

# Bypass EDR's memory protection, introduction to hooking

[medium.com/@fsx30/bypass-edrs-memory-protection-introduction-to-hooking-2efb21acffd6](https://medium.com/@fsx30/bypass-edrs-memory-protection-introduction-to-hooking-2efb21acffd6)

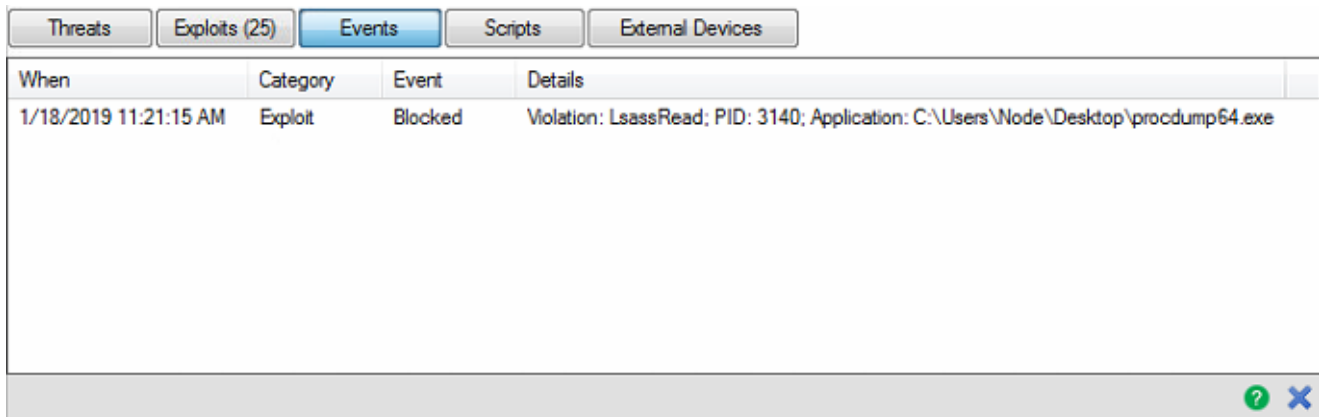
February 3, 2020



Hoang Bui

## Introduction

On a recent internal penetration engagement, I was faced against an EDR product that I will not name. This product greatly hindered my ability to access lsass' memory and use our own custom flavor of Mimikatz to dump clear-text credentials.



The screenshot shows a security tool interface with tabs for Threats, Exploits (25), Events, Scripts, and External Devices. The Events tab is selected, displaying a table with the following data:

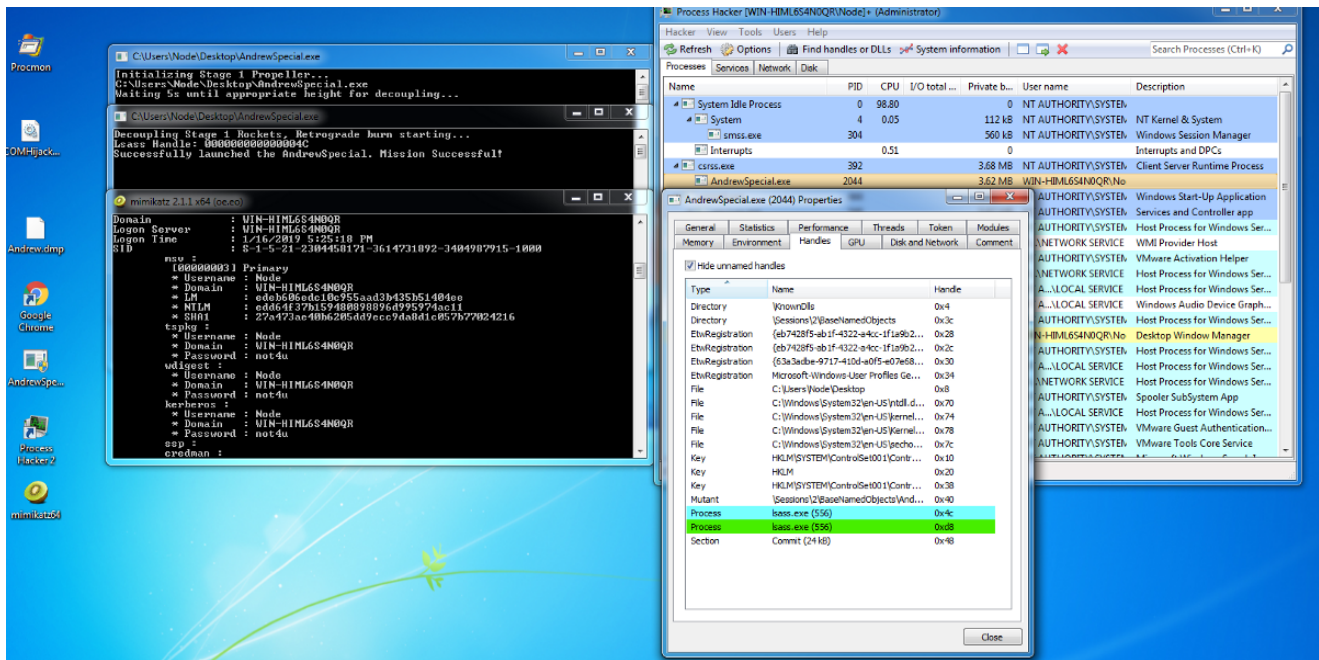
When	Category	Event	Details
1/18/2019 11:21:15 AM	Exploit	Blocked	Violation: LsassRead; PID: 3140; Application: C:\Users\Node\Desktop\procdump64.exe

