



Anti-Anti-Dump and nonintrusive tracers

deroko/ARTeam

Version 1.2 – April 2006

1.	Abstract.....	2
2.	Needed knowledge	3
2.1.	Offset independent code	3
2.2.	Retrieving kernel32.dll base and APIs	4
2.3.	Loader injector	7
2.4.	Hybrid hooking approach	8
2.5.	What next?	10
3.	Memory Manager	11
3.1.	Expanding program memory	11
3.2.	Memory manager for VirtualAlloc and VirtualFree	12
3.3.	Problems with Delphi code	16
3.4.	Memory manager conclusion.....	18
4.	Nonintrusive tracers for Memory Manager	19
4.1.	Writing nonintrusive tracer	20
4.2.	Using PAGE_GUARD with nonintrusive tracer	22
4.3.	PAGE_GUARD in weird conditions aka KiUserExceptionDispatcher improved	24
4.4.	Logging Access	28
4.5.	Invoking driver from tracer.....	29
4.6.	Making stealth nonintrusive tracers	29
4.7.	Nonintrusive tracer conclusion	33
5.	Loader for Loader.....	34
5.1.	Loader for Loader with injected code.....	34
5.2.	Loader for Loader without injected code.....	35
5.3.	Nonintrusive tracers for Debugged process.....	36
6.	Debugging injected code - tips	40
7.	Conslusion	41
8.	References.....	42
9.	Greetings.....	43

Keywords

coding, hooking, anti-dump, nonintrusive, memory manager



1. Abstract

Many protections nowadays try to avoid dumping by cutting parts of code and reallocating them in new buffers. Such buffers are sometimes pain to repair and are place where most people give up. I can name a few protections using this trick: Armadillo, ASProtect SKE, krypton etc. What I have seen so far are olly scripts to repair “cutted code” or eliminated IAT or even external tools. Users of such scripts, tools basically have no idea how to approach to this anti-dump problem. Using scripts and external tools will not result in learning new things.

I’m planning to show some ideas, I have right now which might be used to overcome such problems as cutted code but this isn’t going to be easy to understand. Prior to reading this article you should be familiar with asm programming, PE file format and hooking. Well you have to know basics because I will cover most of important parts in this article. Also I will show you have to write tracer for debugged process.

I think that these tricks haven’t been used yet in RCE or I haven’t seen it. Anyway I’m going to cover some interesting aspects of RCE.



2. Needed knowledge

If you know already all this stuff go to section 3, because I will cover basics of offset independent code and it's injection into target process.

2.1. Offset independent code

Offset independent code is code that is able to execute itself in any given memory location. Such codes are obviously viruses and shell codes, but also some protectors as well use offset independent code. The main question in offset independent code is its ability to access data. For this reason virus writers are using "delta" offset to access variables used by viruses. I'll go fast on this subject and show code immediately and how to reference variables in offset independent code:

```
delta:      call    delta
            pop     ebp
            sub     ebp, offset delta
            ...
            mov     eax, [ebp+kernel32]
            ...
            call    [ebp+GetModuleHandleA]

kernel32    dd      ?
GetModuleHandleA dd    ?
```

As you may see manipulating data in offset independent code is not that hard. Also there are some rules in offset independent code:

- Always use ebp as delta, also esi,edi and ebx may be used as delta because they won't be changed during API calls. I use ebp always because esi/edi combination is used for copying data, and I like to have ebx free or as pointer to some important data (eg. It points to PE in all of my viruses allowing me fast access to any needed file of PE file during infection)

There is also trick to play with delta introduced by Super/29a and later described by Benny/29a [1] to make your compiled code smaller:

```
delta:      call    delta
            pop     ebp

            mov     eax, [ebp+kernel32-delta]
            call    [ebp+GetModuleHandleA-delta]

kernel32    dd      ?
GetModuleHandleA dd    ?
```

Infact, this is very good trick, but since we don't care a lot about code size now, we may use previous method because it is more readable during debugging and requires less typing while programming offset independent code.

Ok, that's all I had to tell you about offset independent code. Don't learn its definition (you will find a lot of them), learn principles.



2.2. Retrieving kernel32.dll base and APIs

Next important issue for any offset independent code is calling APIs in Win32 environment. To be able to do this we first need to locate base of kernel32.dll and there are couple of ways we can accomplish this:

- scanning SEH
- PEB trick by Ratter/29a
- Hardcode value from our loader

Scanning SEH – first we need to know layout of struct describing SEH:

```
kd> dt nt!_EXCEPTION_REGISTRATION_RECORD
+0x000 Next          : Ptr32 _EXCEPTION_REGISTRATION_RECORD
+0x004 Handler       : Ptr32
kd>
```

Every SEH chain will endue with handler pointing to some value in kernel32.dll, of course since this is last EXCEPTION_REGISTRATION_RECORD Next will be set to -1, and we know when we got address inside of kernel32.dll.

Sample code:

```
getkernelbase:
    pushad
    xor     edx, edx
    mov     esi, dword ptr FS:[edx]
__seh:
    lodsd
    cmp     eax, 0FFFFFFFFh
    je      __kernel
    mov     esi, eax
    jmp     __seh
__kernel:
    mov     edi, dword ptr[esi + 4]
    and     edi, 0FFFF0000h
__spin:
    cmp     word ptr[edi], 'MZ'
    jz      __test_pe
    sub     edi, 10000h
    jmp     __spin
__test_pe:
    mov     ebx, edi
    add     ebx, [ebx.MZ_lfanew]
    cmp     word ptr[ebx], 'EP'
    je      __exit_k32
    sub     edi, 10000h
    jmp     __spin
__exit_k32:
    mov     [esp.Pushad_eax], edi
    popad
    ret
```

Well this code is not very optimized, but it shows the logic. Scan SEH till we find Next == -1 and simple get address of handler and search for MZ and PE signatures. Once we find them we got kernel32.dll base.



PEB method – this trick was discovered and used by Ratter/29a, I will give you sample source, but for more explanation please refer to [2].

```
mov     eax, dword ptr FS:[30h]
mov     eax, dword ptr[eax+0ch]
mov     eax, dword ptr[eax+1ch]
mov     eax, dword ptr[eax]
mov     eax, [eax+8]
```

First we retrieve value of PEB, then we go to PEB_LDR_DATA and after that we enter into InInitializationOrderModuleList, first LIST_ENTRY points to ntdll.dll so we go to next list entry and voila we receive address of kernel32.dll base.

HardCoded value for our loader – this is very simple and doesn't require too much knowledge, we use GetModuleHandleA to retrieve base of kernel32.dll and store that value in our offset independent code.

```
pushs   <"kernel32.dll">
call    GetModuleHandleA
mov     kernel32, eax
```

loader:

...

kernel32 dd ?

Not much to talk about it, very simple and straight.

Retrieving API addresses is also very simple; once we get kernel32.dll we may write our own GetProcAddress to search EXPORT table of kernel32.dll for needed APIs.

Export table is located at offset 78h from PE and its layout is:

```
typedef struct _IMAGE_EXPORT_DIRECTORY {
    DWORD Characteristics;
    DWORD TimeDateStamp;
    WORD MajorVersion;
    WORD MinorVersion;
    DWORD Name;
    DWORD Base;
    DWORD NumberOfFunctions;
    DWORD NumberOfNames;
    DWORD AddressOfFunctions; // RVA from base of image
    DWORD AddressOfNames;    // RVA from base of image
    DWORD AddressOfNameOrdinals; // RVA from base of image
} IMAGE_EXPORT_DIRECTORY, *PIMAGE_EXPORT_DIRECTORY;
```



There are 3 important members of this struct for us:

```
DWORD  AddressOfFunctions;    // RVA from base of image
DWORD  AddressOfNames;        // RVA from base of image
DWORD  AddressOfNameOrdinals; // RVA from base of image
```

As you may see in comment those are RVAs simple diagram will make this clear:

array of name RVA		ordinals		array of functions RVA
+-----+		+-----+		+-----+
RVA1 name	<----->	ordinal1	<----->	RVA of API1
+-----+		+-----+		+-----+
RVA2 name	<----->	ordinal2	<----->	RVA of API2
+-----+		+-----+		+-----+
RVA3 name	<----->	ordinal3	<----->	RVA of API3
+-----+		+-----+		+-----+
RVA4 name	<----->	ordinal4	<----->	RVA of API4
+-----+		+-----+		+-----+
RVA5 name	<----->	ordinal5	<----->	RVA of API5
+-----+		+-----+		+-----+

First we scan through `AddressOfNames` searching for name of our API, and we must use index to know at which position of `AddressOfNames` we have found RVA for our API name. Latter that index is used as index into `AddressOfNameOrdinals` which is nothing more then array of words. Once we get our ordinal we use it as index for `AddressOfFunctions` to get RVA of our procedure, then simple add base of dll to RVA and you will get API. Simple as that.

To reduce size of viruses, and latter shellcodes and to avoid detection by scanning for strings, vx writers decided to move to hashes [3,4] using some home made hash, or `crc32` to search for APIs. The smallest and so far the fastest hashing algo was one introduced by z0mbie and consist of `rol/xor`:

```
__1:          rol          eax, 7           ;hash algo (x) by z0mbie
             xor          al, byte ptr [edx]
             inc          edx
             cmp          byte ptr [edx], 0
             jnz          __1
```

So instead of scanning for APIs by name we scan them by hash. Fast enough, and makes small code.

That's all about offset independent code and `kernel32.dll` and retrieving needed APIs. This is sort of generic so I keep all of this stuff in one file and load needed APIs very fast w/o need to code this thing all over.



