



# Shellcodeless JOP: Advanced Code-Reuse Attacks with Jump-Oriented Programming

Bramwell Brizendine<sup>1</sup>, Austin Babcock<sup>2</sup>

<sup>1</sup> Dakota State University, Madison SD 57042, USA  
[bramwell.brizendine@dsu.edu](mailto:bramwell.brizendine@dsu.edu)

<sup>2</sup> Dakota State University, Madison SD 57042, USA  
[austin.babcock@trojans.dsu.edu](mailto:austin.babcock@trojans.dsu.edu)

**Abstract.** Shellcode is often used as a means to achieve arbitrary execution. In Windows, this involves the use of WinAPI functions or Windows syscalls. Historically, an end goal of much of exploitation has been to bypass Data Execution Prevention (DEP) and to prepare the process memory of a binary to such a state that shellcode could be executed. Jump-Oriented Programming (JOP) is a seldom studied form of advanced code-reuse attacks, very different from return-oriented programming (ROP). JOP identifies snippets of code ending in an indirect jump or indirect call (gadgets), and these are chained together to construct exploits. In this paper, we propose and implement shellcodeless JOP attacks. This provides an alternative means of achieving identical functionality to what is done in shellcode, but without the need to bypass DEP. In this paper, we present the design and implementation of shellcodeless JOP, providing a new technique that red teams can use in exploits. In ideal circumstances, our shellcodeless JOP technique can allow for complex shellcode-like functionality to be achieved with ease or with only a relatively small amount of space required for the payload.

**Keywords:** Jump-Oriented Programming, Shellcodeless JOP, Shellcode, Return-Oriented Programming, Reverse Engineering, Software Exploitation, Cyber Operations

## 1. Introduction

Most code-reuse exploit writing techniques focus on leveraging chunks of preexisting code snippets to trigger the execution of arbitrary shellcode. While EDR can defend against shellcode in various ways, ultimately there are myriad techniques to allow for shellcode to work under some circumstances. Methods used to achieve this vary between different attacks. Some exploits may simply redirect the instruction pointer to the shellcode, while others may need to perform elaborate setup first to bypass mitigations. Commonly, Return-Oriented Programming (ROP) exploits will construct a call to a function such as VirtualProtect to change the memory protections set on the region of memory where the shellcode is contained. These goals are not specific to ROP, but could be extended more broadly to code-reuse attacks, which includes Jump-Oriented Programming (JOP). Although the mechanisms used to exploit native

binaries are strikingly different in JOP, the high-level goals are identical: bypass mitigations such as DEP and ASLR and execute shellcode. We could extend this even further to other advanced and specific mitigations, such as those found with Windows 10/11 Exploit Protection, that must be dealt with and addressed.

Often, an end goal with code-reuse attacks is to set up and allow for shellcode to be executed, allowing for arbitrary execution. Although shellcode is position independent, meaning it does not have access to linked libraries, there are ways around that, such as walking the PEB and traversing the PE file format, allowing for API function addresses to be found in the exports table[1]. Once these are found, the APIs can be called freely with appropriate parameters. Often, as with shellcode from MSFVenom, encoding can be employed to obscure analysis, although the core functionality remains largely identical from shellcode to shellcode. Ultimately, it is calling a series of APIs with parameters, while trying to obscure and obfuscate what is being done. While shellcode is one effective way to accomplish this, another way is simply to call APIs directly via JOP. The exploit author can have identical functionality, with the same API's being called, the same parameters, the same structures, etc., but implemented in a different fashion. We christen this technique shellcodeless JOP. This is a way of full embracing the nature of code-reuse attacks, eliminating the need for shellcode or to even bypass DEP. Instead, we can all these APIs directly.

