

Dealing with funny checksum

Version 1.0
February 2013

1. Forewords

After a while, I've decided to write about something interesting which I've found while unpacking one protection, and it will be also nice introduction to one of my tools which I have wrote for fun of it.

However, I won't mention application name here, but to demonstrate checksum check which I have found I will be using one test application, thus you will get idea what happened, and how checksum is defeated

I will also introduce one tool I wrote, which served me well in this particular case. Tool should come with this document, thus I won't describe tool, and it's internals as source code should be well commented

deroko of ARTeam

Disclaimers

All code included with this tutorial is free to use and modify; we only ask that you mention where you found it. This tutorial is also free to distribute in its current unaltered form, with all the included supplements.

All the commercial programs used within this document have been used only for the purpose of demonstrating the theories and methods described. No distribution of patched applications has been done under any media or host. The applications used were most of the times already been patched, and cracked versions were available since a lot of time. ARTeam or the authors of the paper cannot be considered responsible damages the companies holding rights on those programs. The scope of this tutorial as well as any other ARTeam tutorial is of sharing knowledge and teaching how to patch applications, how to bypass protections and generally speaking how to improve the RCE art. We are not releasing any cracked application.

Verification

ARTeam.esfv can be opened in the ARTeamESFVChecker to verify all files have been released by ARTeam and are unaltered. The ARTeamESFVChecker can be obtained in the release section of the ARTeam site: <http://releases.accessroot.com>

Table of Contents

1.	Forewords	1
	Verification	2
1.	Checksum Check	3
1.1.	Test Application description	3
1.2.	Instrumentation comes to the rescue	5
2.	Instrumentation Tool	8
2.1.	Fast Basic Block Lookup	8
2.2.	Self-modifying code handling	8
2.3.	Handling sysenter	9
2.4.	Preserving hooks and entry points	10
2.5.	Child Process trace	10
2.6.	DbgPrint	10
2.7.	Windows 8	10
2.8.	Memory Allocation	10
2.9.	Exception injection	10
2.10.	TF handling	11
2.11.	TODO	11
3.	End	12
3.1.	References	12
3.2.	Conclusions	12
3.3.	Greetings	12

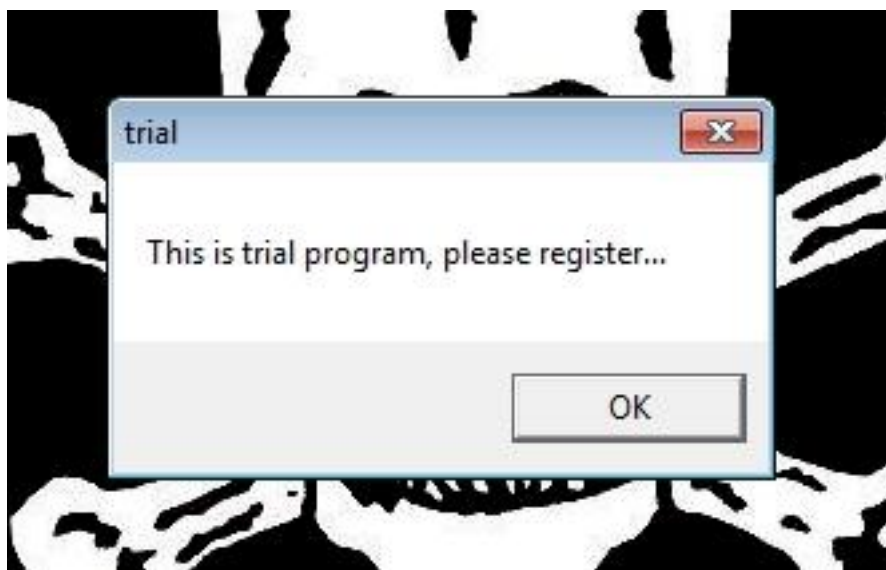
1. Checksum Check

During one of my reversing sessions I stumbled across application which, after unpacking, and a little bit of patching started crashing with ACCESS_VIOLATION at weird location. As always I would use debugger to break in, and see what's going on. This time, there were no clues about crash. What I could see is that EIP is pointing to weird location. Usually through stack layout you can find from where code was executed, but not this time.