2.3. Loader injector

Load injector is actually simple loader that will inject our offset independent code into target process and our injector will do all dirty job for us (hooking for example), maybe you are wondering why I'm not using dll injection? Simple I don't like having 2 extra source files in same folder, and as vx programmer I don't like anything that is not offset independent J Joke. Use whatever makes you happy, but remember, offset independent code is much nicer J and DLL injection is more common for C programmers J

Ok let see how to inject our offset independent code in our target. First we create process in suspended state using `CreateProcessA` as in any normal loader, then we simple use `VirtualAllocEx` and `WriteProcessMemory` to write our offset independent code in target. Then simple store hook at entry point of our target and wait till hook is reached, once we hit hook, redirect context to our offset independent code which will be responsible for hooking and doing all dirty jobs for us and then it will restore original bytes and return to entry point of our target process:

```
push    offset pinfo
push    offset sinfo
push    0
push    0
push    CREATE_SUSPENDED
push    0
push    0
push    0
push    offset progy
push    0
callW   CreateProcessA

push    PAGE_EXECUTE_READWRITE
push    MEM_COMMIT
push    2000h
push    0
push    pinfo.pi_hProcess
callW   VirtualAllocEx      ;allocate big enough block
mov     mhandle, eax

push    0
push    2
push    offset infinite
push    401000h
push    pinfo.pi_hProcess
callW   WriteProcessMemory ;store jmp $ at entry point

push    pinfo.pi_hThread
callW   ResumeThread

mov     ctx.context_ContextFlags, CONTEXT_FULL

__cycle_ep:
push    100h
callW   Sleep
```



```
push    offset ctx
push    pinfo.pi_hThread
callW   GetThreadContext

cmp     ctx.context_eip, 401000h
jne     __cycle_ep

push    pinfo.pi_hThread
callW   SuspendThread

push    0
push    size_loader      ;size of loader
push    offset loader    ;loader code
push    mhandle          ;allocated mem block
push    pinfo.pi_hProcess
callW   WriteProcessMemory

push    mhandle
pop     ctx.context_eip ;eip == my code

push    offset ctx
push    pinfo.pi_hThread
callW   SetThreadContext;set context

push    pinfo.pi_hThread
callW   ResumeThread    ;resume thread
```

If everything went as planned our target is executing offset independent code at this point. No more to say, lets advance further.

2.4. Hybrid hooking approach

Hooking, hmmm, there are two options that are coming to my mind, but only one fits our needs if we are planning to defeat protectors, those are IAT hooking and API hooking. IAT hooking is not an option; because packer will either manually find all APIs or will use GetProcAddress to find needed APIs so it is not very practical. More interesting is 2nd hooking approach which consist of storing hook at entry of API or ret/retn at the end of API so we may control its output.

Let's take a look at one API from kernel32.dll:

```
.text:7C809A81 VirtualAlloc    proc near
.text:7C809A81
.text:7C809A81      mov     edi, edi
.text:7C809A83      push   ebp
.text:7C809A84      mov     ebp, esp
.text:7C809A86      push   [ebp+arg_10]      ; flProtect
.text:7C809A89      push   [ebp+flProtect]  ; flAllocationType
.text:7C809A8C      push   [ebp+flAllocationType] ; dwSize
.text:7C809A8F      push   [ebp+dwSize]      ; lpAddress
.text:7C809A92      push   0FFFFFFFFh       ; hProcess
.text:7C809A94      call   VirtualAllocEx
.text:7C809A99      pop     ebp
.text:7C809A9A      retn   10h
.text:7C809A9A VirtualAlloc    endp
.text:7C809A9D      db     90h
.text:7C809A9E      db     90h
```



We are going to hook VirtualAlloc in such way that it will only call our code that will be responsible for memory allocating and freeing. Also note that various procedures exported in many dlls will end up eventually in kernel32.dll or ntdll.dll to call native APIs. So if we know what we are hooking we may also call later API that internally would be called by our hooked API. Look at VirtualAlloc, it will call VirtualAllocEx so we may simulate this and there is no need for us to save and execute original instructions, the ones overwritten with our hook. This also saves us some time because there is no need to use Length Disassemble Engine to determine length of instructions. Note that using Length Disassemble Engine and saving old bytes is not hard to do, it is very simple but for this article it is not necessary. When we are hooking at ret/retn we must use Length Disassemble Engine to locate ret/retn. The approach I use for hooking kernel32.dll is to run LDE and to lookup for ret/retn and check if those are padded with nops. Let see some code:

```

;ebx - where to redirect
;edi - pointer to api
;esi - CMD_RETN      - hook api at ret/retn
;      - CMD_ENTRY   - hook api at entry

CMD_RETN      equ      1
CMD_ENTRY     equ      2

hook_api:      pusha

                lea      ecx, [ebp+dummy]

                push     ecx
                push     PAGE_EXECUTE_READWRITE
                push     1000h
                push     edi
                call     [ebp+VirtualProtect]

                test     esi, CMD_ENTRY
                jz        __hook_at_ret

                mov      ecx, edi
                mov      al, 0e9h
                stosb
                add      ecx, 5
                sub      ebx, ecx
                mov      [edi], ebx
                jmp      __exit_hook

__hook_at_ret:  ;most APIs are padded with nop, if no nop, fail!!!
                push     edi
                call     ldex86
                add      edi, eax
                cmp      byte ptr[edi], 0c3h
                je        __check_ret
                cmp      byte ptr[edi], 0c2h
                jne       __hook_at_ret

                cmp      word ptr[edi+3], 9090h
                jne       __exit_hook          ;failed
                jmp      __hook_api

__check_ret:   cmp      dword ptr[edi+1], 90909090h
                jne       __exit_hook
```



```
__hook_api:      mov     ecx, edi
                  mov     al, 0e9h
                  stosb
                  add     ecx, 5
                  sub     ebx, ecx
                  mov     [edi], ebx

__exit_hook:     popa
                  retn
```

As you can see from source code this is “hybrid” hooking approach and this is all we need for this article. You will understand why I use this approach once we come to memory manager organization and redirecting allocated buffers in range of our protected program.

2.5. *What next?*

Next? I will show you how to write memory manager in ring3 and also how to write good nonintrusive tracer/logger for your target.



3. Memory Manager

Maybe you are asking why I need memory manager for Anti-Anti-Dump. It is very simple, and I'll try to explain it as simple as possible. When you run some protected app (aspr, armadillo for example), protector will allocate a lots of buffers where original code will be redirected. Sometimes it is hard, if not impossible, to force protector to store all data in program range so we can later dump it. Sometimes this is not possible due to fact that there is so much allocated buffers, they are allocated and released by protector itself during target unpacking. One simple approach is to hook VirtualAlloc, in near future VirtualAllocEx, and even NtAllocateVirtualMemory in ntdll.dll, and to return to proggy buffer that can be easily dumped by us later, once we hit oep, or polymorphic oep, wherever it may be stored. Like this protector will reallocate all data in our dumpable range which may be dumped later by one special plug-in for LordPE, which I wrote earlier but it will work like a charm under these conditions. To avoid too much memory usage and to keep track what is being allocated and released, I will use similar concept as is used by Intel CPU to translate virtual to physical memory. We may also organize our Memory manager as heap with linked lists but that approach requires more coding.

3.1. Expanding program memory

Let's say we have our program with size of 9000h bytes and we are not sure if any section in program may be overwritten by redirecting some buffer to its range. In such cases we are sure we need to allocate buffer in target program that maybe dumped later without a problem. In such conditions we are going to use loader to increase size of proggy range:

- CreateProcessA in suspended state
- read whole proggy memory with ReadProcessMemory
- use NtUnmapViewOfSection to free memory used by our target
- use VirtualAllocEx to allocate bigger buffer starting from same base address as target
- use WriteProcessMemory to write back original program

Sample code:

```
push    offset pinfo
push    offset sinfo
push    0
push    0
push    CREATE_SUSPENDED
push    0
push    0
push    0
push    offset proggy
push    0
callW   CreateProcessA

push    PAGE_READWRITE
push    MEM_COMMIT
push    c_size
push    0
push    -1
callW   VirtualAllocEx
mov     esi, eax

push    0
push    c_size
```



```
push    esi
push    c_start
push    pinfo.pi_hProcess
callW   ReadProcessMemory

push    c_start
push    pinfo.pi_hProcess
callW   NtUnmapViewOfSection

mov     eax, c_size
add     eax, NEW_MEM_RANGE

push    PAGE_EXECUTE_READWRITE
push    MEM_COMMIT or MEM_RESERVE
push    eax
push    c_start
push    pinfo.pi_hProcess
callW   VirtualAllocEx

push    0
push    c_size
push    esi
push    eax
push    pinfo.pi_hProcess
callW   WriteProcessMemory
```

c_start - base of progy
c_size - size of progy
NEW_MEM_RANGE - increased size of our target

After this we are injecting loader as offset independent code in target which will be responsible for hooking VirtualAlloc and VirtualFree to return memory ranges inside of new range of progy. But the real question is how we are going to write this memory manager.

3.2. Memory manager for VirtualAlloc and VirtualFree

We know where our new buffer is located? Yep, it is at the end of original progy range. To describe state of each page in my buffer I will also use table of 4byte long entries to describe state of each page (I will call it PTE). We have to keep track of every page/range allocated by VirtualAlloc because once VirtualFree is called we may mark pages as not used anymore and latter return them when new VirtualAlloc is called. Without this we may even overflow our buffer and cause page_fault if we don't keep track of allocated buffers.

First I wrote memory manager using structs to describe start, range and type of allocated buffers, similar to heaps but latter I figured that it is too slow, and leads to memory leaks. Then I remembered how it is all done in Intel CPU and Virtual to physical translation. Virtual Addresses are used as index into PDE/PTE which except physical frame also have state of each page (user/supervisor, present/not present, R/W etc.) held in 4bytes entry. So why not adopt this tech and do similar thing in ring3 and write similar memory manager? It is fast, uses page index to access data that keeps state of each page, etc.