For those who recommends ProcDump

## The Wrong Path

So now, as an ex-malware author — I know that there are a few things you could do as a driver to accomplish this detection and block. The first thing that comes to my mind was Obregistercallback which is commonly used by many Antivirus products. Microsoft implemented this callback due to many antivirus products performing very sketchy winapi hooks that reassemble malware rootkits. However, at the bottom of the msdn page, you will notice a text saying “*Available starting with Windows Vista with Service Pack 1 (SP1) and Windows Server 2008.*” To give some missing context, I am on a Windows server 2003 at the moment. Therefore, it is missing the necessary function to perform this block.

After spending hours and hours, doing black magic stuff with csrss.exe and attempting to inherit a handle to lsass.exe through csrss.exe, I was successful in gaining a handle with PROCESS\_ALL\_ACCESS to lsass.exe. This was through abusing csrss to spawn a child process and then inherit the already existing handle to lsass.



There is no EDR solution on this machine, this was just an PoC

However, after thinking “I got this!” and was ready to rejoice in victory over defeating a certain EDR, I was met with a disappointing conclusion. The EDR blocked the shellcode injection into csrss.exe as well as the thread creation through *RtlCreateUserThread*. However, for some reason — the code while failing to spawn as a child process and inherit the handle, was still somehow able to get the `PROCESS_ALL_ACCESS` handle to lsass.exe.

WHAT?!

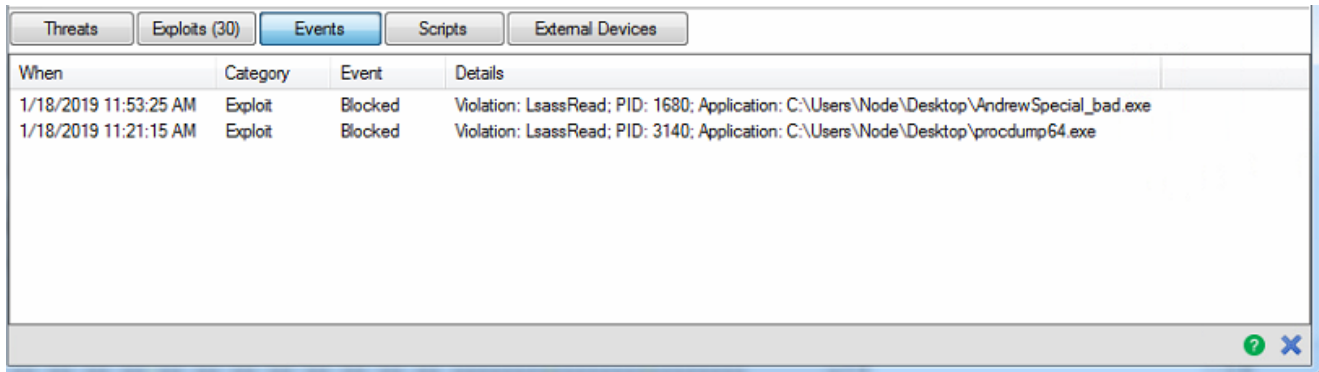
Hold up, let me try just opening a handle to lsass.exe without any fancy stuff with just this line:

```
HANDLE hProc = OpenProcess(PROCESS_ALL_ACCESS, FALSE, lsasspid);
```