Setting memory break points or hardware breakpoints was out of the question. We are talking about ~50mb of code. Also static analyze was out of the question. I didn't want to spend more than 1 hour on unpacking this target. It was more like exercise application, and not 1 month project to analyze it in details. Even more complicated was that stack, and state of registers would be different on every occasion, and executed from different threads.

1.1. Test Application description

Test Application which I wrote for this article pretty much mimics behavior of original applications checksum check. Our Test Application will have one window, and every 10 seconds there will be popup letting us know that we are using trial application:



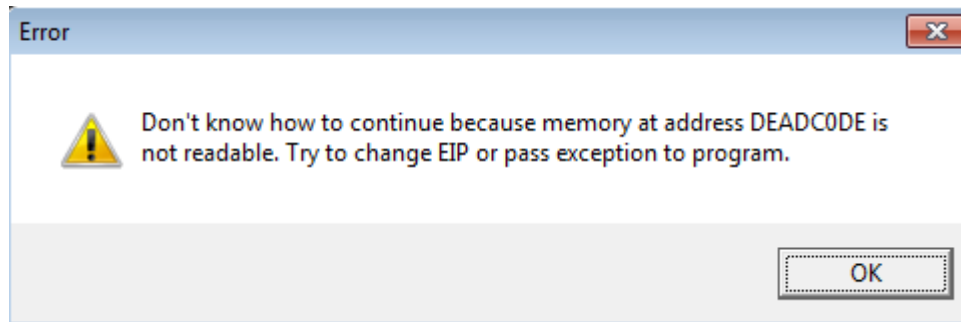
Let look in debugger where this trial message is coming from:

<pre> 0100113E: PUSH 0 01001140: PUSH testapp.010021A4 01001145: PUSH testapp.01002150 0100114A: PUSH DWORD PTR SS:[EBP+8] 0100114D: CALL DWORD PTR DS:[<&USER32.MessageBoxW>] </pre>	<pre> Style = MB_OK MB_APPLMODAL Title = "trial" Text = "This is trial program, p hOwner MessageBoxW </pre>
---	---

Ok, we found it, let's patch it:

<pre> .0100113C: 7313 .0100113E: 6A00 .01001140: 68A4210001 .01001145: 6850210001 .0100114A: FF7508 .0100114D: 83C410 .01001150: 909090 </pre>	<pre> jnz push push 0 push 0010021A4 ;'trial' push 001002150 ;'This is trial d, [ebp+8] add esp, 010 nop </pre>
--	---

Our patch is ready, and if we run application we will get crash. Of course, now is time to load it into debugger and see what's going on.



Now let's have a look at stack:

0006F894	DEADC0DE	↓
0006F898	DEADC0DE	↓
0006F89C	DEADC0DE	↓
0006F8A0	DEADC0DE	↓
0006F8A4	DEADC0DE	↓
0006F8A8	DEADC0DE	↓
0006F8AC	DEADC0DE	↓
0006F8B0	DEADC0DE	↓
0006F8B4	DEADC0DE	↓
0006F8B8	DEADC0DE	↓
0006F8BC	DEADC0DE	↓
0006F8C0	DEADC0DE	↓
0006F8C4	DEADC0DE	↓
0006F8C8	DEADC0DE	↓
0006F8CC	DEADC0DE	↓
0006F8D0	DEADC0DE	↓
0006F8D4	DEADC0DE	↓
0006F8D8	DEADC0DE	↓
0006F8DC	DEADC0DE	↓
0006F8E0	DEADC0DE	↓
0006F8E4	DEADC0DE	↓
0006F8E8	DEADC0DE	↓
0006F8EC	DEADC0DE	↓
0006F8F0	DEADC0DE	↓
0006F8F4	DEADC0DE	↓
0006F8F8	DEADC0DE	↓
0006F8FC	DEADC0DE	↓
0006F900	DEADC0DE	↓
0006F904	DEADC0DE	↓
0006F908	DEADC0DE	↓
0006F90C	DEADC0DE	↓
0006F910	DEADC0DE	↓
0006F914	DEADC0DE	↓
0006F918	DEADC0DE	↓
0006F91C	DEADC0DE	↓
0006F920	DEADC0DE	↓
0006F924	DEADC0DE	↓
0006F928	DEADC0DE	↓
0006F92C	DEADC0DE	↓
0006F930	DEADC0DE	↓