After a little bit of thinking I got what I needed, my Page Entry:

```
31                                2    1    0
+-----+-----+-----+-----+
| FIRST PAGE INDEX                | R | P |
+-----+-----+-----+-----+
```

P - present bit indicates if page is committed or it is free page

R - reserved bit set only of VirtualAlloc is called with MEM_RESERVE

FIRST PAGE INDEX aka FPI - holds index of first page, used to determine size of committed block

And buffer layout is:

```
+-----+-----+-----+-----+
| Progy   | VirtualAlloc/Free buffer | PTE      |
+-----+-----+-----+-----+
```

Your PTE size should be = (buffer/1000h)*4 in my case I allocate 1000 pages and 4 pages to describe state of each page.

To access PTE you need index of a given page, and to get index is very simple. Lets say that our progy starts at 400000h and ends at 500000h, end of progy range is start of our buffer, PTE is located in last 4 pages of our buffer so:

```
mov     esi, [edi+memstart] ;esi=500000h
add     esi, NEW_MEM_RANGE ;esi+1004000h
sub     esi, 4000h          ;structs for memory manager(PTEs)
```

memstart = end of progy

NEW_MEM_RANGE = 1004000h ;1000*1000 pages + 4000h for PTE

Now to get index we are simply going to do, assuming eax has virtual address:

```
mov     edx, eax
sub     edx, [ebp+memstart] ; -500000h
shr     edx, 0ch            ; index into edx
```

Voila, now by simple accessing [esi+edx*4] we can see if this page is allocated, reserved or it is available. Of course, this is my implementation, you may, and probably will in your implementations organize a little bit different PTE, use more bits, etc.

Allocating memory in such condition is meter of finding free PTEs large enough to hold our requested range. Basically we are using 4000h to describe state of 1000000h bytes of buffer. Isn't that sweet? You have to love Intel and their great idea to organize Physical to Virtual memory translation. Of course, you may allocate smaller buffer, and smaller PTE, that's up to you J

Now I will tell you about FIRST PAGE INDEX and why it is important. I will show you later on how to use nonintrusive tracers on such buffers but let me tell you about FPI. FPI is used to see how big buffer is allocated and how big buffer we have to free in call to VirtualFree. FPI has index of first page and is set in every PTE describing certain range. If FPI is 1 in 3 PTEs that means that this



memory buffer starts at INDEX 1 in PTE, and all pages that have FPI 1 are part of same allocated buffer, this helps us later during coding nonintrusive tracer and also during freeing memory because sometimes VirtualFree is called as VirtualFree(page_base, 0, MEM_DECOMMIT) and we will have memory leaks if we don't store somewhere how big buffer is. Maybe you are wondering why I didn't store size in first PTE and marked following PTEs as present only, instead of storing FPI in each one of them. Simple, FPI will let me know in nonintrusive tracer how big is memory buffer on which I have to change protection. If exception occurs on 3rd page you will change protection only on that page, but I want also to change protection on whole range allocated by call to one VirtualAlloc. I think this makes more sense now J

Look at code with comments I think you will understand this:

```
allocatememory      proc
                    arg     virtualbase
                    arg     range
                    arg     flags
                    arg     memprotection

                    local   numofpages:dword
                    local   dummy_var:dword
                    local   virtualaddress:dword

deltaalloc:         call    deltaalloc
                    pop     edi
                    sub     edi, offset deltaalloc

                    mov     esi, [edi+memstart]
                    add     esi, NEW_MEM_RANGE
                    sub     esi, 4000h      ;structs for memory manager(PTEs)

                    mov     eax, range
                    mov     edx, eax
                    shr     eax, 0ch
                    and     edx, 0FFFh

                    test    edx, edx
                    jz      __mm0
                    inc     eax

__mm0:              mov     numofpages, eax

                    cmp     virtualbase, 0
                    jne     __commitpage    ;commit reserved pages???? yep

                    ;find free block big enough and commit pages
                    ;starting from index 1
                    mov     ecx, 1

__cycle_empty:      test    dword ptr[esi+ecx*4], 1 ;committed?
                    jnz     __next_pte
                    test    dword ptr[esi+ecx*4], 2 ;reserved?
                    jz      __check_size

__next_pte:         inc     ecx
                    cmp     ecx, 1000h
                    jne     __cycle_empty
```



```
__check_size:    mov     eax, numofpages
                 add     eax, ecx

__cycle_size:    dec     eax
                 test    dword ptr[esi+eax*4], 1
                 jnz     __next_pte
                 test    dword ptr[esi+eax*4], 2
                 jnz     __next_pte

                 cmp     eax, ecx
                 jne     __cycle_size

                 ;at this point we have found PTEs large enough to
                 ;describe needed memory buffer
                 mov     eax, numofpages    ;ecx is index used to get page
                 add     eax, ecx

                 mov     edx, ecx
                 shl     edx, 2             ;FPI
                 mov     ebx, flags        ;1 for P or 2 for R

__add_pages:     dec     eax
                 mov     dword ptr[esi+eax*4], 0 ;set PTE to 0
                 or      dword ptr[esi+eax*4], edx ;set FPI
                 or      dword ptr[esi+eax*4], ebx ;set flags

                 cmp     eax, ecx
                 jne     __add_pages

__done:          shl     ecx, 0ch
                 add     ecx, [edi+memstart]
                 mov     virtualaddress, ecx
                 jmp     __exitalloc

__commitpage:    push    virtualbase
                 pop     virtualaddress

                 mov     eax, virtualbase
                 sub     eax, [edi+memstart]
                 shr     eax, 0ch
                 mov     ecx, numofpages
                 mov     edx, eax
                 shl     edx, 2

__commit_em:     mov     dword ptr[esi+eax*4], 0 ;clear pte
                 or      dword ptr[esi+eax*4], 3 ;flags
                 or      dword ptr[esi+eax*4], edx ;fpi

                 inc     eax
                 loop    __commit_em

__exitalloc:     mov     eax, virtualaddress
                 leave
                 retn    10h
                 endp
```



Note how I start from PTE with index 1:

```
; starting from index 1
mov     ecx, 1
```

This is very important, empty PTE is set to 0 by my deallocate memory, it can happen that later allocated and released range will come after the one with FPI = 0, in such case, searching for memory range with FPI of 0 will result in returning bigger range. Of course, we may check R and P bits in PTE but those are 4 extra lines in source code and make code less understandable.

Now if you carefully read this source you will see how it is easy to write nice memory manager from ring3 using PTEs, VirtualFree is even simpler to write:

deallocate memory:

```
mov     esi, [ebp+memstart]
add     esi, NEW_MEM_RANGE
sub     esi, 4000h           ;pointer to PTE

;freeing using index and FPI to find all pages
mov     edx, eax
sub     edx, [ebp+memstart]
shr     edx, 0ch           ;index into eax
mov     ecx, edx           ;FPI into ecx
mov     eax, edx

cmp     eax, 1000h
jnb     __exit_free

__freemem:
mov     edx, [esi+eax*4]
shr     edx, 2
cmp     edx, ecx           ;FPI...
jne     __exit_free

mov     dword ptr[esi+eax*4], 0 ;clear PTE
inc     eax
jmp     __freemem

__exit_free:
retn
```

Very simple, that is all I had to tell about memory manager.

3.3. Problems with Delphi code

Delphi is very important to understand in this case, ASProtect SKE uses virtual.dll written in Delphi; well this is a big problem. I faced with this problem after playing a little bit with ASProtect and other Delphi apps trying to figure why in the name of God my code fails even if I simulate everything in right way. After a couple of hours of tracing I was able to identify problems and here are those:



- @System@@FreeMem
- @System@SysFreeMem

To make this engine work, both procedures must return 0 on success, or proc fails and Delphi app will exit. Even if I simulate everything, it will fail somehow so only solution is to patch these 2 procedures in aspr virtual.dll or any other Delphi proggy:

```
-----  
mov eax, 0  
retn  
-----
```

The next problem is how to locate those two procedures? Using signatures is the only thing that crosses my mind. second call to VirtualAlloc will allocate buffer for aspr virtual.dll so store that address and insert non intrusive debugger that will gain control once we hit int3h - push/ret to aspr virtual.dll and we know when is good time to start scanning for our signatures.