Shellcodeless code-reuse attacks have been done previously before, in the context of ROP exploitation [2]. This whitepaper introduces a similar technique for JOP. The idea behind a shellcodeless JOP attack is to construct one or more Windows API calls that will perform the malicious actions normally handled by shellcode. Under ideal circumstances, with just a relatively small set of gadgets and capabilities, a considerably long and powerful series of malicious API calls can be constructed. These types of attacks can prove to be just as powerful as those using shellcode. This should be partly self-evident given the fact that identical API calls in many instances can be called with the same parameters, the results of which can be used in subsequent APIs, involving multiple complex structures. In other cases, some of this may be constrained by Windows 10/11 Exploit Protection, but in other cases, some of those features will not be used, and the shellcodeless JOP attack can proceed without restrictions, allowing for matching functionality to be achieved. It is also useful to bear in mind that some may still be using older versions of the Windows operating systems with more limited threat protection.

We can distinguish shellcodeless JOP attacks from those of normal shellcode in the skill level required to achieve them. At present, these must be written by hand, so it demands not only an ability to utilize Jump-Oriented Programming, but the attacker must also be versatile and knowable in Assembly, the writing of shellcode, and the usage of Windows APIs in programming. While shellcode has many unique conventions that the author of shellcodeless JOP attacks need not concern themselves with, it would be wise to have some familiarity with some of the functionality commonly present in shellcode. A skilled author of shellcodeless JOP attacks could implement an extremely large range of functionality. For instance, a malicious executable could be downloaded and executed via a shellcodeless JOP attack, as we will discuss throughout this paper. In our @Hack 2021 talk, we provide a live demonstration of such a shellcodeless JOP attack.

This paper presents a model for shellcodeless JOP, delineating the requirements for achieving such. Section 2 introduces some of the relevant background with shellcodeless ROP attacks in the Windows environment and a background on JOP itself. Section 3 presents the design of shellcodeless JOP. Section 4 is the central focus of this paper, as we discuss the implementation details of shellcodeless JOP, while also providing an instantiation of shellcodeless JOP.

## 2. Background

In section two, we provide a background on shellcodeless ROP attacks, limited as it may be. We also provide a brief introduction to JOP as well, highlighting some of the past work that has been doing.

### 2.1 Shellcodeless ROP Attacks

Prior to our work, there has been no known usage or concept of a shellcodeless JOP attack. However, there have been instances where ROP has been used in a similar manner. We could call these shellcodeless ROP.

An example of a recent shellcodeless ROP exploit from Cooke [2] effectively demonstrates the power of shellcodeless ROP. His exploit utilizes three APIs and four calls to reach its goal. First, the exploit utilizes `LoadLibrary` to load the `msvcrt.dll` library and obtain a handle to this module. Next, the handle is used with `GetProcAddress` to obtain the address of the `System` function, which can execute an arbitrary command. A function call to this API is constructed, using the command to add a user onto the system. Following this, another function call to `System` is used to elevate the account to Administrator privileges. Throughout the exploit, sequences of gadgets are used to write values into memory which would otherwise be impossible due to null bytes. Despite the complete lack of shellcode found within the exploit, the author has still managed to create a new Administrator level account on the victim machine. With Cooke's shellcodeless ROP attack, we can see that there is no need to even bypass DEP, as DEP is typically bypassed to allow for shellcode to be executed in exploits. He models his approach after the tried-and-true approach of using the *pushad* gadget to populate all register values with WinAPI parameters, prior to executing `VirtualProtect` or `VirtualAlloc`. Effective with those APIs, it is equally effective with others, though some require additional set up.

### 2.2 Jump-Oriented Programming

JOP is a state-of-the-art form of code-reuse attacks. Categorizing JOP may be useful as a human construct, but we emphasize these distinctions are arbitrary, as there can be intermixing of the different styles. The first method is the Bring Your Own Pop Jump (BYOPJ) [3], where a register can be loaded with an address, which is then executed. The next method utilizes the dispatcher gadget, allowing the attacker to craft a dispatch table in memory and use a dispatcher to execute individual functional gadgets [4]. The third approach to JOP [5–7] is a real-world variation on BYOP, combining functional and dispatcher gadgets as a more labyrinthine chain, allowing for a greater variety of indirect jumps and calls.