Looks ugly, as whole stack up to top is filled with DEADCODE, thus we don't have any starting point. Remember this checksum check is thrown from different threads, and from different code parts from ~50mb file, and it's thrown at different execution times. Register layout is also messed up:

```

EAX DEADCODE
ECX DEADCODE
EDX DEADCODE
EBX DEADCODE
ESP 0006F8A4
EBP DEADCODE
ESI DEADCODE
EDI DEADCODE
EIP DEADCODE

```

No obvious reason why this happened, and from where exception is triggered. What we can assume is that:

- **jmp/call/ret/iretd** is used to redirect execution to DEADCODE
- **NtContinue** might have been used to set EIP to DEADCODE
- Hook in some api, APC delivery to DEADCODE, thread execution to DEADCODE etc. but all of these belong to **jmp/call/ret/iretd** cases

I've observed in my real target **NtContinue**, and concluded that it's not used for eip redirection. What remains are these 4 instructions. But how do I break there? How do I track them? I'm not going to analyze this file in details. Remember, this was an unpacking exercise for me.

1.2. Instrumentation comes to the rescue

Only way to break at certain instructions is to instrument application. No other way. You have already 2 instrumentation frameworks – **PIN Tool**, and **DynamoRIO**, but this time I won't use any public tools. I will use my own instrumentation library which I wrote in spare time.

After several runs, we can see that application is dying at address **DEADCODE** thus I will instrument my tool to make `jmp $` when **call/jmp/ret** are leading to **DEADCODE**.

```
void    instrumentCallJumpRet(
    __in px86dis px86,
    __in unsigned long dest,
    __in unsigned long src)
{
    if (dest == 0xDEADCODE){
        DbgPrint(("s -- found killing code..", __FUNCTION__));
        DbgPrint(("s -- dst = %.08X src = : %.08X", __FUNCTION__,
dest, src));
        __asm jmp $
    }
}
```

Now let's run my tool, and watch output in **DbgView**:

```
41  5:32:44 PM  [4088] imageAdd -- adding image to the list : \device\harddiskvolu
42  5:32:44 PM  [4088] imageAdd -- adding image to the list : \device\harddiskvolu
43  5:32:44 PM  [4088] imageAdd -- adding image to the list : \device\harddiskvolu
44  5:32:44 PM  [4088] imageAdd -- adding image to the list : \device\harddiskvolu
45  5:32:44 PM  [4088] instrumentCallJumpRet -- found killing code..
46  5:32:44 PM  [4088] instrumentCallJumpRet -- dst = DEADCODE src = : 01001036
```

Now if we attach with debugger to this process, we might see what is last instruction executed, before exception happens:

```
01001021  JE SHORT 0100102C
01001023  MOV ECX,EAX
01001025  MOV EAX,DEADCODE
0100102A  REP STOS DWORD PTR ES:[EDI]
0100102C  RDTSC
0100102E  AND EAX,0FC
01001033  SUB ESP,EAX
01001035  POPAD
01001036  RETN
01001037  POP EDI
01001038  POP ESI
01001039  POP EBX
0100103A  POP EBP
0100103B  RETN
0100103C  INT3
0100103D  INT3
```

Ok, we are good. We see that **DEADCODE** is used to fill "something", now we can load file into IDA, and see what's going on. Remember this is test application, so what I will present is similar how it was done in the real target, but checksum check was checking only certain parts of the code, not whole code section.