Easier solution would be to dump file at entry of virtual.dll and to scan in IDA for addresses of this two procs. That would be very nice and easy locating w/o single chance to fail J

Anyway here are two procedures from IDA and dumped virtual.dll (note that these are present in all Delphi apps):

```
.dumped:00496564 ; __fastcall System::__linkproc__ FreeMem(void)  
.dumped:00496564 @System@@FreeMem$qqrqv proc near  
.dumped:00496564  
.dumped:00496564         test     eax, eax  
.dumped:00496566         jz      short locret_496572  
.dumped:00496568         call   ds:off_4CE01C  
.dumped:0049656E         or     eax, eax  
.dumped:00496570         jnz     short loc_496573  
.dumped:00496572  
.dumped:00496572 locret_496572:  
.dumped:00496572         retn  
.dumped:00496573  
.dumped:00496573 loc_496573:  
.dumped:00496573         mov     al, 2  
.dumped:00496575         jmp     sub_4965CC  
.dumped:00496575 @System@@FreeMem$qqrqv endp
```

And also:

```
.dumped:00497114 ; __fastcall System::SysFreeMem(void *)  
.dumped:00497114 @System@SysFreeMem$qqrpv proc  
.dumped:00497114  
.dumped:00497114 var_4             = dword ptr -4  
.dumped:00497114  
.dumped:00497114         push    ebp  
.dumped:00497115         mov     ebp, esp  
.dumped:00497117         push    ecx  
.dumped:00497118         push    ebx  
.dumped:00497119         push    esi  
.dumped:0049711A         push    edi  
.dumped:0049711B         mov     ebx, eax  
.dumped:0049711D         xor     eax, eax  
.dumped:0049711F         mov     ds:dword_4D042C, eax
```



```
.dumped:00497124      cmp     ds:byte_4D0428, 0
.dumped:0049712B      jnz     short loc_49714C
.dumped:0049712D      call    @System@_16436 ; System::_16436
.dumped:00497132      test    al, al
.dumped:00497134      jnz     short loc_49714C
.dumped:00497136      mov     ds:dword_4D042C, 8
.dumped:00497140      mov     [ebp+var_4], 8
.dumped:00497147      jmp     loc_4972AD
.dumped:0049714C
```

3.4. Memory manager conclusion

If you have understood all of this then applications that are trying to make themselves anti-dump will fail. You will have everything in one dump able range. Also to be able to see whole prog range in LordPE make sure to increase imagesize in PEB:

```
mov     eax, dword ptr fs:[30h]
mov     eax, [eax+0ch]
mov     eax, [eax+14h]
add     dword ptr[eax+18h], NEW_MEM_RANGE
```

That's all regarding memory manager in ring3. I hope you got idea, and that you are able now to write your own memory manager for your targets. Good luck.



4. Nonintrusive tracers for Memory Manager

Once we have managed to force our code to store everything in one dump able range we are ready to log accesses to those buffers. Some protectors will store polymorphic oep in allocated buffer so we have to be able to log eip access to such buffers. There are plenty of options but we are going to use PAGE_GUARD here, as I did in oepfinder X.Y.Z introduced in [6].



4.1. Writing nonintrusive tracer

Nonintrusive tracers, well, there is a lot to tell about them, really a lot[5,6]. But I'll try to tell you a few important things; soon, after reading this part you will understand how it is important to understand them and how useful they are.

I covered nonintrusive tracers and loaders in [5,6] so I will mention them very briefly in this article. Nonintrusive tracer concept consists of hooking KiUserExceptionDispatcher and processing all exceptions by our code. If we handle exception, we simply call NtContinue or if we don't handle exception we are returning to KiUserExceptionDispatcher. Let's have a look at sample of nonintrusive tracer template:

```
nonintrusive:      mov     ecx, [esp+4]
                   mov     ebx, [esp]

                   pushad
                   call    deltakiuser
deltakiuser:       pop     ebp
                   sub     ebp, offset deltakiuser

...

retkiuser0:        popad
                   mov     [ecx.context_dr0], 0
retkiuser:         push    0deadc0deh
                   ret
```

Don't be confused with `mov [ecx.context_dr0], 0`; you will understand it when we get to chapter 4.3.

For now let's take a look at KiUserExceptionDispatcher:

```
.text:7C90EAE0      mov     ecx, [esp+arg_0]
.text:7C90EAF0      mov     ebx, [esp+0]
.text:7C90EAF3      push    ecx
.text:7C90EAF4      push    ebx
.text:7C90EAF5      call    _RtlDispatchException@8
.text:7C90EAF6      or      al, al
.text:7C90EAF7      jz      short loc_7C90EB0A
.text:7C90EAF8      pop     ebx
.text:7C90EAF9      pop     ecx
.text:7C90EAB0      push    0
.text:7C90EAB1      push    ecx
.text:7C90EAB2      call    _ZwContinue@8
.text:7C90EAB3      jmp     short loc_7C90EB15
.text:7C90EAB4      loc_7C90EB0A:
.text:7C90EAB5      pop     ebx
.text:7C90EAB6      pop     ecx
.text:7C90EAB7      push    0
.text:7C90EAB8      push    ecx
.text:7C90EAB9      push    ebx
.text:7C90EABA      call    _ZwRaiseException@12
```



As you may see I'm talking same things as in [6] but just follow me a little bit longer and you will see why. We are going to hook 1st 2 instructions of KiUserExceptionDispatcher, also note that if we want to get stealth code we may hook `_RtlDispatchException` and insert our tracer there. Possibilities are endless. When we hook `KiUserExceptionDispatcher` we have to simulate overwritten bytes. Those are:

```
.text:7C90EAEBC          mov     ecx, [esp+4]
.text:7C90EAF0          mov     ebx, [esp]
```

ecx = pointer to CONTEXT

ebx = pointer to EXCEPTION_CODE

We can simply decide by examine value of ebx if we are going to process this exception or we are going to pass execution back to `KiUserExceptionDispatcher`:

```
nonintrusive:          mov     ecx, [esp+4]
                        mov     ebx, [esp]

                        pushad
                        call     deltakiuser
deltakiuser:           pop     ebp
                        sub     ebp, offset deltakiuser

                        cmp     dword ptr[ebx], EXCEPTION_BREAKPOINT
                        je       __bp_conditions
                        cmp     dword ptr[ebx], EXCEPTION_GUARD_PAGE
                        jne      retkiuser0
...
retkiuser0:            popad
                        mov     [ecx.context_dr0], 0
retkiuser:             push    0deadc0deh <--- changed in hook engine
                        ret     to point to right
```

Since `KiUserExceptionDispatcher` is exported in `ntdll.dll` we may simple locate it using `GetProcAddress` and get address where we want to hook. Note also that I'm using delta offset all the time because all of this code is executed inside of my injected code:

```
mov     eax, [ebp+KiUserExceptionDispatcher]
lea     ebx, [ebp+nonintrusive]
mov     byte ptr[eax], 0e9h
mov     ecx, eax
add     ecx, 5
sub     ebx, ecx
mov     dword ptr[eax+1], ebx

add     eax, 7
mov     dword ptr[ebp+retkiuser+1], eax
```

With this hook you maybe sure that each exception will go trough your hook and you can process it. Of course, nonintrusive tracers have its down fails, because those can't be adapted to 2 processes when we have debugger/debuggy scenario since exceptions should be handled by debugger process. On other hand, if all exceptions are passed with `DBG_EXCEPTION_NOT_HANDLED` our nonintrusive tracer will work without a problem. Just idea, but why not hook `WaitForDebugEvent`



and pass all exceptions with `DBG_EXCEPTION_NOT_HANDLED`. Juts idea, think about itJ , and read about it in 5.3 chapter J

Second and the biggest down fail of nonintrusive tracers are garbage instructions that deal with stack. If we carefully examine processing of Exception we might see also that `CONTEXT` and Exception code are copied from `ring0` to user stack. If some polymorphic instruction obfuscated stack copying will not be performed and progy will be terminated. This is special case when we are single stepping traced code and we have to determine if next instruction is about to modify stack so we can prevent single stepping or emulate such instruction inside of our tracer. For this I would recommend z0mbie's XDE engine [7].

Oki, I hope you got basic idea.

4.2. Using *PAGE_GUARD* with nonintrusive tracer

`PAGE_GUARD` is used as one-shot access alarm. But also note that `PAGE_GUARD` is NOT removed once exception occurs. Debugger (our nonintrusive tracer) should change protection on a page prior to returning execution back to place from where exception occurred. We may also use `PAGE_NOACCESS`, we will, change protection on faulting page one way or another. But advantage of using `PAGE_GUARD` comes from a fact that we also know error code that is passed to our tracer, that's the only reason why we are using `PAGE_GUARD`. Also there is also possibility for protector to access memory at `0x00000000h` which usually comes from such code:

```
xor     eax, eax
push    offset sehhandle
push    dword ptr fs:[eax]
mov     dword ptr fs:[eax], esp
mov     [eax], eax      <-- Exception
```

And we have to manually determine if exception is caused by dummy usage of registers or it is caused by access to our page with `page_noaccess` set J In 4.3 I will also cover how to distinguish dummy usage of regs from read/write to our ranges.

To set `PAGE_GUARD` we will go back to memory manager and set it on a specified range using this simple formula:

```
__setpageguard:    or     memprotection, PAGE_GUARD
                  lea     eax, dummy_var

                  push    eax
                  push    memprotection
                  push    range
                  push    virtualaddress
                  call     [edi+VirtualProtect]
```

Simple isn't it? This should be stored at very end of our memory manager. It is very simple, and you probably now understand why all allocated buffers are stored on a page boundary? Because no meter you do `VirtualProtect` will set access starting from:

Address to set access = address and `0FFFFFF00h`



Accessing faulting page is more then simple using FPI, tnx to FPI concept we can easily find VirtualAddress and size of any given range:

```
getfpi:                push    esi
                       mov     esi, [ebp+memstart]
                       add     esi, NEW_MEM_RANGE-4000h
                       sub     eax, [ebp+memstart]

                       shr     eax, 0ch           ;get index for this page

                       mov     eax, [esi+eax*4]
                       shr     eax, 2           ;get FPI
                       pop     esi
                       retn

                       ;eax FPI, returns base of page
getvafromfpi:          shl     eax, 0ch
                       add     eax, [ebp+memstart]
                       retn

                       ;eax FPI; returned from getfpi, return size
getsizefromfpi:        push    esi ecx edx ebx

                       mov     esi, [ebp+memstart]
                       add     esi, NEW_MEM_RANGE-4000h

                       mov     ecx, eax
                       xor     ebx, ebx

__cycle_fpi:           mov     edx, [esi+ecx*4]
                       shr     edx, 2
                       cmp     edx, eax         ;compare FPIs
                       jne     __gotsizefromfpi
                       inc     ebx
                       inc     ecx
                       jmp     __cycle_fpi

__gotsizefromfpi:      shl     ebx, 0ch
                       mov     eax, ebx
                       pop     ebx edx ecx esi
                       retn
```

Now it is up to you to implement different memory manager if you don't like mine. It is completely up to you, the main idea of article is you to get idea about this technique.

