    char *ptr = (char *) ExAllocatePoolWithTag(0, 0, 0);
    KdPrint(("A: %08p \n", ptr));
  }
  (...)

Result when loaded as testdrv

testdrv: A: 89DC61B8
testdrv: A: 8980E328
testdrv: A: 89DA11A8
testdrv: A: 8976E7B8
testdrv: A: 89963E08
Root cause: ExAllocatePoolWithTag(0) returns a valid chunk

POOLCODE _ExAllocatePoolWithTag@12 proc
(...)
    mov     edi, [ebp+NumberOfBytes]
(...)
    test    edi, edi        ; if (NumberOfBytes == 0)
    jnz     short loc_4E4132
    inc     edi              ; NumberOfBytes = 1;
POOLCODE:004E4132:
    add     edi, 0Fh        ; ROUNDUP(NumberOfBytes);
(...)

User-land tests

Compiled with: cl alloc0-user.c /link /dynamicbase as to enable ASLR

c:\win7rtm>alloc0-user.exe
Malloc returns : 000518F8 00051908 00051918 00051928 00051938 00051948 00051958
HeapAlloc returns : 00263FD8 00263FE8 00263FF8 00264008 00264018 00264028 00264038
VirtualAlloc returns : NULL NULL NULL NULL NULL NULL


c:\win7rtm>alloc0-user.exe
Malloc returns : 000818F8 00081908 00081918 00081928 00081938 00081948 00081958
HeapAlloc returns : 00223FD8 00223FE8 00223FF8 00224008 00224018 00224028 00224038
VirtualAlloc returns : NULL NULL NULL NULL NULL NULL


c:\win7rtm>alloc0-user.exe
Malloc returns : 000B18F8 000B1908 000B1918 000B1928 000B1938 000B1948 000B1958
HeapAlloc returns : 003C3FD8 003C3FE8 003C3FF8 003C4008 003C4018 003C4028 003C4038
VirtualAlloc returns : NULL NULL NULL NULL NULL NULL


c:\win7rtm>alloc0-user.exe
Malloc returns : 000A18F8 000A1908 000A1918 000A1928 000A1938 000A1948 000A1958
HeapAlloc returns : 00413FD8 00413FE8 00413FF8 00414008 00414018 00414028 00414038
VirtualAlloc returns : NULL NULL NULL NULL NULL NULL


c:\win7rtm>alloc0-user.exe
Malloc returns : 006318F8 00631908 00631918 00631928 00631938 00631948 00631958
HeapAlloc returns : 00163FD8 00163FE8 00163FF8 00164008 00164018 00164028 00164038
VirtualAlloc returns : NULL NULL NULL NULL NULL NULL
Zero allocs on UNIX

<table>
<thead>
<tr>
<th>Operating System</th>
<th>User-land exposed</th>
<th>Kernel-land exposed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux (Debian / 2.6)</td>
<td>Yes</td>
<td>Yes (*)</td>
</tr>
<tr>
<td>Linux (Debian / 2.6 / PaX)</td>
<td>Yes</td>
<td>No (*)</td>
</tr>
<tr>
<td>FreeBSD (5.5)</td>
<td>No (**)</td>
<td>Yes</td>
</tr>
<tr>
<td>NetBSD (3.0.1)</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>OpenBSD (4.4)</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Solaris (10)</td>
<td>Yes</td>
<td>Yes (***)</td>
</tr>
<tr>
<td>MacOsX (Leopard)</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

(*) Default linux kmalloc returns constant 0x10 (vulnerable to NULL ptr dereference unless mmap protection). PaX returns 0xFFFFF000 which is unmapped memory.

(**) Userland FreeBSD allocator returns a constant 0x800 on malloc(0). We suppose this is not exploitable (unless someone can exploit userland NULL derefs..)