IDA output of a whole function:

```
.text:01001005 sub_1001005      proc near
.text:01001005                mov     edi, edi
.text:01001007                push    ebp
.text:01001008                mov     ebp, esp
.text:0100100A                mov     eax, large fs:4
.text:01001010                push    ebx
.text:01001011                push    esi
.text:01001012                push    edi
```

```

.text:01001013      lea     edi, [ebp+4]
.text:01001016      sub     edi, 100h
.text:0100101C      sub     eax, edi
.text:0100101E      shr     eax, 2
.text:01001021      jz      short loc_100102C
.text:01001023      mov     ecx, eax
.text:01001025      mov     eax, 0DEADC0DEh
.text:0100102A      rep stosd
.text:0100102C      loc_100102C:
.text:0100102C      rdtsc
.text:0100102E      and     eax, 0FCh
.text:01001033      sub     esp, eax
.text:01001035      popa
.text:01001036      retn

```

As we can see, code reads **TEB.StackBase**, and gets address of return address to calculate stack size needed for wipe. It also subtracts 100h from stack, thus even some previous stack frame is wiped (which we could have used to locate some previously called procedure which return address is still on the stack). Before ret is hit, esp is moved somewhere in this stack randomly thus we can't pinpoint at least stack offset at which wipe happened.

Lets follow reference to this function:

```

.text:01001041      sub_1001041      proc
.text:01001041
.text:01001041
.text:01001041      pbData          = byte ptr -1Ch
.text:01001041      pdwDataLen      = dword ptr -0Ch
.text:01001041      hProv           = dword ptr -8
.text:01001041      hHash           = dword ptr -4
.text:01001041
.text:01001041      mov     edi, edi
.text:01001043      push    ebp
.text:01001044      mov     ebp, esp
.text:01001046      sub     esp, 1Ch
.text:01001049      mov     eax, large fs:30h
.text:0100104F      push    ebx
.text:01001050      push    esi
.text:01001051      mov     esi, [eax+8]
.text:01001054      mov     eax, [esi+3Ch]
.text:01001057      push    edi
.text:01001058      add     eax, esi
.text:0100105A      movzx   ecx, word ptr [eax+14h]
.text:0100105E      push    0F0000040h      ; dwFlags
.text:01001063      push    1                ; dwProvType
.text:01001065      xor     ebx, ebx
.text:01001067      push    ebx              ; szProvider
.text:01001068      lea     edi, [ecx+eax+18h]
.text:0100106C      push    ebx              ; szContainer
.text:0100106D      lea     eax, [ebp+hProv]
.text:01001070      push    eax              ; phProv
.text:01001071      call    ds:CryptAcquireContextW
.text:01001077      lea     eax, [ebp+hHash]
.text:0100107A      push    eax              ; phHash
.text:0100107B      push    ebx              ; dwFlags
.text:0100107C      push    ebx              ; hKey
.text:0100107D      push    8003h            ; Algid
.text:01001082      push    [ebp+hProv]      ; hProv
.text:01001085      call    ds:CryptCreateHash
.text:0100108B      mov     eax, [edi+0Ch]