#### 2.2 Dispatcher Gadget Paradigm

The dispatcher gadget paradigm [4] is the approach this research favors. A dispatch table, containing addresses of functional gadgets, is created anywhere in memory. Functional gadgets can be viewed as being similar to ROP gadgets, used to deal with mitigations or set up WinAPI calls. The dispatcher is a special gadget that orchestrates control flow. It can advance forwards or backwards in a predictable fashion; it then dereferences and executes functional gadgets. An exploit writer can place functional gadgets inside the dispatch table. After each functional gadget, the dispatcher is called again, advancing to the next functional gadget until the JOP chain is complete, as seen in the diagram.

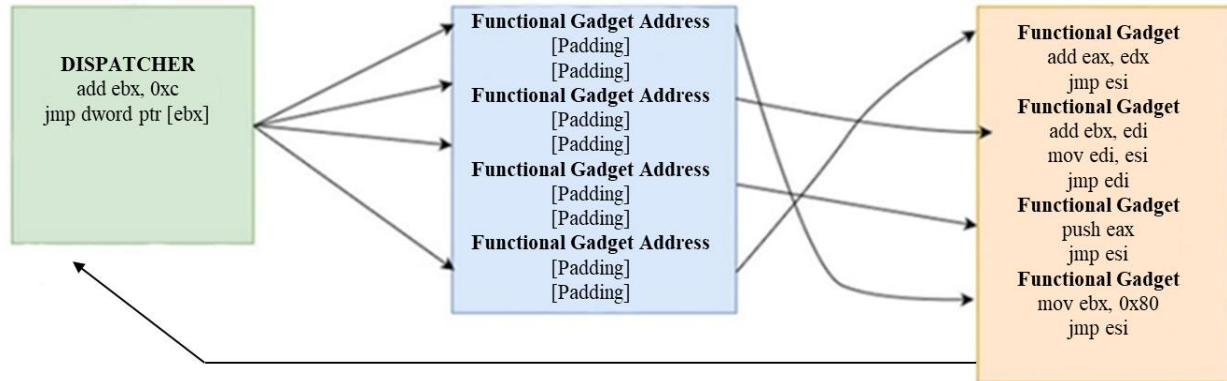


Figure 1. Control flow in JOP is established via a dispatch table and dispatcher gadget, allowing for functional gadgets to be executed one after the other.

While some make the distinction between JOP and call-oriented programming (COP)[8], they actually are one in the same. The primary difference is indirect calls push the address of the next instruction onto the stack. This could interfere with WinAPI arguments being set up. However, this can be compensated for with a small stack pivot, such as a *pop* or *add esp, 4*, restoring the stack to what it was. Thus, by intermixing indirect jumps and calls, we can significantly enrich the JOP attack surface. To distinguish between them seems unnecessarily pedantic, not reflective of real-world usage.

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

While ROP is widely used, JOP has only very rarely been so. In fact, prior to our work with JOP [9–12], there had only been a handful of academic articles on JOP [3, 4, 13] nearly a decade ago, and since then, virtually nothing. The primary reason for such was the lack of available tools, as finding all the appropriate JOP gadgets would be a time-consuming, tedious task without a dedicated tool. Moreover, there was considerable information missing on how to perform JOP in a modern Windows environment, and in fact there were no working exploits available for potential users of JOP. That has changed, with the introduction of the JOP ROCKET [9, 10, 14], allowing for JOP gadgets to be found. We have also provided considerable documentation on the mechanics of doing JOP, including many of the finer points. More recently, we have taken this to the next level, allowing for automatic generation of JOP exploits[11, 12]. This would allow a user to use the generated JOP chain to bypass DEP with VirtualProtect or VirtualAlloc. As with all such tools that use a determinate recipe to create an exploit chain, sometimes these may require additional adjustments.

### 3. Design

We propose a model for shellcodeless JOP, which forgoes the use of custom shellcode and instead executes API calls to perform malicious actions. Shellcodeless JOP is a code-reuse attack utilizing code snippets ending in *JMP* or *CALL* that constructs one or more API calls to perform complex functionality that would otherwise be performed via JOP. In a normal JOP exploit, often a goal is simply to set up API calls to bypass DEP. Generally, most of the mechanics of normal JOP exploits apply to shellcodeless JOP attacks.