And what do you know, I got a handle with `FULL CONTROL` over lsass.exe. The EDR did not make a single fuzz about this. This is when I realized, I started off the approach the wrong way and the EDR never really cared about you gaining the handle access. It is what you do afterward with that handle that will come under scrutiny.

## Back on Track

Knowing there was no fancy trick in getting a full control handle to lsass.exe, we can now move forward to find the next point of the issue. Immediately calling `MiniDumpWriteDump()` with the handle failed spectacularly.



When	Category	Event	Details
1/18/2019 11:53:25 AM	Exploit	Blocked	Violation: LsassRead; PID: 1680; Application: C:\Users\Node\Desktop\AndrewSpecial_bad.exe
1/18/2019 11:21:15 AM	Exploit	Blocked	Violation: LsassRead; PID: 3140; Application: C:\Users\Node\Desktop\procdump64.exe

Let's dissect this warning further. "Violation: LsassRead". I didn't read anything, what are you talking about? I just want to do a dump of the process. However, I also know that to make a dump of a remote process, there must be some sort of WINAPI being called such as ReadProcessMemory (RPM) inside MiniDumpWriteDump(). Let's look at MiniDumpWriteDump's source code at [ReactOS](#).

◆ dump\_exception\_info()

2

```
static unsigned dump_exception_info ( struct dump_context * dc,
                                     const MINIDUMP_EXCEPTION_INFORMATION * except
                                     )
```

Definition at line 391 of file minidump.c.

```
393 {
394     MINIDUMP_EXCEPTION_STREAM mdExcpt;
395     EXCEPTION_RECORD          rec, *prec;
396     CONTEXT                   ctx, *pctx;
397     DWORD                     i;
398
399     mdExcpt.ThreadId = except->ThreadId;
400     mdExcpt.__alignment = 0;
401     if (except->ClientPointers)
402     {
403         EXCEPTION_POINTERS ep;
404
405         ReadProcessMemory(dc->hProcess,
406                          except->ExceptionPointers, &ep, sizeof(ep), NULL);
407         ReadProcessMemory(dc->hProcess,
408                          ep.ExceptionRecord, &rec, sizeof(rec), NULL);
409         ReadProcessMemory(dc->hProcess,
410                          ep.ContextRecord, &ctx, sizeof(ctx), NULL);
411
412         prec = &rec;
413         pctx = &ctx;
414     }
415     else
416     {
417         prec = except->ExceptionPointers->ExceptionRecord;
418         pctx = except->ExceptionPointers->ContextRecord;
419     }
420     mdExcpt.ExceptionRecord.ExceptionCode = prec->ExceptionCode;
421     mdExcpt.ExceptionRecord.ExceptionFlags = prec->ExceptionFlags;
422     mdExcpt.ExceptionRecord.ExceptionRecord = (DWORD_PTR)prec->ExceptionRecord;
423     mdExcpt.ExceptionRecord.ExceptionAddress = (DWORD_PTR)prec->ExceptionAddress;
424     mdExcpt.ExceptionRecord.NumberParameters = prec->NumberParameters;
425     mdExcpt.ExceptionRecord.__unusedAlignment = 0;
426     for (i = 0; i < mdExcpt.ExceptionRecord.NumberParameters; i++)
427         mdExcpt.ExceptionRecord.ExceptionInformation[i] = prec->ExceptionInformation[i];
428     mdExcpt.ThreadContext.DataSize = sizeof(*pctx);
429     mdExcpt.ThreadContext.Rva = dc->rva + sizeof(mdExcpt);
430
431     append(dc, &mdExcpt, sizeof(mdExcpt));
432     append(dc, pctx, sizeof(*pctx));
433     return sizeof(mdExcpt);
434 }
```

3

Referenced by MiniDumpWriteDump().