```

```

.text:0100108E      push     ebx                ; dwFlags
.text:0100108F      push     dword ptr [edi+10h] ; dwDataLen
.text:01001092      add      eax, esi
.text:01001094      push     eax                ; pbData
.text:01001095      push     [ebp+hHash]        ; hHash
.text:01001098      call     ds:CryptHashData
.text:0100109E      push     ebx                ; dwFlags
.text:0100109F      lea      eax, [ebp+pdwDataLen]
.text:010010A2      push     eax                ; pdwDataLen
.text:010010A3      lea      eax, [ebp+pbData]
.text:010010A6      push     eax                ; pbData
.text:010010A7      push     2                  ; dwParam
.text:010010A9      push     [ebp+hHash]        ; hHash
.text:010010AC      mov      [ebp+pdwDataLen], 10h
.text:010010B3      call     ds:CryptGetHashParam
.text:010010B9      push     [ebp+hHash]        ; hHash
.text:010010BC      call     ds:CryptDestroyHash
.text:010010C2      push     ebx                ; dwFlags
.text:010010C3      push     [ebp+hProv]        ; hProv
.text:010010C6      call     ds:CryptReleaseContext
.text:010010CC      push     4
.text:010010CE      pop      ecx
.text:010010CF      add      esi, 40h
.text:010010D2      lea      edi, [ebp+pbData]
.text:010010D5      xor      eax, eax
.text:010010D7      repe     cmpsd
.text:010010D9      pop      edi
.text:010010DA      pop      esi
.text:010010DB      pop      ebx
.text:010010DC      jz       short locret_10010E3
.text:010010DE      call     DestroyStack
.text:010010E3
.text:010010E3 locret_10010E3:
.text:010010E3      leave
.text:010010E4      retn
.text:010010E4 sub_1001041     endp

```

Code is not that hard to understand. What it does, is to compute md5 sum of code section, and compare it against md5 sum which is stored at **imagebase+0x40** in case that these 2 don't match, function **DestroyStack** is called. Now we can simply patch out this **DestroyStack** function, and everything will be working. We have defeated checksum check which was crashing this program.

2. Instrumentation Tool

In this chapter I will outline some design features of this tool for which I think are nice. However, there are many things to be done with the tool, and updates will be always available, either on **ARTeam** website, and/or my web site. You may find links in **References** section.

Why was this tool developed? One reason was that I always like to have full control over code I'm using, and to be able to fast fix bugs.

First of all this tool operates on Basic Blocks. Basic Block is everything which can be executed without eip redirection instructions.

```
        mov     esi, eax
        mov     edi, edx
        mov     ecx, 200h
__loop: mov     eax, [esi]
        mov     [edi], eax
        add     esi, 4
        add     edi, 4
        dec     ecx
        jnz     __loop
```

In this case, everything until jnz is considered Basic Block. Theoretically speaking Basic Block in terms of disassembling should be from __loop until jnz, but in case of instrumentation that's not the case, as everything here can be executed inside of single Basic Block. Now that we have idea what Basic Block is, I will describe some things which gave me a small headache during development

2.1. Fast Basic Block Lookup

One of the most important things is fast Basic Block lookup. There are several possible ways of doing it. One which was original idea was to use lists. Lists are nice, and provide nice interface for basic block lookups, but considering that every time when we need new basic block, and that's always, we have to cycle all lists. Obviously lists are not good choice. My design uses portion of EIP as **index** to area where I keep all data about memory mappings, and every memory mapping has array of **4 * 0x1000** bytes, thus basic block lookup happens fast using this pseudo formula:

```
pvmmap = get_vmmap(eip);
pbbl = pvmmap->bbl_array[eip & 0xFFF];
```

This is very fast, and although it seems that this code consumes a lots of memory, which is true, during my testing it seems like the best solution. Of course, there can be always added certain **Garbage Collector** which will wipe out basic blocks, and pages which are not executed that often.

2.2. Self-modifying code handling

Self modifying code is very tricky to handle. To handle all possible cases I keep always track of all mappings for every page inside of a given process. Every page can have flags which describe it's state : **VMMAP_READ**, **VMMAP_EXEC**, **VMMAP_WRITE**, **VMMAP_WAS_WRITE**. First three flags are clear, but **VMMAP_WAS_WRITE** has certain meaning which will happen in this case:

- Memory is executed, thus basic block is built
- Memory is given write protection - prot |= VMMAP_WRITE
- Memory is written
- Memory protection is restored – prot, clear VMMAP_WRITE, set VMMAP_WAS_WRITE
- Memory is executed