In our design, Shellcodeless JOP attacks often begin with a series of stack pivots to move the stack pointer to a portion of controlled memory. Under ideal circumstances, we can have a series of API parameter values and pointers to the relevant API's already populated. However, often it will be necessary to use JOP gadgets to construct function parameters, as is often the case with ROP or JOP. Once an API call is ready, JOP will execute the function; later it will restore the JOP control flow if necessary, to maintain control over the exploit. Later, additional supporting API calls can be made to continue performing more complex actions. For certain functions, pointers to strings, structures, or other values may be required. These pointer values sometimes can be hardcoded into the exploit script itself or more likely generated dynamically via JOP gadgets. Likewise, data contained at the pointer location can be provided via the initial exploit script.

All exploits in shellcode employ Windows API functions, and similarly malware derives its functionality from the use of shellcode. Shellcodeless JOP attacks possess the capability to be just as strong as those other. As will be discussed in Section 4, exploits are able to reach a point where functions from any library can be executed, and execution of different functions can continue indefinitely until there is a lack of payload space for the exploit. With a short number of calls to the correct functions, powerful payloads equivalent to those of more traditional attacks can be achieved. Additionally, no API calls will need to be wasted bypassing DEP, as shellcodeless attacks do not execute custom code inserted into memory.

## 4. Implementation

This section will outline the steps required to implement an effective shellcodeless JOP exploit. Important notes for development of individual steps will be discussed, as well as solutions to various problems that arise along the way. Additionally, examples of powerful API calls capable of performing realistic exploitation tasks are examined in this section.

### 4.1 Constructing the Exploit

For our proposed shellcodeless JOP exploit to work, it must be possible to effectively construct API calls without ruining the control flow mechanisms of JOP. If control flow is lost and not restored, the program may terminate prematurely. Another major constraint is simply running out of available space within the payload. Lack of a large space may constrain some of the functionality that can be achieved. The steps required to successfully set up JOP and perform an API call are straightforward, though order may vary slightly from exploit to exploit: (1.) ensure JOP control flow registers are set; (2.) perform one or more stack pivots to a suitable location; (3.) build the required function parameters in memory; (4.) retrieve the address of the function and perform the call.

#### 4.1.1 Control Flow Registers

With JOP exploits, a dispatcher gadget and dispatch table need to maintain control over execution. Addresses of these need to be loaded into the proper registers before the exploit can start. In some cases, it may be possible to control the values of these registers via the payload itself without using any gadgets. However, it will almost always be necessary to use either a JOP gadget or a short series of ROP gadgets to set the JOP control flow registers. While using a JOP gadget to perform this action is preferable in that it allows for only JOP to be used throughout the exploit, it is often not feasible in practice. Since the JOP control flow is not yet established, this gadget would have to somehow load in values for both the dispatcher gadget and dispatch table, then passing execution to the dispatcher gadget. In some situations, bad bytes could be an issue, and bitwise operations may be required to help deal with this. In practice, such a JOP gadget will often not exist in a given binary. Instead, a short ROP chain can be used for the same purpose. The ROP gadgets that can accomplish this are plentiful for nearly any desired register.

### 4.1.2 Initial Stack Pivots

Next, a series of multiple stack pivots can be used. This step is often critical, as the location of ESP at the time of exploitation may land at a location far removed from where the payload is; thus we must redirect EIP to a location under our control. With JOP, often it is the case that we can use a series of stack pivots to reach a precise location. We can also provide padding to allow us to get to our desired destination more closely. In some cases, if we cannot precisely predict it, we can use a series of JOP nops to take us to the initial stack pivot gadgets. Stack pivots can relocate ESP to a place in memory where there is a large amount of space for the payload, and more control is available. For example, in some binaries, null bytes may not be accepted at the location of the stack at the beginning of exploitation. However, further down in memory an additional copy of the payload may occur where all the null bytes are included. Put another way, the initial payload could be truncated due to a null byte, but the rest of the payload could appear in memory elsewhere, potentially reachable via a series of stack pivots. For example, this may occur in a binary that crashes when parsing a string created after accepting a sequence of bytes over the network. It is important to check for this, as null-byte compatible shellcodeless JOP exploits are extremely powerful, as expanded upon in Section 4.2.1. In short, these sometimes can allow for highly complex malicious actions to be performed with relative ease, rather than having to more tediously construct each API parameter, one by one.