(***) Solaris kernel allocator returns NULL on zero size. Checking the return of such function is necessary, else NULL dereference happens.
B. Examples of fixed problems

Disclaimer: The names have changed to protect the guilty.
Kernel bug due to a zero allocation

__kernel_entry BOOL MyNTSyscallEntryPoint(HANDLE h, PSTRUCT1 pData)
{
    PSTRUCT1 safedata = Handle2Ptr(h);
    DWORD count;
    try {
        PSTRUCT2 curData = ProbeAndRead(pData); // user-controlled data
        if (curData.flags & COND_FLAG) // user-controlled test
            count = (curData.field1 * sizeof(HANDLE)) + // count can be 0
                    (curData.field2 * sizeof(HDRTYPE)) +
                    (curData.field3 * sizeof(DWORD));
    (...)
    retval = _SetData(safedata, curData, count); // calling internal function
    (...)
}
Kernel bug continued

// Precondition: count can be 0
BOOL _SetData(PSTRUCT1 safedata, PSTRUCT2 curData, DWORD count)
{
    PSTRUCT2 tmparray;
    if (pcur->flags & FLAG_ENABLED) { // we will enter here
        tmparray = UserAllocPool(count, USERTAG_POOL); // zero-allocation happens!
        if (temp == NULL) return -ERR; // The check is bypassed
        safedata->array = tmparray + sizeof(BIGSTRUCT); // Dangling ptr arithmetic
        // Copying 0 bytes: nothing happens ! (but see next slide)
        try { RtlCopyMemory(safedata->array, curData->array, count); }
        except { return -ERR; }
    }
    return ESUCCESS;
}
__kernel_entry NTSTATUS NtOtherRelatedSyscall(HANDLE h)
{
    PSTRUCT1 p1;
    PSTRUCT2 p2;
    (...)
    p1 = Handle2Ptr(h); // We look up the same kernel structure
    if (p1->flags & FLAG_ENABLE) { // this is the previous corruption condition!
        if (p1->array == NULL) return -ERR;
        // p1->field2 is 0 but p1->array is dangling!
        // Invalid array lookup (but no escalation of privilege, crash only)
        p2 = p1->array[p1->field2];
    }
    (...)
}
Early conclusions

• A zero allocation does not lead to a systematic problem (copying few bytes of memory in a zero allocated chunk is generally safe)
• When a real problem shows up, it is not always a severe security threat (sometimes only a crash)
• A local check is not enough to assess the severity of a bug (dangling kernel pointers can be reused across system calls)

A zero allocation can also lead to a real security vulnerability. We now give such example...
Ex2: zero alloc leads to heap overflow

```c
__kernel_entry NtSomeNewSyscall(PVOID lpParam)
{
    PSETTING psetting;
    PLARGE_STRING pls;
    if (lpParam) {
        try {
            psetting = (PSETTING)lpParam;
            // No validation on count (can be 0!)
            count = ProbeAndReadUlong((&(psetting->DataLength)) + 1);
            pls = UserAllocPoolWithQuota(count, USERTAG_FLAGS); // Zero allocation
            RtICopyMemory(pls, (PBYTE) lpParam, count); // Nothing happens as count = 0
        }
        retval = _NewInternalFunction(NULL, pls); // Internal function is called
    }
    (...)
```
EoP bug explained (continued)

Void _OtherInternalFunction(PVOID p, PLARGE_STRING lparam)
{
    LARGE_UNICODE_STRING str;
    PLARGE_STRING pstr;
    if (p == NULL) { // we satisfy this condition (first param)
        pstr = lParam; // Remember lParam is dangling
        cbAlloc = pstr->Length + sizeof(WCHAR); // Alloc size set from uninitialized data
        // Another uncontrolled allocation happens
        str.Buffer = UserAllocPoolWithQuota(cbAlloc, USERTAG_FLAG);
        try {
            str.Length = pstr->Length; // Uninitialized length is used
            memcpy(str.Buffer, pstr->Buffer, str.Length); // The worse can happen...
            str.Buffer[str.Length / sizeof(WCHAR)] = 0; // ... twice.
        }
        (...)
    }
}
Part II
Automated analysis
The need for automated analysis