If you are thinking as I'm right now you are wondering how to know if EXCEPTION_GUARD_PAGE occurred due to execution of guarded page or exception occurred from read/write from/to guarded page? Without use of ring0 this is nearly impossible. I say nearly because to be sure if exception occurred due to execute or read/write you have to keep track in your memory manager information about guard page.



4.3. *PAGE_GUARD in weird conditions aka KiUserExceptionDispatcher improved*

As I was coding my little project, I figured that data passed back to me from KiUserExceptionDispatcher wasn't enough to fulfil my needs. My scenario was probably one of the worst scenarios that some programmer can run into. I'm trying to force ASProtect SKE 2.2 with Advanced OEP protection to store all allocated buffers into one huge buffer which may be dumped later on as one file. To lookup for start of polymorphic oep, obfuscated trough ASPR code I planned to use PAGE_GUARD on every new allocated buffer so I can determine and log when EIP hits allocated buffer, but here comes big problem.

Since all buffers are allocated in one huge memory buffer and my code is acting like memory manager (freeing/allocating pages) I'm not sure when eip is in a guessed range because there is sooooo many of them located at various places in my buffer. One thing that I really need is something that is not passed as information to KiUserExceptionDispatcher. Yep, all I need is content of cr2 register to get faulting address, so if EIP matches content of cr2 then we are in our guarded page.

Solution for this trick from ring3 would be to keep information for each page and once PAGE_GUARD exception occurs we may determine if page_guard occurred with EIP in page w/o PAGE_GUARD or PAGE_GUARD exception occurred with EIP in PAGE_GUARD marked page. This is very good solution for this trick to work, but requires a little bit more coding, and reorganization of my ring3 memory manager to include also protection for every page. Because I'm lazy to recode it, I wrote little addition for KiUserExceptionDispatcher.

There is also possibility for us to hook KiTrap0E and to pass back in context struct content of cr2 register. Hmmmm this would be nice. cr2 register will hold faulting virtual address, and since access to guarded page is nothing more then exception we may also get value of cr2 and be sure if exception occurred due to execution or due to read/write. Of course, KiUserExceptionDispatcher will not return address of faulting address which is very bad for us so I had to improve it to return value of cr2 register:

```
cr2          = faulting_address0
eip          = faulting_address0
exceptioncode = EXCEPTION_GUARD_PAGE
```

Bingo, log it, and remove protection on this page and wait for next access. But how to get content of cr2 register???? Yep, this is really pain in you know where but it is doable. Lets have a look at KiTrap0E and it's logic:

```
.text:804DAF25 _KiTrap0E:
.text:804DAF25      mov     word ptr [esp+2], 0
.text:804DAF2C      push    ebp
.text:804DAF2D      push    ebx
.text:804DAF2E      push    esi

...

.text:804DB0ED      mov     ecx, 3
.text:804DB0F2      mov     edi, eax
.text:804DB0F4      mov     eax, 0C0000006h
.text:804DB0F9      call   CommonDispatchException
```



So far all KiTraps will call CommonDispatchException which will save CONTEXT and ERROR_CODE on stack and redirect EIP to KiUserExceptionDispatcher. Following CommonDispatchException takes us here:

```
.text:804D8A8D CommonDispatchException proc near
.text:804D8A8D
.text:804D8A8D
.text:804D8A8D
.text:804D8A8D      sub     esp, 50h
.text:804D8A90      mov     [esp+50h+var_50], eax
.text:804D8A93      xor     eax, eax
.text:804D8A95      mov     [esp+50h+var_4C], eax
.text:804D8A99      mov     [esp+50h+var_48], eax
.text:804D8A9D      mov     [esp+50h+var_44], ebx
.text:804D8AA1      mov     [esp+50h+var_40], ecx
.text:804D8AA5      cmp     ecx, 0
.text:804D8AA8      jz      short loc_804D8AB6
.text:804D8AAA      lea     ebx, [esp+50h+var_3C]
.text:804D8AAE      mov     [ebx], edx
.text:804D8AB0      mov     [ebx+4], esi
.text:804D8AB3      mov     [ebx+8], edi
.text:804D8AB6
.text:804D8AB6 loc_804D8AB6:
.text:804D8AB6      mov     ecx, esp
.text:804D8AB8      test    dword ptr [ebp+70h], 20000h
.text:804D8ABF      jz      short loc_804D8AC8
.text:804D8AC1      mov     eax, 0FFFFh
.text:804D8AC6      jmp     short loc_804D8ACB
.text:804D8AC8
```

```
-----
.text:804D8AC8
.text:804D8AC8 loc_804D8AC8:
.text:804D8AC8      mov     eax, [ebp+6Ch]
.text:804D8ACB
.text:804D8ACB loc_804D8ACB:
.text:804D8ACB      and     eax, 1
.text:804D8ACE      push    1
.text:804D8AD0      push    eax
.text:804D8AD1      push    ebp
.text:804D8AD2      push    0
.text:804D8AD4      push    ecx
.text:804D8AD5      call    _KiDispatchException@20
.text:804D8ADA      mov     esp, ebp
.text:804D8ADC      jmp     Kei386EoiHelper@0
.text:804D8ADC CommonDispatchException endp
```

Ok! it will end up sooner or later in _KiDispatchException so we follow it:

```
.text:804F318D ; __stdcall KiDispatchException(x,x,x,x,x)
.text:804F318D _KiDispatchException@20 proc near
.text:804F318D
.text:804F318D      push    390h
.text:804F3192      push    offset dword_804F3278
.text:804F3197      call    __SEH_prolog
.text:804F319C      mov     eax, ds:___security_cookie
.text:804F31A1      mov     [ebp-1Ch], eax
.text:804F31A4      mov     esi, [ebp+8]
.text:804F31A7      mov     [ebp-2ECh], esi
```



```
.text:804F31AD      mov     ecx, [ebp+0Ch]
.text:804F31B0      mov     [ebp-2F0h], ecx
.text:804F31B6      mov     ebx, [ebp+10h]
.text:804F31B9      mov     [ebp-2F8h], ebx
.text:804F31BF      db      3Eh
.text:804F31BF      mov     eax, ds:0FFDFF020h
.text:804F31C5      inc     dword ptr [eax+504h]
.text:804F31CB      mov     dword ptr [ebp-2E8h], 10017h
.text:804F31D5      cmp     byte ptr [ebp+14h], 1
.text:804F31D9      jz      loc_804F5A76
.text:804F31DF      cmp     ds:_KdDebuggerEnabled, 0
.text:804F31E6      jnz     loc_804F5A76
.text:804F31EC      loc_804F31EC:
.text:804F31EC      lea     eax, [ebp-2E8h]
.text:804F31EC      push    eax
.text:804F31F2      push    ecx
.text:804F31F3      push    ebx
.text:804F31F4      call    _KeContextFromKframes@12
.text:804F31F5      mov     eax, [esi]
.text:804F31FA      cmp     eax, 80000003h
.text:804F31FC      jnz     loc_804F5A20
.text:804F3201      dec     dword ptr [ebp-230h]
.text:804F320D      loc_804F320D:
.text:804F320D      xor     edi, edi
.text:804F320F      loc_804F320F:
.text:804F320F      cmp     byte ptr [ebp+14h], 0
.text:804F3213      jnz     loc_804F58C3
.text:804F3219      cmp     byte ptr [ebp+18h], 1
.text:804F321D      jnz     loc_80516D98
.text:804F3223      mov     eax, ds:_KiDebugRoutine
.text:804F3228      cmp     eax, edi
.text:804F322A      jz      loc_80505721
.text:804F3230      push    edi
.text:804F3231      push    edi
.text:804F3232      lea     ecx, [ebp-2E8h]
.text:804F3238      push    ecx
.text:804F3239      push    esi
.text:804F323A      push    dword ptr [ebp-2F0h]
.text:804F3240      push    ebx
.text:804F3241      call    eax
.text:804F3243      test    al, al
.text:804F3245      jz      loc_80505721
.text:804F324B      loc_804F324B:
.text:804F324B      push    dword ptr [ebp+14h]
.text:804F324E      push    dword ptr [ebp-2E8h]
.text:804F3254      lea     eax, [ebp-2E8h]
.text:804F325A      push    eax
.text:804F325B      push    dword ptr [ebp-2F0h]
.text:804F3261      push    ebx
.text:804F3262      call    _KeContextToKframes@20
.text:804F3267      loc_804F3267:
.text:804F3267      mov     ecx, [ebp-1Ch]
.text:804F326A      call    @xHalReferenceHandler@4
```



```
.text:804F326F      call    __SEH_epilog
.text:804F3274      retn    14h
.text:804F3274  _KiDispatchException@20 endp ; sp = -14h
.text:804F3274
```

Since this code doesn't make too much sense without live debugging we may start Tracing with Softice and we will find our magic bytes very very soon:

```
.text:804F5959      call    _ProbeForWrite@12
.text:804F595E      mov     ecx, 0B3h
.text:804F5963      lea     esi, [ebp+var_2E8]
.text:804F5969      rep movsd
```

Instruction at [804F5969](#) is responsible for copying CONTEXT to user stack, for KiUserExceptionDispatcher of course, if we are able to hook that instruction we may also store content of cr2 register and pass its value back to ring3 in some field of CONTEXT struct. Such fields are dr0/4 regs. Our nonintrusive tracer should clear cr2 value if we don't process it.