### 4.1.3 Building Function Parameters

First, it will be useful to check a resource such as Microsoft's Windows documentation before deciding on parameter values. In many cases, it is not necessary to worry about the exact purpose of every parameter. Many functions contain parameters that are either optional or are useful only when combined with other function calls in a traditional programming scenario. For example, consider the `CreateProcessA` function:

```

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

```

Figure 2. MSDN documentation for `CreateProcessA`.

Note how even without exploring documentation on each parameter, it can be seen that several are optional and likely will not need to be dealt with. Here is an example of a call to this function, showing in a debugger, during a shellcodeless JOP exploit:

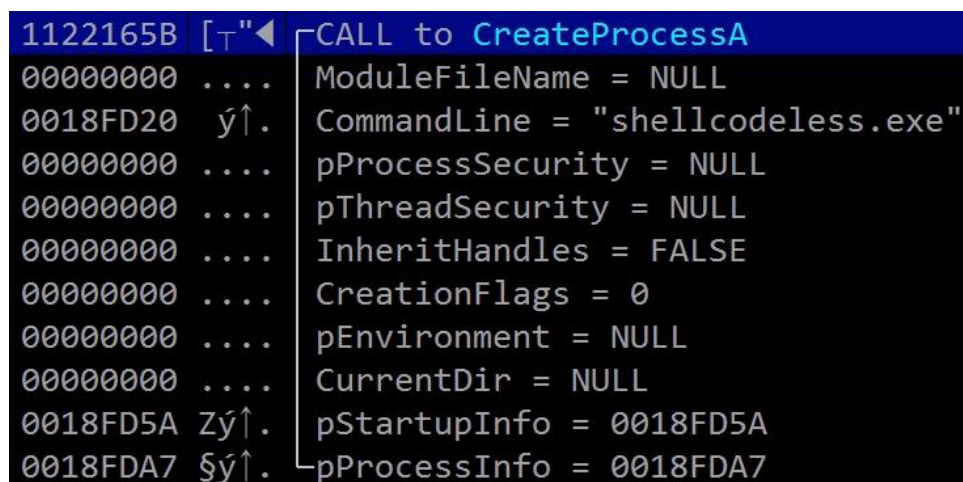


Figure 3. Immunity debugger showing information regarding a call to `CreateProcessA`.

More than half of the parameters supplied for this particular function call are NULL or zero. This is because they are optional, as seen in the documentation; they serve no purpose in the context of the exploit. However, it will still be important to review the documentation on each parameter individually, as occasionally parameters marked as optional will still need specific values. For example, the `lpCommandLine` parameter is only optional if `lpApplicationName` has been supplied. If they are both NULL, the function will not know what command to execute. Sometimes it is the case that official documentation could be sparse or lacking, and alternative sources other than MSDN may be required.

It is also necessary to examine the variable type for each parameter as well. This can be seen in the middle column in the documentation above in figure 2. Additionally, Microsoft uses Hungarian notation for their parameter names, so the start of each name shows its type as well. Many parameters are pointers. Thus, a string for instance would not be supplied directly; instead, it would be necessary to provide a pointer to the start of the aforementioned string. The exploit author will need to supply the expected values in memory somewhere. That is to say, they are not likely to already be present in memory but would be supplied via some input from the malware author, such as the payload. Next, a pointer to that location in memory must be supplied, allowing it to be used as a parameter. While not always possible, it is best practice to programmatically construct the addresses of these pointers rather than hardcoding them into the payload. Given an option, programmatically supplying these values adds greater portability to the shellcodeless JOP, allowing for it to be more reliable.

Address	Gadget
0x43da8822	mov ebx, ebp; jmp ecx
0x62ad7355	pop eax; jmp ecx;
0x62bfbb11	add eax, ebx; jmp ecx;
0x62ad7132	push eax; jmp ecx;

Figure 4. Dynamic generation for pointer addresses. The `EBP` register is used as a reference point to the value of `ESP`.