- Even though we do a lot of code review, human error can lead to missing bugs.
- Important is not how many vulnerabilities you find, but how many you miss.
- Some particular classes are recurrent. We want to fix those and prove the absence of bug with higher assurance before shipping (for older products: before customers hit them).
- We introduce the use of HAVOC for property-based security assessment of C source code.
Automated techniques
Dataflow analysis (DFA)

DFA helps to find approximate conservative solution to hard problems.

• Most compilers (GCC, LLVM, Phoenix, etc) use dataflow analysis techniques and intermediate language representation during code translation. (Ex: Static Single Assignment – SSA – form)
• With some implementation effort, an analyst can build a program analyzer within the compiler.
• Many security bugs can already be detected with such technique (ex: memory leaks, double free, uninitialized vars, etc)

**Good:** fast, scalable, conservative (no forgotten behaviour)
**Bad:** hardcoded algorithms, *too conservative (false positives)*
A data dependence graph

- Data dependence graphs (DDG) are built first based on use-def chains. In below graph, nodes are variables, green are malloc nodes while red are free nodes. Arrows are dependences between variables.

- Reachability, liveness or value flow analysis can be performed on top of DDG.

- Analysis is more refined when control predicates are retained in the IR.
Automated technique 2: Model checking

- Model checking is an implementation of formal verification of programs that operates at a higher level than data-flow analysis. MC is suitable to resolve more complex properties in big systems such as concurrent hardware and programs.

1. A specification formula $F$ is written in a chosen formal logic
   ex: Linear Temporal Logic (LTL), but there are many others...
   ex: $G (\text{Alloc}(x) \Rightarrow x \neq 0)$ stands for: Always ($G$) if Alloc($x$) is true then $x$ is not 0

2. The program to analyze is represented as a state-transition system
   - Nodes are states of the analyzed program.
   - Transitions represent changes in the values of system variables.
   - Our example introduces a predicate Alloc($x$) that is only true on allocation sites

See next slide..
Model checking (cont)

3. The algorithm checks statically if the formula is satisfied at runtime:
   - Check exhaustively all sequence of transitions in the system.
   - If any system path makes the formula false, there is a state where
     program specification is violated (e.g. we found a bug)

   **Good:** Universal technique, completely automated.
   Many existing tools (SPIN, SLAM, BLAST, SPOT, etc)

   **Bad:** make state space explicit (combinatorial explosion)

• *Model checkers can be used in conjunction with flow analyzers to verify
  complex program properties expressed in a formal logic.*
We can reach a state where \( G(\text{Alloc}(x) \Rightarrow (x \neq 0)) \) is false.
Automated techniques (3)
Theorem Proving (TP)

• TP is a mathematical technique of logical proof checking that can reason precisely about call-free/loop-free programs.
• A verification condition (VC) is constructed that captures all paths within each procedure of the analyzed program.
• Pre-conditions and post-conditions (e.g. assert(F)) are enforced/checked at specific program points.
  – Pre/post-conditions can be written manually, prover will check them.
  – We can write a wrapper tool that will also generate pre/post-conditions automatically when the program locations of interest can be easily guessed.
Theorem proving

• Checking a formula F can have two results:
  
  – For program points of interest, no scenario can violate the requested pre/post condition. The program is free of such mistakes at checked program points. Or..
  
  – We cannot prove F at some program point P, the verifier emits a warning and provide the counter-example (faulty) trace ending at P.