In this case, with `VMMAP_WAS_WRITE` we know that we have to check if basic block has changed. For this I use special field inside of basic blocks which keep original bytes for this basic block. If change happens, we will catch it, rebuild basic block, and then clear `VMMAP_WAS_WRITE` flag. The best case for this flag you will see in **WriteProcessMemory** usage:

```
call someptr          <---- build BBL
WriteProcessMemory(GetCurrentProcess(), <someptr>, <mydata>, 5, 0);
-> NtProtectVirtualMemory(someptr); <-- PAGE_EXECUTE_READWRITE
-> NtWriteVirtualMemory(someptr);   <-- make change
-> NtProtectVirtualMemory(someptr); <-- PAGE_EXECUTE_READ
call someptr          <---- free and rebuild BBL due to WAS_WRITE
```

For basic blocks which are executed in `VMMAP_WRITE` area, all instructions which are doing write, are considered as an end of basic block. Why is this so, next example will give more insight:

```
call    __delta
__delta: pop    ebp          <----+
        sub    ebp, offset __delta      |
        mov    eax, offset __write      |
        mov    byte ptr[ebp+eax], 0c3h  +--- this would be one bblock
__write: nop                    |      which would lose control
        mov    ecx, edx              <----+ when executed live
```

In this case, everything until write instruction is considered as basic block, thus we never lose control in case instruction is doing memory write inside of basic block.

2.3. Handling sysenter

sysenter is very tough to instrument. What we usually expect is that instruction continues execution after itself. With **sysenter** that's not the case, as **sysexit** in kernel will return to **ntdll!KiFastSystemCallRet** which is **ret** instruction. There are several ways to cheat here, and I will outline some of them:

- Every **sysenter** redirect to **int 2e**
- Before **sysenter**, change return address on stack thus **ret** from **KiFastSystemCallRet** will return to instrumentation library.
- Hook **ntdll!KiFastSystemCallRet** to always enter instrumentation library, thus control is never lost. This approach causes some problems

Int 2e approach is nice, and easy to implement, but I didn't like it, as it causes possible detection to happen. Eg. If **sysenter** is present on system due to **cpuid**, and registers are not set as expected we can assume that we are instrumented.

Change ESP ret address – very nice approach. I've used it at the beginning, and I'll outline how it was done:

```
mov     [esp], offset __retsysenter
sysenter
__retsysenter:
```

In this case **ret** in **ntdll!KiFastSystemCall** will return after **sysenter**. Which is very nice, and requires to keep list of return addresses according to stack index, thus we know where we should continue. Nice, but also easily detected. Call **NtReadVirtualMemory** with "current ESP – 0x8" for example, and check if **ret** address goes to **ntdll** or somewhere else.

Hook ntdll!KiFastSystemCallRet – This is far the best solution. Hooking **ntdll!KiFastSystemCallRet**, only problem is how to distinguish between calls which are coming from my tool, and the ones which are coming from instrumented code. I will show my unique solution in next section, as it's part of **preserving hooks in ntdll.dll**.

2.4. Preserving hooks and entry points

Application which we are instrumenting can remove some of our entry hooks. One of them is also **ntdll!KiFastSystemCallRet**, or **ntdll!KiUserExceptionDispatcher** for example. What we can do, is to monitor write instructions which are modifying this code, imagine performance impact which is huge as it is, not counting this extra code, or we can do something even better!! We will **remap ntdll.dll** to another base for instrumented process, thus our hooks always remain in place, and if application wants to hook it's own **KiUserExceptionDispatcher**, it can do so, our original **KiUserExceptionDispatcher** will remain intact. This also covers case of **ntdll!KiFastSystemCallRet**, as our hook will no be removed.

What I do, not to cause conflict between my calls to **Nt*** apis, which will end up with call to **sysenter**, is to walk all of **Nt*** APIs and hook them to point to **KiIntSystemCall** which is **int 2e**, thus no conflict between my code, and **sysenter**, and **KiFastSystemCallRet**

2.5. Child Process trace

It's very important to keep control over child process, thus all applications can be instrumented. To do this, I have special case for instrumenting **NtCreateProcess/NtCreateProcessEx/NtCreateUserProcess**. Common for these functions is that handle of a new process is always stored in first argument. What happens then is mapping of **ntdll.dll** in new process, as we might expect that application will write some changes to **ntdll.dll** (for sandbox maybe) while process is still suspended, and it expects it to be at certain base, at this point, new thread is injected into process which will perform hooking, and initialization of instrumentation library, and control will be returned to father process.