The sequence of gadgets seen above is an example of dynamic generation of pointer addresses. First, a value related to the stack pointer is put into EBX using the MOV instruction. Afterwards, an offset to the correct value in memory is popped into EAX. This value and the stack address are added together. Finally, the value is pushed onto the stack as a parameter. Steps such as these ensure that addresses are consistently correct, even if the stack's location changes when running the exploit in different environments.

It is also important to consider that different types of pointer variables will expect different types of data at the address. Looking at figure 2, it can be seen that of the parameters that were not null or zero, three different pointer types were used: LPSTR, LPSTARTUPINFOA, and LPPROCESS\_INFORMATION.

The first type, LPSTR, should be familiar to most programmers and exploit authors – it is simply a null-terminated string. While this type is a simple ASCII string, it is possible that a function will ask for a LPWSTR – that is, a pointer to a wide-character string. Technically, the formatting of wide-character strings can vary, but it is part of Microsoft's coding conventions to use UTF-16LE. For our purposes, UTF-16LE should be safe to use without problems arising. In some cases, it may be beneficial to encode plaintext strings to avoid detection. These could then be decoded by simple instructions, such as XOR gadgets. In actual practice, this would involve additional time and labor to set up, and it may not always be possible, given the available attack surface. Using **stack strings**, and pushing strings one DWORD at a time, also could help lower detection rates by EDR.

The LPSTARTUPINFOA and LPPROCESS\_INFORMATION pointers, on the other hand, refer to custom structures that are specific to this and other related functions. When faced with structure pointers such as these, more research into the documentation will be necessary. Just as the function itself contains a list of variables with specific types and requirements, so too does each structure. It will be important to view documentation each variable in the structure and determine which, if any, need to be specific values to allow the function to execute properly. After having determined the required values for each portion of the structure, it must be built in memory somewhere. Like the previously discussed structure, a point for it must be found and supplied as the appropriate function parameter.

```
typedef struct _PROCESS_INFORMATION {
    HANDLE hProcess;
    HANDLE hThread;
    DWORD  dwProcessId;
    DWORD  dwThreadId;
} PROCESS_INFORMATION, *PPROCESS_INFORMATION, *LPPROCESS_INFORMATION;
```

Figure 5. MSDN documentation for the PROCESS\_INFORMATION structure required by the CreateProcessA function.



```
# dummy structure for use in CreateProcessA

processInfoStruct = struct.pack('<L', 0x00000000)
processInfoStruct += struct.pack('<L', 0x00000000)
processInfoStruct += struct.pack('<L', 0x00000000)
processInfoStruct += struct.pack('<L', 0x00000000)
```

Figure 6. Exploit code showing construction of a fake `PROCESS_INFORMATION` structure.

Once the desired parameter values are known, they must be put into memory so the function can use them. The process of writing function parameters may vary depending on the gadgets available, function being called, and whether bad bytes are an issue. If bad bytes are not an issue and the desired parameter values are known and do not need to be dynamically generated, it may be possible to write them directly into the payload. ESP will need to be pivoted to the start of the parameters in memory, then the function can be called. If parameters need to be generated via JOP, they will need to be written into memory somehow.

One way to write a parameter into memory is via the *push* instruction. In many cases, it will be most efficient to use a combination of JOP-generated parameters and parameters supplied directly with the payload. For this reason, a stack pivot should often be used to make sure the *push* places the finished parameter onto the stack at the correct location without overwriting any existing parameters. An additional stack pivot may be needed before calling the function to ensure the correct parameters are being used. The correct location for the stack pivot to land on is the very beginning of the set of parameters – the return address.

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

Figure 7. Writing a parameter to memory via the *PUSH* operation..