Good: Precise technique (no approximation), scales well (modular analysis).
Bad: May require users to specify procedure contracts manually

We have applied TP on many kernel components (some of 1 million LOC)
int f(UINT val, bool mode)
{
    UINT size, pad = 0;
    if (val > 8) return ERR;
    size = val * 2;
    if (mode == M32)
        pad = sizeof(T32);
    else if (mode == M64)
        pad = sizeof(T64);
    size += pad;
    PTYPE ptr = Alloc(size);
    __requires(size != 0)
(...)

int f(UINT val, bool mode) { /* Inferred formula */
    UINT size, pad = 0;
    if (val > 8) return ERR;
    size = val * 2;
    if (mode == M32)
        pad = sizeof(T32);
    else if (mode == M64)
        pad = sizeof(T64);
    size += pad;
    PTYPE ptr = Alloc(size);

    __requires(size != 0)
int f(UINT val, bool mode) **Inferred formula**
{
    UINT size, pad = 0;  // f1: pad=0
    if (val > 8) return ERR;
    size = val * 2;  // f2: f1 && (size == val*2) && (val <= 8)
    if (mode == M32)
        pad = sizeof(T32);
    else if (mode == M64)
        pad = sizeof(T64);
    size += pad;
    PTYPE ptr = Alloc(size);

    **__requires(size != 0)**
int f(UINT val, bool mode)  **Inferred formula**
{
    UINT size, pad = 0;
    f1: pad=0
    if (val > 8) return ERR;
    size = val*2;
    f2: f1 && (size == val*2) && (val <= 8)
    if (mode == M32)
        pad = sizeof(T32);
        f3: pad=sizeof(T32) && (size == val*2) && (val <= 8)
    else if (mode == M64)
        pad = sizeof(T64);
    size += pad;
    PTYPE ptr = Alloc(size);

    __requires(size != 0)
int f(UINT val, bool mode) **Inferred formula**
{
    UINT size, pad = 0;
    if (val > 8) return ERR;
    size = val*2;
    f1: pad=0
    if (mode == M32)
        pad = sizeof(T32);
    else if (mode == M64)
        pad = sizeof(T64);
    f2: f1 && (size == val*2) && (val <= 8)
    size += pad;
    f3: pad=sizeof(T32) && (size == val*2) && (val <= 8)
    PTYPE ptr = Alloc(size);
    f4: pad=sizeof(T64) && (size == val*2) && (val <= 8)

    __requires(size != 0)
int f(UINT val, bool mode)  **Inferred formula**
{
    UINT size, pad = 0;
    if (val > 8) return ERR;
    size = val*2;
    if (mode == M32)
        pad = sizeof(T32);
    else if (mode == M64)
        pad = sizeof(T64);
    size += pad;
    PTYPE ptr = Alloc(size);

    // Requires size != 0
}

f1: pad=0
f2: f1 && (size == val*2) && (val <= 8)
f3: pad==sizeof(T32) && (size == val*2) && (val <= 8)
f4: pad==sizeof(T64) && (size == val*2) && (val <= 8)

f5: (size == val*2 + pad) && (val <= 8) &&
    (pad==0 || pad==sizeof(T32) || pad==sizeof(T64))

__requires(size != 0)
int f(UINT val, bool mode)  
{  
UINT size, pad = 0;  
f1:  pad=0  
if (val > 8) return ERR;  
size = val*2;  
f2:  f1 && (size == val*2) && (val <= 8)  
if (mode == M32)  
    pad = sizeof(T32);  
f3:  pad=sizeof(T32) && (size == val*2) && (val <= 8)  
else if (mode == M64)  
    pad = sizeof(T64);  
f4:  pad=sizeof(T64) && (size == val*2) && (val <= 8)  
size += pad;  
f5:  (size == val*2 + pad) && (val<=8) &&  
    (pad=0 || pad=sizeof(T32) || pad=sizeof(T64))  
PTYPE ptr = Alloc(size);  
f6:  0 <= size <= 16 + MAX(0, sizeof(T32), sizeof(T64))  
    __requires(size != 0)  
Precondition violation!