1

Multiple calls to RPM

As you can see by, the function (2) dump\_exception\_info(), as well as many other functions, relies on (3) RPM to perform its duty. These functions are referenced by MiniDumpWriteDump (1) and this is probably the root of our issue. Now here is where a bit of experience comes into play. You must understand the Windows System Internal and how WINAPIs are processed. Using ReadProcessMemory as an example — it works like this.

ReadProcessMemory is just a wrapper. It does a bunch of sanity check such as nullptr check. That is all RPM does. However, RPM also calls a function “NtReadVirtualMemory”, which sets up the registers before doing a *syscall* instruction. Syscall instruction is just telling the CPU to enter kernel mode which then another function ALSO named NtReadVirtualMemory is called, which does the actual logic of what ReadProcessMemory is supposed to do.

— — — — — -Userland — — — — — | — — — Kernel Land — — —



kernel32.ReadProcessMemory			
Address	Bytes	Opcode	Comment
<b>kernel32.ReadProcessMemory</b>			
kernel32.Reac48 83 EC 38		<b>sub</b> rsp,38	56
kernel32.Reac48 88 44 24 60		<b>mov</b> rax,[rsp+60]	
kernel32.Reac48 89 44 24 20		<b>mov</b> [rsp+20],rax	
kernel32.ReacE8 0D77FEFF		<b>call</b> kernel32.GetUserDefaultUILanguage+5D0	->->KERNELBASE.ReadProcessMemo
kernel32.Reac48 83 C4 38		<b>add</b> rsp,38	56
kernel32.ReacC3		<b>ret</b>	

### ReadProcessMemory

ntdll.NtQueryAttributesFile+B			
Address	Bytes	Opcode	Comment
<b>ntdll.NtClearEvent</b>			
ntdll.NtClearEvent	4C 8B D1	<b>mov</b> r10,rcx	
ntdll.ZwClearEvent+3	B8 3B000000	<b>mov</b> eax,0000003B	59
ntdll.ZwClearEvent+8	0F05	<b>syscall</b>	
ntdll.ZwClearEvent+A	C3	<b>ret</b>	
ntdll.ZwClearEvent+B	0F1F 44 00 00	<b>nop</b> [rax+rax+00]	
<b>ntdll.NtReadVirtualMemory</b>			
ntdll.NtReadVirtualMemory	E9 DFFDF9CF	<b>jmp</b> 47050084	
ntdll.ZwReadVirtualMemory+5	00 00	<b>add</b> [rax],al	
ntdll.ZwReadVirtualMemory+7	00 0F	<b>add</b> [rdi],cl	
ntdll.ZwReadVirtualMemory+9	05 C30F1F44	<b>add</b> eax,441F0FC3	636.25
ntdll.ZwReadVirtualMemory+E	00 00	<b>add</b> [rax],al	
<b>ntdll.ZwOpenEvent</b>			
ntdll.ZwOpenEvent	4C 8B D1	<b>mov</b> r10,rcx	
ntdll.NtOpenEvent+3	B8 3D000000	<b>mov</b> eax,0000003D	61
ntdll.NtOpenEvent+8	0F05	<b>syscall</b>	
ntdll.NtOpenEvent+A	C3	<b>ret</b>	
ntdll.NtOpenEvent+B	0F1F 44 00 00	<b>nop</b> [rax+rax+00]	

### NtReadVirtualMemory

For the RPM function, it looks fine. It does some stack and register set up and then calls ReadProcessMemory inside Kernelbase (Topic for another time). Which would eventually leads you down into ntdll's NtReadVirtualMemory. However, if you look at NtReadVirtualMemory and know what the most basic detour hook look like, you can tell that this is not normal. The first 5 bytes of the function is modified and the rest are left as-is. You can tell this by looking at other similar functions around it. All the other functions follows a very similar format:

0x4C, 0x8B, 0xD1, // mov r10, rcx; NtReadVirtualMemory

0xB8, 0x3c, 0x00, 0x00, 0x00, // eax, 3ch — aka syscall id

0x0F, 0x05, // syscall

0xC3 // ret

With one difference being the syscall id (which identifies the WINAPI function to be called once inside kernel land). However, for NtReadVirtualMemory, the first instruction is actually a JMP instruction to an address somewhere else in memory. Let's follow that.