The figure above shows an example of a parameter being written to memory. The first gadget is used to *pop* a parameter value into ECX, and an XOR key into EDX. Next, an *xor* instruction leaves the finished value in ECX. A stack pivot is used to move ESP to the correct location before *push ecx* writes the finished parameter into memory. If the parameter being written was the return address, ESP would already be in the correct location for a function call, provided the rest of the parameters were already finished. However, if this was a different parameter, calling the function without performing an additional stack pivot would result in the function using incorrect parameters.

Other methods of writing values to memory for use as parameters exist as well. Gadgets containing instructions such as *mov dword ptr* may allow the developer to perform parameter writes without the use of stack pivoting. However, unlike with ROP, gadgets containing this instruction are

relatively rare in JOP, and thus likely they will not be an option. It can be possible to get creative with other ways of writing values directly to memory. For instance, there are many other instructions that can deference a memory location and allow a value under the attacker's control to be moved there, such as *xor*, *add*, *sub*, etc.

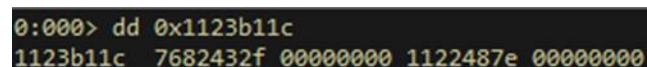
#### 4.1.4 Retrieving the Function Address

In addition to ensuring that parameters have been set up properly for an API call, the address of the API function itself is needed. The simplest way is to it may be possible to find a pointer to the required function within the binary somewhere, and simply deference the pointer and JMP to the address. This can be relatively simple, and many tools exist to find these. JOP ROCKET finds pointers to VirtualAlloc and VirtualProtect. However, shellcodeless JOP will often require more exotic types of API calls to perform the malicious actions a shellcode normally would. Pointers to these functions often are unavailable, and other methods will need to be explored in order to gain access to them.

##### 4.1.4.1 Function Offsets Within DLLs

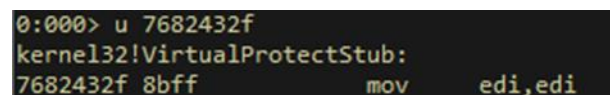
One option to find function addresses without a direct pointer is through the use of another function address that is already available within the binary. If the available function and the desired function are contained within the same DLL, the new function address can be found by finding the offset of each function within the DLL. The distance between these two functions can then be calculated, and the result can be used as an offset to reach the desired function. Although this technique relies on the specific version of the DLL being used and thus lacks portability, it can provide access to new APIs where otherwise there would be none. While it is dependent upon a specific version of a DLL, if both parties have the same release of Windows that has been updated, there is a reasonable likelihood they may have the same DLL. Thus, the currency of shellcodeless JOP could be limited; exploits may need to be updated from time to time. While not difficult to do, this does require a more advanced skill set than a so-called script kiddie would possess.

To get the address of a function via an offset, the actual function location referenced by the pointer of the original function should be verified. A debugger can be used to dereference the pointer and inspect the disassembly at the address. Though it should be mentioned that the exact combination of tools used to find these offsets can vary.



```
0:000> dd 0x1123b11c
1123b11c  7682432f 00000000 1122487e 00000000
```

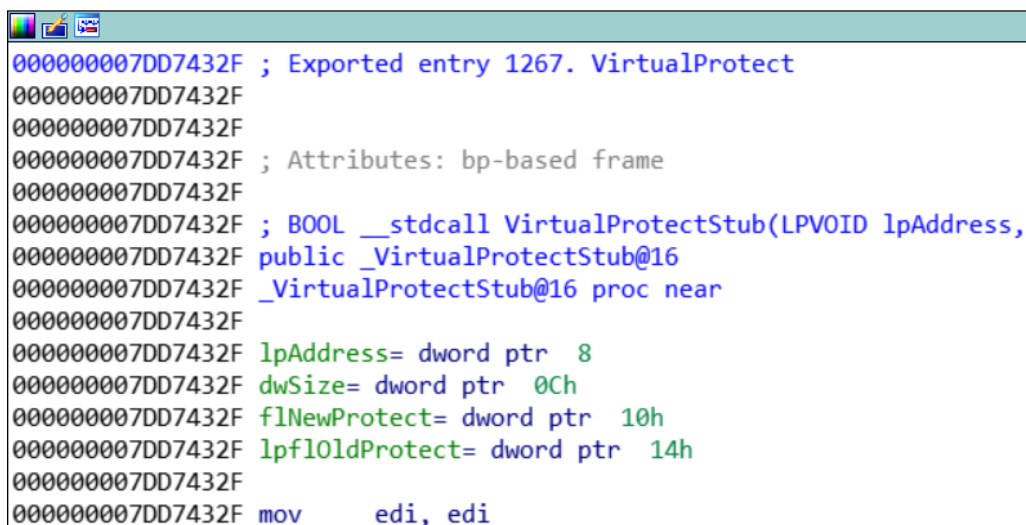
Figure 8. WinDbg screenshot -- dereferencing a VirtualProtect pointer.



```
0:000> u 7682432f
kernel32!VirtualProtectStub:
7682432f 8bff      mov     edi,edi
```