In practice, we analyze each path separately to avoid approx. at merge points.
Automated analysis : HAVOC

• HAVOC: Heap Aware Verifier for C programs
  – A static analysis tool for source code:
    • Based on the Boogie theorem prover
    • Uses a documented code IR : BoogiePL
    • Decision procedure based on constraint solving (Z3)
    • User can specify properties to be checked via annotations

• Project developed at Microsoft Research in the RiSE team
  – Plug-in for the Microsoft C/C++ compiler
  – Detailed user manual

• MSEC use HAVOC for the assessment of software security properties:
  – To find new bugs.
  – To find variants of existing bugs.
HAVOC: The big picture

C program → BoogiePL

Annotations → Memory model

BoogiePL program (.bpl file)

Boogie VCGen

Verification condition

SMT Solver (Z3)
Decision procedures for types, lists, arrays

Verified

Warning
HAVOC analysis modes

- HAVOC can run either as a local checker (intra-procedural analysis only) or a global checker (inter-procedural analysis)
  - In local checking mode, function parameters and return values of callees are always assumed untrusted.
    - Leads to more false positives.
    - Analysis converges fast.
  - In global checking mode, preconditions are propagated to called functions at call sites. Parameters and return values are no more assumed untrusted.
    - Global checking performs inter-procedural inference using a fixed-point algorithm over the call graph.
Example of HAVOC annotations

• Our property of interest is very easily expressible using preconditions over the allocation API.

__requires__(Size != 0)
PVOID ExAllocatePool(POOL_TYPE PoolType, SIZE_T Size);

• The analyzer can prove that 98% of allocation sites are secure wrt. zero allocations using local analysis only. The rest shows up in the warning list.
  – 100 warnings for 1 million LOC (numbers differ for other properties).
  – Can be refined with global analysis to obtain perfect result. The analysis becomes much more expensive to resolve the remaining 2% of cases.

• A simple local check found multiple escalation of privilege bugs in the windows kernel.
Using HAVOC to find complex bugs in the remote kernel attack surface

- Inter-modules bugs are hard to find with static analysis as they involve multiple interfaced components implemented in many millions of LOC.
- How to find all such bugs quickly?
  - Restrict the analysis to the server driver.
  - Understand what happens behind interface calls (ex: IoDoRequest())
  - We enforce post-conditions on the parameters of the interface API

```c
__requires(size != 0)
BOOL IoDoRequest(PVOID data, UINT size);
```
Part III: generalization
Near-zero allocation analysis
Near-zero allocation analysis

• Sometimes zero allocations are correctly filtered by wrapper macros that ensures NULL is returned if 0 is passed as a size. Those API are obviously not vulnerable to zero allocations.
• Dangerous behaviour can still happen if the size variable is not properly handled. We give an example of such remote overflow bug.
• We currently have no automated analysis for finding those bugs (beside existing generic and noisy buffer overflow finders) but we found some by code review.
Remote heap overflow due to near-zero allocation bug

```c
STATUS DecodeTextString(PBYTE inputdata, DWORD aSize, PSTATUSINFO status) {
    //status->integercount is read on the wire (remotely controlled) and can be 0.
    eStatus = PKT_GetValueLength(inputdata, aSize, &status->integerCount, status);
    // Could allocate only 2 bytes
    status->tmpbuf = malloc(status->integerCount + 2);
    if (NULL == status->tmpbuf)
        return EOOMEM;
    // nothing happens as we copy 0 byte
    memcpy(status->tmpbuf, inputdata, status->integerCount);
    // byteCpd stays 0!
    status->bytesCpd += status->integerCount;
    // tmpbuff is 2 bytes and non initialized!
    bool res = HeaderFieldStringConvert(status);
    (...)
```
Remote near-zero alloc bug (2)