We are going to hook "`lea esi, [ebp+var_2E8]`" because its size is big enough to store hook - push/ret combo, note that `rep movsd` must not be hooked because there is possibility for BSOD during driver unloading. It might happen, as described already by Mark Russinovich, that some thread is interrupted during execution of code (hook) in driver's address space, if we unload driver at that point, when interrupted thread is scheduled to run it will return to no existing memory and you will get PAGE_FAULT and BSOD. I haven't encountered it yet, but it is possible. Just have it in mind if you get BSOD and still you have no idea what is causing it J

Also we must be able to identify our process, to accomplish this I will use cr3 trick since cr3 has physical address of PDE, and every process has it's own memory then I'm able to identify process w/o problem [11]. Note also that I'm using KeStackAttachProcess which is not needed; we can get cr3 value directly from KPROCESS struct (part of EPROCESS):

```
kd> dt nt!_KPROCESS
+0x000 Header          : _DISPATCHER_HEADER
+0x010 ProfileListHead : _LIST_ENTRY
+0x018 DirectoryTableBase : [2] Uint4B    <-- here it is J
+0x020 LdtDescriptor   : _KGDTENTRY
```

Offset is 25963h from base of ntoskrnl.exe and we are ready to pool this trick. Let's do it!!!! You must lookup this instruction in your version of ntoskrnl.exe because offsets might be a little different depending on ntoskrnl.exe.

There is also one MORE condition which is very very very important and that is saving of cr2!!!! cr2 is not saved, if other page fault occurs then cr2 will hold that address. This is dangerous for us, because page_fault handler is also used to bring to memory pages stored in pagefile. Only way to save cr2 value is to hook int0eh and get cr2 only if, and only if, it is our process. Later on, our hook will peek up content of saved cr2 and pass it to our code in context.context_dr0.



```
hookint0eh          label    dword
                   push     eax
                   mov      eax, cr3
                   cmp      eax, c_cr3
                   jne      __exit_int0e

                   mov      eax, cr2
                   mov      c_cr2, eax

__exit_int0e:       pop      eax
                   jmp      cs:[oldint0eh]

hookmycode          ;saves cr2 content in context.context_dr0
                   label    dword
                   push     eax
                   mov      eax, cr3
                   cmp      eax, c_cr3
                   jne      __exithook

                   lea      esi, [ebp-2e8h]
                   mov      eax, c_cr2
                   mov      [esi.context_dr0], eax

__exithook:         pop      eax
                   lea      esi, [ebp-2e8h]
retaddr:            push     0deac0deh
                   ret
```

Now in your nonintrusive tracer you will only test if EIP = DR0 and if so we log access if not, we remove PAGE_GUARD and set int 3h after faulting instruction. And we have improved KiUserExceptionDispatcher to act as we want J

Also this will help us to distinguish ACCESS_VIOLATION caused by access to our pages if we are using PAGE_NOACCESS instead of PAGE_GUARD.

4.4. Logging Access

Once we know how to approach problem, we have to log access to our pages, this can be done using wiring to file or much better solution would be to use OutputDebugStringA, and DbgView from www.sysinternals.com to get output.

Sample code:

```
__log:              lea      ecx, [ebp+format5]
                   lea      ebx, [ebp+buffer]

                   push     eax
                   push     ecx
                   push     ebx
                   call     [ebp+wsprintfA]
                   add      esp, 0ch

                   push     ebx
                   call     [ebp+OutputDebugStringA] ;log accesss

...
format5             db      "eip log : 0x%.08X", 13, 10, 0
```



Now, you should get a lot of data in DbgView, save log and study addresses, later, exclude those which are repeating and reduce size of log file by running tracer again.

4.5. Invoking driver from tracer

Well actually this is used to signal driver what PID to take care of, driver is installed at this point, but we have to signal it somehow to start tracing us:

```
push    0
push    0
push    OPEN_EXISTING
push    0
push    0
push    GENERIC_READ or GENERIC_WRITE
push    offset driver
callW   CreateFileW
mov     dhandle, eax

call    DeviceIoControl, eax, 20h, o pid, 4, o pid, 4, o dwbytes, 0

push    dhandle
callW   CloseHandle

...
driver:                unis    <\\.\ring0>
pid = pid of process we are tracing
```

I'm also using same method to identify process as in article Loader from ring0 [11].

4.6. Making stealth nonintrusive tracers

Oki, here we are dealing with hypothetical new protector that will scan for our hooks in KiUserExceptionDispatcher and we want to defeat such future protector. How to do that? Not an easy task but we are going to do this tnx to already shown trick by yates [8].

KiUserExceptionDispatcher is procedure that will never, never return, it will call NtContinue or NtRaiseException. Let's see what is happening:

- exception occurs
- ntoskrnl.exe receives control via KiTrapXX
- KiTraps are actually entries in IDT, and depending on exception we have two possible scenarios of stack layout at the entry of KiTrapXX:

+-----+	+-----+
EFLAGS	EFLAGS
+-----+	+-----+
CS	CS
+-----+	+-----+
EIP	EIP
+-----+	+-----+
	Error Code
	+-----+



Since some Exceptions don't generate Error and to make exiting from ring0 easier no meter what exception occurred some of KiTrapXX have push 0 to simulate Error code such are for example KiTrap01 and KiTrap03:

```
_KiTrap01
0008:804D8D7C  PUSH      00          <--- dummy Error Code
0008:804D8D7E  MOV       WORD PTR [ESP+02],0000
0008:804D8D85  PUSH      EBP
0008:804D8D86  PUSH      EBX
0008:804D8D87  PUSH      ESI
0008:804D8D88  PUSH      EDI
0008:804D8D89  PUSH      FS
```

```
_KiTrap03
0008:804D915B  PUSH      00          <--- dummy Error Code
0008:804D915D  MOV       WORD PTR [ESP+02],0000
0008:804D9164  PUSH      EBP
0008:804D9165  PUSH      EBX
0008:804D9166  PUSH      ESI
0008:804D9167  PUSH      EDI
0008:804D9168  PUSH      FS
```

But KiTrap0E (page fault handler) doesn't have push 0 because Error Code is stored on stack.

```
_KiTrap0E
0008:804DAF25  MOV       WORD PTR [ESP+02],0000
0008:804DAF2C  PUSH      EBP
0008:804DAF2D  PUSH      EBX
0008:804DAF2E  PUSH      ESI
0008:804DAF2F  PUSH      EDI
0008:804DAF30  PUSH      FS
0008:804DAF32  MOV       EBX,00000030
```

Returning from interrupt is performed using simple IRETD instruction which is similar to ret and will also jmp to EIP saved on stack. After exception is processed and ring0 decides to call KiUserExceptionDispatcher it will store address of KiUserExceptionDispatcher on stack so IRETD will simple return to KiUserExceptionDispatcher:

```
0008:804F5A0F  MOV       EAX,[_KeUserExceptionDispatcher]
0008:804F5A14  MOV       [EBX+68],EAX
```

```
:dd ebx+68
0010:EEC21DCC 7C90EAE C 0000001B 00000246 0013FCD0      îê |....F.....
0010:EEC21DDC 00000023 00000000 00000000 00000000      #.....
```

As you can see there is overwritten EIP with address of KiUserExceptionDispatcher, saved CS, saved Eflags, saved esp and saved SS on stack. Now what we are going to do is to hook those instructions so it will point to some other code in ntdll.dll which we have stored using method presented by yates [8]. Also better solution would be to avoid scanning of ntdll.dll stored on disk with one loaded in memory to redirect everything to UserSharedData which is mapped in user mode as ReadOnly at:



```
kd> ? SharedUserData
Evaluate expression: 2147352576 = 7ffe0000
kd>
```

But from ring0 it is mapped at:

```
#define KI_USER_SHARED_DATA          0xffdf0000
```

So we can write there from ring0 and redirect our Exceptions to that code which will be responsible for jmping to KiUserExceptionDispatcher or it will simply call our nonintrusive tracer stored somewhere in memory of traced prog. [ring0stealthtracer folder] Make sure that you understand this code before you run it. I'm using PID this time to identify process.

First we locate base of ntoskrnl.exe

```
                iMOV    esi, ZwCreateFile
                and     esi, 0FFFFFF00h

__find_base:    cmp     word ptr[esi], 'ZM'
                je      __ntoskrnlbase
                sub     esi, 1000h
                jmp     __find_base

__ntoskrnlbase: mov     ntoskrnlbase, esi
```

Then we have to locate _KeUserExceptionDispatcherVariable which is not exported. Luckily for us this one is stored on dword boundary and there is only one occurrence of KiUserExceptionDispatcher in ntoskrnl.exe so we can search for it using address of KiUserExceptionDispatcher:

```
                mov     edi, esi
                mov     ebx, esi
                add     ebx, dword ptr[ebx+3ch]
                mov     ecx, [ebx.NT_OptionalHeader.OH_SizeOfImage]
                shr     ecx, 2
                cld
                mov     eax, kiuser
                repnz   scasd
                sub     edi, 4
```

Once we have located not exported _KeUserExceptionDispatcher we may store our code into UserSharedData, and overwrite _KeUserExceptionDispatcher with address of our code:

```
                push    edi

                mov     KiUserExceptionDispatcher, kiuser
                mov     edi, kiusershareddata+100h
                mov     esi, offset shareddatahook
                mov     ecx, shareddatahooksize
                cld
                rep     movsb

                pop     edi
                mov     dword ptr[edi], kiusershareddataring3+100h
...
kiusershareddata    equ     0ffdf0000h
kiusershareddataring3 equ    07ffe0000h
```



Bingo!