47050084			
Address	Bytes	Opcode	Comment
47050084	48 B8 F06628F5FE070...	<b>mov</b> rax,CyMemDef64.dll+66F0	(608471372)
4705008E	FF E0	<b>jmp</b> rax	
47050090	48 B8 B06B28F5FE070...	<b>mov</b> rax,CyMemDef64.dll+6BB0	(610044232)
4705009A	FF E0	<b>jmp</b> rax	
4705009C	48 B8 506C28F5FE070...	<b>mov</b> rax,CyMemDef64.dll+6C50	("@UVAWH?IS?H??")
470500A6	FF E0	<b>jmp</b> rax	
470500A8	48 B8 F06E28F5FE070...	<b>mov</b> rax,CyMemDef64.dll+6EF0	("@UWAUAVAWH?IS?H??")
470500B2	FF E0	<b>jmp</b> rax	

CyMemDef64.dll

Okay, so we are no longer inside ntdll's module but instead inside CyMemdef64.dll's module. Ahhhhh now I get it.

The EDR placed a jump instruction where the original NtReadVirtualMemory function is supposed to be, redirect the code flow into their own module which then checked for any sort of malicious activity. If the checks fail, the Nt\* function would then return with an error code, never entering the kernel land and execute to begin with.

## The Bypass

It is now very self-evident what the EDR is doing to detect and stop our WINAPI calls. But how do we get around that? There are two solutions.

### Re-Patch the Patch

We know what the NtReadVirtualMemory function *SHOULD* look like and we can easily overwrite the jmp instruction with the correct instructions. This will stop our calls from being intercepted by CyMemDef64.dll and enter the kernel where they have no control over.

### Ntdll IAT Hook

We could also create our own function, similar to what we are doing in Re-Patch the Patch, but instead of overwriting the hooked function, we will recreate it elsewhere. Then, we will walk Ntdll's Import Address Table, swap out the pointer for NtReadVirtualMemory and points it to our new fixed\_NtReadVirtualMemory. The advantage of this method is that if the EDR decides to check on their hook, it will look unmodified. It just is never called and the ntdll IAT is pointed elsewhere.

## The Result

I went with the first approach. It is simple, and it allows me to get out the blog quicker :). However, it would be trivial to do the second method and I have plans on doing just that within a few days. Introducing AndrewSpecial, for my manager Andrew who is currently battling a busted appendix in the hospital right now. Get well soon man.

```

C:\Users\Node\Desktop>AndrewSpecial.exe
RPM: 0000000076E8C2A0 ----- ntRUM: 00000000770B02A0
Decoupling Stage 1 Rockets, Retrograde burn starting...
ntReadVirtualMemory Syscall is 3c
Press any key to continue . . .
Got lsass.exe handle 94
Successfully launched the AndrewSpecial. Mission Successful!

C:\Users\Node\Desktop>

```

When	Category	Event	Details
1/18/2019 1:49:59 PM	Exploit	Blocked	Violation: LsassRead; PID: 4036; Application: C:\Users\Node\Desktop\AndrewSpecial_bad.exe
1/18/2019 11:53:25 AM	Exploit	Blocked	Violation: LsassRead; PID: 1680; Application: C:\Users\Node\Desktop\AndrewSpecial_bad.exe
1/18/2019 11:21:15 AM	Exploit	Blocked	Violation: LsassRead; PID: 3140; Application: C:\Users\Node\Desktop\procdump64.exe

AndrewSpecial.exe was never caught :P

## Conclusion

This currently works for this particular EDR, however — It would be trivial to reverse similar EDR products and create a universal bypass due to their limitation around what they can hook and what they can't (Thank you KPP).

Did I also mention that this works on both 64 bit (on all versions of windows) and 32 bits (untested)? And the source code is available [HERE](#).

Thank you again for your time and please let me know if I made any mistake.