Bool HeaderFieldStringConvert(PSTATUSINFO status)
{
    if (NULL == status->finalbuf) {
        // We will enter here when the first header string entry is sent.
        // The first string has to be non-empty as byteCpd is well checked for 0 here
        if (0 == status->bytesCpd) return(-EINCOMPLETE);
        status->finalbuf = calloc(status->bytesCpd);
        if (NULL == status->finalbuf) return (EOOMEM);
    } else {
        // We enter here for next strings
        newlen = strlen(status->finalbuf);
        // No test if byteCpd is 0. The buffer will only grow of one byte
        status->finalbuf = realloc(status->finalbuf, (status->bytesCpd + newlen + 1) );
        if (NULL == status->finalbuf) return (-EOOMEM);
        // Each entry is concatenated (separated by a coma)
        *(status->finalbuf + newlen)= HEADER_COMMA_CHAR;
        newlen++;
    }
}(...)
Remote near-0 alloc bug (3)

Next in HeaderFieldStringConvert() :

(...)  
PCHAR charSet = (status->tmpbuff[0] & 07F) // Use of uninitialized data  
status->bytesCpd--; // BUG: if byteCpd was 0, bytecpd becomes very big!  
switch (charSet) // We might enter in any of the switch branch due to uninit charSet  
{  
  (...)  
  default:  
    // “finalbuf” overflow will happen in ConvertFromASCII()  
    eStatus = ConvertFromASCII(status->tmpbuff+1, status->bytesCpd,  
                                status->finalbuf + newlen));  
(...)

A regular string copy then happens with big size in ConvertFromASCII (heap overflow)
Conclusion

• We approach the complex problem of finding subclasses of buffer overflows automatically.
• Boundary values are generally more problematic for developers.
• We reduced the problem space to understand what happens when allocation size is zero or in the neighbourhood of zero.
• This restriction leads to the construction of a precise analyzer with very small false positive rate that scales to millions of LOC (need source code)
Related work

• Related people:
  – TWC MSEC Science UK team.
  – Microsoft Prefast/CSE team (Windows static analysis team).

• Related properties:
  – Forgetting to test NULL after allocation is a different problem. We don’t need a complex analysis to find such errors.
  – Analysis of large allocations leading to the NULL page being mapped is a different property. Zero allocations generally don’t lead to memory leaks (hint: find unbounded allocation sites!)
Questions?

• Thanks for attending.

• Greets:
  – Josh Lackey, Damian Hasse & Alex Lucas for their experienced feedback on the presented material.
  – Shuvendu Lahiri, Mark Wodrich, Matt Miller, Peter Beck, Thomas Garnier for their support in security vulnerability research in Microsoft.
  – Enrico Perla and the PaX team for their assistance on UNIX testing.

• For any enquiry: ivanegue@microsoft.com
Bonus slide: Exploitability / Mitigations
Windows Kernel Heap ASLR

MALLOC returns: 000518F8 00051908 00051918 00051928 00051938 00051948 00051958
HeapAlloc returns: 00263FD8 00263FE8 00263FF8 00264008 00264018 00264028 00264038

MALLOC returns: 000818F8 00081908 00081918 00081928 00081938 00081948 00081958
HeapAlloc returns: 00223FD8 00223FE8 00223FF8 00224008 00224018 00224028 00224038

MALLOC returns: 000B18F8 000B1908 000B1918 000B1928 000B1938 000B1948 000B1958
HeapAlloc returns: 003C3FD8 003C3FE8 003C3FF8 003C4008 003C4018 003C4028 003C4038

MALLOC returns: 000A18F8 000A1908 000A1918 000A1928 000A1938 000A1948 000A1958
HeapAlloc returns: 00413FD8 00413FE8 00413FF8 00414008 00414018 00414028 00414038

MALLOC returns: 006318F8 00631908 00631918 00631928 00631938 00631948 00631958
HeapAlloc returns: 00163FD8 00163FE8 00163FF8 00164008 00164018 00164028 00164038

Heap ASLR on the windows kernel uses 5 bits of entropy (Windows 7 / Vista)