Lets see how it looks if it is traced with softICE?

```
_KiTrap03
0008:804D915B  PUSH      00
0008:804D915D  MOV       WORD PTR [ESP+02],0000
0008:804D9164  PUSH      EBP
0008:804D9165  PUSH      EBX
0008:804D9166  PUSH      ESI
0008:804D9167  PUSH      EDI
0008:804D9168  PUSH      FS
...
0008:804F5A0F  MOV       EAX,[_KeUserExceptionDispatcher]
0008:804F5A14  MOV       [EBX+68],EAX
0008:804F5A17  OR        DWORD PTR [EBP-04],-01
...
:dd _KeUserExceptionDispatcher
0023:80552AF0  7FFE0100  7C90EAD0  7C90EAC0  00002626  ... .ê |.ê |&&..
```

Here is stack layout prior to iretd:

```
0023:F0E9DDCC  7FFE0100  0000001B  00000246  0013FCC8  ... ....F.....
0023:F0E9DDDC  00000023  00000000  00000000  00000000  #.....
```

Then we enter into our code in UserSharedData:

```
001B:7FFE0100  CALL      7FFE0105
001B:7FFE0105  POP       EDX
001B:7FFE0106  SUB       EDX,F7129409
001B:7FFE010C  MOV       ECX,FS:[0020]          <-- get PID from TEB
001B:7FFE0112  CMP       DWORD PTR [EDX+F712943B],-01 <-- not tracing
001B:7FFE0119  JZ        ntdll!KiUserExceptionDispatcher jmp to KiUser
001B:7FFE011B  CMP       [EDX+F712943B],ECX     <-- traced progy PID?
001B:7FFE0121  JNZ       ntdll!KiUserExceptionDispatcher nope we call KiUser
001B:7FFE0123  JMP       [EDX+F7129437]         <-- we jmp to nonintrusive
001B:7FFE0129  JMP       [EDX+ntdll!KiUserExceptionDispatcher] <-- go back to KiUser
```

Not traced progy, so we continue with our quest in KiUserExceptionDispatcher:

```
ntdll!KiUserExceptionDispatcher
001B:7C90EAE0  MOV       ECX,[ESP+04]
001B:7C90EAF0  MOV       EBX,[ESP]
001B:7C90EAF3  PUSH      ECX
001B:7C90EAF4  PUSH      EBX
```

I hope you understood this stealth tech? Good luck with such future protector!!

Note that this trick was used by Barnaby Jack in his famous article [9] about ring0 shell code.



4.7. Nonintrusive tracer conclusion

I hope you got basic idea, all codes and samples I have provided are planed to be used with hook in ntoskrnl.exe, take care, you will probably need to change offsets in my driver src J All codes for drivers are provided as is, due to wrong offsets their usage without previous offset changing may and probably will result in BSOD, system crash or data loss or system corruption. You are using compiled driver binaries at your OWN RISK!!!



5. Loader for Loader

Loader for Loader is actually a name for loaders that will deal with 2 process protections. I have made one loader for some game that used loader to load actual game. Since I had 2 loaders I named this technique “Loader for Loader”, there is probably somewhere on inet my small txt describing this tech, but I think I should cover it right here, especially because we have some 2 processes protectors.

We have to gain control over 2nd processes, to do that we are going to hook CreateProcessA and wait it to finish so we may get PROCESS_INFORMATION struct with ProcessID and ThreadID.

- ProcessId is required so we can open process and write to its memory allocate buffers, etc...
- ThreadId is required because we have to control something, Get/SetThreadContext, Resume/SuspendThread.

Note: we don't need pid and tid if we are hooking from injected offset independent code because we already have ProcessHandle and ThreadHandle stored in PROCESS_INFORMATION struct.

I will cover three methods here and one which is not for sure described yet in RCE world:

5.1. Loader for Loader with injected code

Idea here is to inject our memory manager in second process, whole concept consist of hooking CreateProcessA at ret at which point we will have error code (if process is executed w/o a problem) and also we will have filled PROCESS_INFORMATION struct with handled to process and primary thread. Basically you will have here 2 offset independent codes. Example is included in [lfinjected folder].

Let's take a look at our example, I will use this test app for second Loader for Loader method so just bare in mind what we are dealing with here:

```
push    offset mutexname
push    0
push    MUTEX_ALL_ACCESS
callW   OpenMutexA
test    eax, eax
jnz     __2ndprocess

push    offset mutexname
push    0
push    0
callW   CreateMutexA

push    offset pinfo
push    offset sinfo
push    0
push    0
push    0
push    0
push    0
```



```
    push    0
    push    offset progy
    push    0
    callW   CreateProcessA

    push    -1
    push    pinfo.pi_hProcess
    callW   WaitForSingleObject
    jmp     __exit

__2ndprocess:
    push    40h
    push    offset mtitle
    push    offset mtext
    push    0
    callW   MessageBoxA

__exit:
    push    0
    callW   ExitProcess
```

As you can see simple application that will create mutex and depending on mutex print message or create new instance of process. To pool loader for loader here, we are going to inject our code into this target in such way that 1st part of injected code will hook CreateProcessA and second part will be injected. Basically you have two offset independent codes but luckily you will have to use getkernelbase and search exports from kernel32.dll only once since kernel32.dll offset will not change from process to process.

Concept:

- CreateProcess in suspended state
- Inject offset independent code
- Let offset independent code hook CreateProcessA in our target process
- Once we hit our hook in CreateProcessA at ret get values of phandle and thandle
- Now our offset independent code should use WriteProcessMemory and VirtualAllocEx to inject new loader in 2nd process and basically we are repeating same steps as in our original loader to inject offset independent code

Due to the size of loader I will not present code in this document, you may lookup for source in [If injected folder] which is commented very well.

5.2. Loader for Loader without injected code

This is much easier solution, and requires much less coding and understanding of offset independent code but it is not very good if process is created w/o CREATE_SUSPENDED because we have to insert CREATE_SUSPENDED flag manually which will also require bigger code.

Concept:

- load 1st app with CREATE_SUSPENDED flag
- insert jmp \$ in CreateProcessA
- when we hit hook read address of PROCESS_INFORMATION, dwCreationFlags and ret address from stack
- or dwCreationFlags with CREATE_SUSPENDED and store it back, remove hook, and insert new at instruction pointed by return address from stack.



- When we hit second hook, read `PROCESS_INFORMATION` and you will get pid and tid
- Use `OpenProcess` and `OpenThread` to play with new process

Check [lflwoinjected folder] for complete implementation with comments

5.3. Nonintrusive tracers for Debugged process

I will use Armadillo 4.3 crackme here with Standard Protection + Debug Blocker to inject my tracer in it, of course, this also applies for memory manager code, but to make work as simple as possible I will inject nonintrusive tracer into it to break at OEP. Lol, guess what you got DebugBlocker armadillo oep finder with this tutorial.

This is not very hard to put in practice, all you need is a little bit imagination, and knowledge how Debugging is performed in ring3 with Windows Debug APIs [10], or you can read some tutorial about debug loaders.

To make this trick work we are going to hook `WaitForDebugEvent` in such way that we are the first ones to get its output and we will be able to examine content of `DEBUG_EVENT` struct. We are only interested in `EXCEPTION_DEBUG_EVENT` which will be passed to our code, and all other events are passed back to prog. We are calling `ContinueDebugEvent` with `DBG_EXCEPTION_NOT_HANDLED` which will call `KiUserExceptionDispatcher` in our traced prog. To make this trick work we also have to hook `CreateProcessA` at `retn` in armadillo to be able to inject our nonintrusive tracer.

Code and target are located in [armadillo_oep folder], at this point I would like to thank to Teddy Rogers for his great unpackme collection at <http://www.tuts4you.com/unpackme/> because we are going to use armadillo unpackme from his site J

Let's see what is going on in armadillo after `CreateProcessW` is called:

```
004949EC . 52          PUSH EDX
004949ED . 6A 02       PUSH 2
004949EF . 68 A4B44C00 PUSH armadill.004CB4A4
004949F4 . 8B45 10     MOV EAX,DWORD PTR SS:[EBP+10]
004949F7 . 50         PUSH EAX
004949F8 . 8B4D 08     MOV ECX,DWORD PTR SS:[EBP+8]
004949FB . 8B11       MOV EDX,DWORD PTR DS:[ECX]
004949FD . 52         PUSH EDX
004949FE . FF15 D0304C00 CALL DWORD PTR DS:[<&KERNEL32.ReadProcessMem>
```

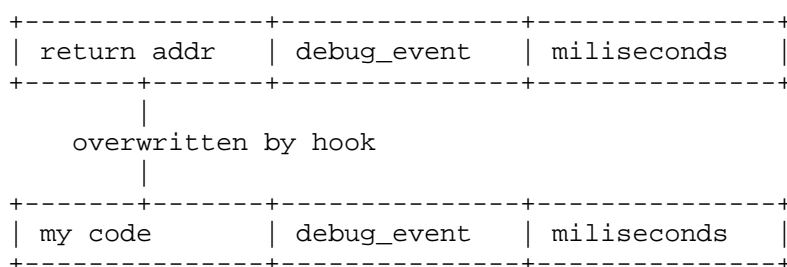
At this point Armadillo is reading 2 bytes from entry point so it can insert `jmp $` hook at entry point and attach later to process with `DebugActiveProcess`, of course, armadillo is using `Sleep/GetThreadContext` to locate when prog is at entry point and in infinite loop:

```
0048FD8D > \A1 A0B54C00 MOV EAX,DWORD PTR DS:[4CB5A0]
0048FD92 . 8B48 08     MOV ECX,DWORD PTR DS:[EAX+8]
0048FD95 . 51         PUSH ECX
0048FD96 . FF15 C4314C00 CALL DWORD PTR DS:[<&KERNEL32.DebugActivePro>
```



From now on, we are entering into Debug Loop with WaitForDebugEvent and calls to ContinueDebugEvent. “Hybrid” hooking of WaitForDebugEvent will not work here because ret is not padded with nops so we are going to perform a little bit smarter hooking approach.