2.6. DbgPrint

I usually like using **OutputDebugStringA/W** to output everything to **DbgView** thus I don't need to write proper logging code, but in this case every **OutputDebugStringA/W** will result in exception (that's how these functions works). To overcome this extra work, I wrote my own implementation of **OutputDebugStringA/W** so all logging can be seen inside of **DbgView** without triggering any exception from my code.

2.7. Windows 8

Windows 8 has some mitigations techniques to mitigate ROP, and one of them is to check if stack is inside of **TEBs StackBase/StackLimit**, thus every time I enter into instrumentation code, I replace **StackBase/StackLimit** with **StackBase/StackLimit** of my stack for instrumentation

2.8. Memory Allocation

Memory allocation is very tricky. I could write my own, and waste time, si I've decided to use already available, and proven memory allocation code known as **Doug Lee's malloc** . No need to describe it here.

2.9. Exception injection

Sometimes, instruction which I'm emulating can cause exception. Thus in this case I inject exception through function called **tracInjectException** . This function is used in case of single step exception injection, and access violation during emulation of **ret/call/jmp**.

Function reconstructs EXCEPTION_POINTERS with EXCEPTION_RECORD and CONTEXT, and executes shadow KiUserExceptionDispatcher from remapped ntdll.dll

2.10. TF handling

One of the most important features which I wanted to be present in this code is TF handling. That was a must have for me, and it has to be supported within the tool. Current implementation is not bullet proof but it can handle single step encryption/decryption. Usually, TF is easy to handle, whenever it's set we need to execute one instruction, and simulate exception via `TraceInjectException`. All of this works great, and that's how it had to be done. But tricky part comes when instruction which we are single stepping causes exception. TF will become part of CONTEXT structure, and single step exception will happen in `ntdll!KiUserExceptionDispatcher`:

```
xor     eax, eax
pushfd
or      dword ptr[esp], 100h
popfd
call    [eax]    <--- STATUS_ACCESS_VIOLATION <-- TF = 1
```

In this case, we need to fill `CONTEXT.EFlags` with TF, and also to inject single step after execution of 1st instruction inside of `ntdll!KiUserExceptionDispatcher`. This all sounds way too easy, but in fact, it's more complicated in real life than I've expected. This was certain case I had to handle, and more you can find inside of source code of this tool.

2.11. TODO

There are many things left to add and fix, properly handle some cases of which I can think of like drX processing. In this case only drX on execution should be handled, as others will be triggered by code it's self when it reads/writes to memory. But as tool is stable atm, and can be used to instrument many applications, I've decided to release it, it's easy to customize it if required.

3. End

3.1. References

Pin – A Dynamic Binary Instrumentation Tool

<http://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool>

DynamoRIO

<http://www.dynamorio.org/>

ARTeam

<https://accessroot.com>

cr4zyserb – deroko of ARTeam

<http://deroko.phearless.org>

Doug Lee's malloc

<http://g.oswego.edu/dl/html/malloc.html>

3.2. Conclusions

Well in this small write up, and introduction of my tool, I wanted to present different ways in fighting protections, and also need to know when/how/what tools to use. I always preferred to use custom written tools to do the job for me, instead of relying on any public tool. No reason for that, I'm used to it, and I hope you will too. I don't expect anybody to use this tool, but if only one person learned something from this tool, and article, my mission is accomplished.

3.3. Greetings

I would like to say thank you to all my mates from **ARTeam**, although we haven't been active that much in last few years, to **ex members of 29a** group for sharing their knowledge, friendly people at **unpack.cn**, **woodman.com**, and **forum.exetools.com**.



С вером у Бога, СЛОБОДА ИЛИ СМРТ

deroko of ARTeam