Smart hooking approach is one where we are going to overwrite ret address stored on stack and to execute WaitForDebugEvent, once it tries to go back, well it will go back to our code J Oki, still with me? Maybe some graph will make more sense J



Now once WaitForDebugEvent tries to return to its caller it will end up in my code, and we are now processing exceptions as we want:

Sample hooking code:

```
;
;first we take a few bytes needed to insert our hook
;
mov     esi, [edi+WaitForDebugEvent]
xor     ecx, ecx
push    esi

__get5bytes:
push    esi
call    ldex86
add     ecx, eax
add     esi, eax
cmp     ecx, 5
jb      __get5bytes

pop     esi

push    edi
lea     edi, [edi+rippedbytes]
cld
rep     movsb
pop     edi

mov     dword ptr[edi+goback+1], esi
;
; copy bytes to buffer where they will be executed safely
;
mov     esi, [edi+WaitForDebugEvent]
mov     ecx, esi
lea     ebx, [edi+hookwaitfordebugetevent]
mov     byte ptr[esi], 0e9h
add     ecx, 5
sub     ebx, ecx
mov     dword ptr[esi+1], ebx
```



Overwritten bytes are stored in rippedbytes variable:

```
hookwaitfordebugevent    label    dword
                          pusha
                          call     deltahook
deltahook:               pop      ebp
                          sub      ebp, offset deltahook
                          mov      eax, [esp+8*4]
                          ;save old return address
                          mov      [ebp+waitfordebugeventretaddress], eax
                          mov      dword ptr[ebp+retorig+1], eax
                          lea      ebp, [ebp+mywaitfordebugevent]
                          mov      [esp+8*4], ebp
                          popa
rippedbytes               db       30      dup(90h)
goback:                  push     0deac0deh <- WaitForDebugEvent + ripped bytes
                          ret
```

Once we are done with call to WaitForDebugEvent it will end up here:

```
mywaitfordebugevent      proc
                          mov      ecx, [esp-8]      ;here is debugevent
                          pusha
```

Note also that offsets are negative because ret and leave have aligned stack since windows APIs are using stdcall convention (except wsprintfA, DbgPrint, some other???).

When we access variables in such case we have to calculate them in negative offset, also take care not to push anything to stack at this moment because each push will corrupt data stored on stack which are essential for us in such hooking approaches.

Now we are taking care of exceptions and are only processing EXCEPTION_DEBUG_EVENT:

```
mywaitfordebugevent      proc
                          mov      ecx, [esp-8]      ;here is debugevent
                          pusha
                          call     deltamydebug
deltamydebug:           pop      edi
                          sub      edi, offset deltamydebug

                          mov      ebx, ecx

                          cmp      [ebx.de_code], EXCEPTION_DEBUG_EVENT
                          jne      __return_to_original

                          cmp      [ebx.de_u.ER_ExceptionCode], EXCEPTION_BREAKPOINT
                          jne      __passexception
                          cmp      [edi+firstint3h], 1
                          je       __passexception
                          ;set page guard in remote process

                          pushv    <dd      ?>
                          push     PAGE_EXECUTE_READWRITE or PAGE_GUARD
                          push     [edi+c_range]
                          push     [edi+c_start]
```



```
        push    [edi+phandle]
        call    [edi+VirtualProtectEx]

        push    DBG_CONTINUE
        push    [ebx.de_ThreadId]
        push    [ebx.de_ProcessId]
        call    [edi+ContinueDebugEvent]
        mov     [edi+firstint3h], 1
        jmp     __l33t

__passexception:    push    DBG_EXCEPTION_NOT_HANDLED
                   push    [ebx.de_ThreadId]
                   push    [ebx.de_ProcessId]
                   call    [edi+ContinueDebugEvent]

;
; now question is where should we return!?!? we may store dummy
; error code in debug_event.code and processed on knowing that
; nothing will foobar our code =)
;
__l33t:            mov     [ebx.de_code], 0deadc0deh        ;l33t

__return_to_original: popa
retorig:          push    0deadc0deh ;changed with ret address
                   ret
                   endp
```

We are only taking care of 1st int 3h which comes from DebugBreak, that one should be passed with ContinueDebugEvent(DBG_CONTINUE), in other cases we are passing exceptions to debuggy using ContinueDebugEvent(DBG_EXCEPTION_NOT_HANDLED) letting our nonintrusive tracer to play with memory of traced process, also to avoid armadillo foobaring our work we set debug_event.code to 0deadc0deh so it will not process debug_event J , or we might set ThreadId or ProcessId with some garbage values so ContinueDebugEvent will do nothing to our traced process.

Now everything is on our nonintrusive tracer stored in debuggy, and wait till it tells you that oep is found. Also in nonintrusive tracer I have stored jmp \$ once you hit “Ok” in MessageBoxA, so all you have to do is dump process and, of course, terminate it from Task Manager or Process Explorer by Mark Russinovich.

Good luck J

Code is located in [armadillo_oep folder]



6. Debugging injected code - tips

Essential in debugging this kind of code is ability to view code of second process, and especially when we are dealing with loader for loader concept, or tracing of debugged app.

There are a few tricks I would like to pin point:

- use `int 3h` in your code and `bpint 3` in SoftICE or `i3here on` to break in suspicious parts of code
- use `ADDR` to see what is actually going on in 2nd process in loader for loader scenario
- when writing nonintrusive tracers use `jmp $` and `ctrl+d` to break with softice, since `bpint 3` or `i3here on` will occur too many times due to logic of nonintrusive tracers, or if you are using `drX` registers to step over faulting instruction then `int 3h` in code will work.

Those are all tips regarding debugging such loaders, Good luck J Also, same tricks apply for debugging injected DLL.



7. Conclusion

I showed some tricks I have had in mind and which I coded. I hope this will be helpful to some people, and especially to newbie reversers to understand how it is important programming knowledge in RCE world. Some will disagree or agree with me, some will like my way, some will spit on it, some will rip it and say that I failed and recode it, but anyway, I gave you a lot of things to think about, and remember perfections comes with practice. If you don't succeed at first, try again, answer is laying somewhere around. It took me 2 days to figure memory manager vs. Delphi code, but it took me only 4 compiling of armadillo_oep to make working code J

Codes for each thing I've shown here are supplied with this document, for better understanding I recommend reading source code also. Tasm32 DDK is also included.

S verom u Boga, deroko/ARTeam

All the code provided with this tutorial is free for public use, just make a greetz to the authors and the ARTeam if you find it useful to use. Don't use these concepts for making illegal operation, all the info here reported are only meant for studying and to help having a better knowledge of application code security techniques.



8. References

- [1] Optimization of 32 bit code, Benny/29a, <http://vx.netlux.org/29a/29a-4/29a-4.215>
- [2] Gaining important datas from PEB under NT boxes, Ratter/29a, <http://vx.netlux.org/29a/29a-6/29a-6.224>
- [3] Billy Belcebu Virus Writing Guide 1.0 for Win32, Billy Belcebu, <http://vx.netlux.org/29a/29a-4/29a-4.202>
- [4] Retrieving API's addresses. LethalMind, <http://vx.netlux.org/29a/29a-4/29a-4.227>
- [5] Solution to The Amazing Picture downloader, deroko, http://www.crackmes.de/users/warrantyvoider/the_amazing_picture_downloader/
- [6] Unpacking and Dumping ExeCryptor and coding loader for it, deroko, <http://tutorials.accessroot.com>
- [7] eXtended (XDE) disassembler engine, z0mbie, <http://vx.netlux.org/29a/magazines/29a-8.rar>
- [8] Anti-Anti-Bpm, yates, <http://www.yates2k.net/syscode/bpm.rar>
- [9] Remote Windows Kernel Exploitation - Step into ring0, Barnaby Jack, <http://www.eeye.com/~data/publish/whitepapers/research/OT20050205.FILE.pdf>
- [10] Win32 Debug API Part 1/2/3, Iczelion, <http://win32asm.cjb.net/>
- [11] Loader from ring0, deroko, ARTeam eZine #1, <http://ezine.accessroot.com>

Some useful tutorials to learn about loaders theory:

- [12] Shub-Niggurath and ThunderPwr coding loader series, <http://tutorials.accessroot.com>
- [13] Createing Loaders & Dumpers – Crackers Guide to Program Flow Control, yates, <http://www.yates2k.net/lad.txt>
- [14] Using Memory Breakpoints, Shub-Niggurath, <http://tutorials.accessroot.com>

Some articles that might be interesting to read:

- [15] DLL Injection Tutorial, Darawk, <http://www.edgeofnowhere.cc/viewtopic.php?p=2441382>
- [16] Three Ways to Inject Your Code into Another Process, Robert Kuster, The Code Project <http://www.codeproject.com/threads/winspy.asp>
- [17] InjLib – A Library that implements remote code injection for all Windows versions, Antonio Feijao, The Code Project, <http://www.codeproject.com/library/InjLib.asp>

Useful tools:

- [18] OllyAdvanced plug-in, MaRKuS TH-DJM, <http://www.tuts4you.com/forum/index.php?showtopic=7092>
<http://omega.intechhosting.com/~access/forums/index.php?showtopic=2542>
- [19] LordPE plug-in, deroko, <http://deroko.phearless.org/dumpdll/>
- [20] Ice-Ext, Stan, <http://stenri.pisem.net/>



9. Greetings

I wish to tank all the ARTeam members for sharing their knowledge, to 29a virus writing group for one of the best e-zines, to my friends from phearless e-zine, to all my friends from Reversing Labs forum, and to all great coders out there... and of course you for reading this article.



<http://cracking.accessroot.com>

© [ARTeam] 2006