Figure 9. WinDbg screenshot -- inspecting the destination of the address.

After the location has been verified, a tool such as IDA Disassembler can be used to inspect its virtual address within the DLL. Below, it is shown that the corresponding virtual address is 0x7dd7432f:



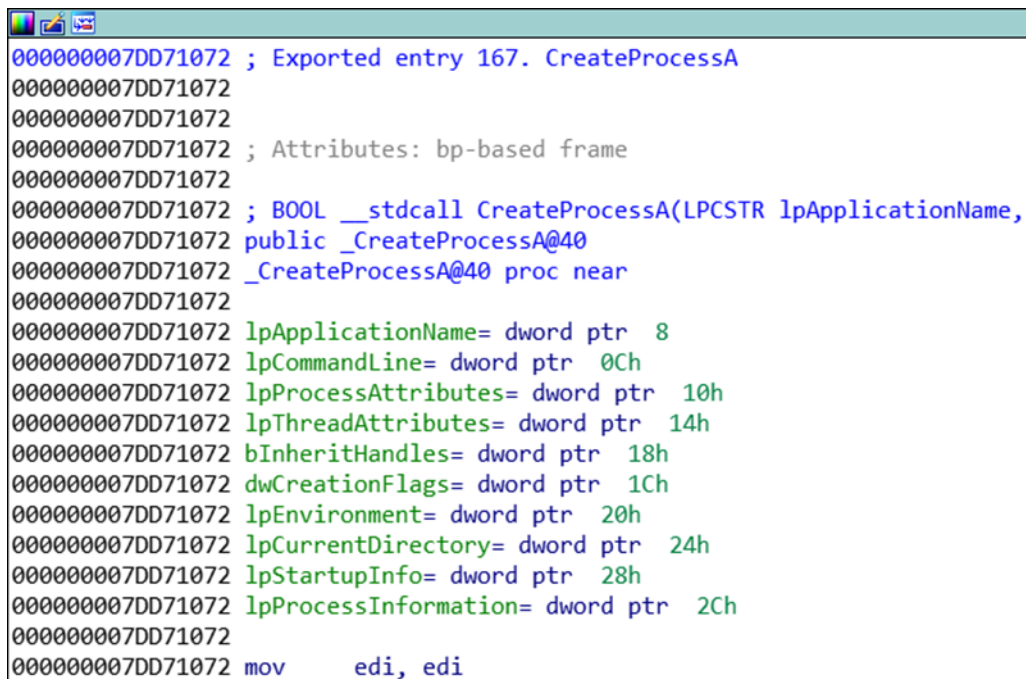
```

000000007DD7432F ; Exported entry 1267. VirtualProtect
000000007DD7432F
000000007DD7432F
000000007DD7432F ; Attributes: bp-based frame
000000007DD7432F
000000007DD7432F ; BOOL __stdcall VirtualProtectStub(LPVOID lpAddress,
000000007DD7432F public _VirtualProtectStub@16
000000007DD7432F _VirtualProtectStub@16 proc near
000000007DD7432F
000000007DD7432F lpAddress= dword ptr 8
000000007DD7432F dwSize= dword ptr 0Ch
000000007DD7432F flNewProtect= dword ptr 10h
000000007DD7432F lpflOldProtect= dword ptr 14h
000000007DD7432F
000000007DD7432F mov     edi, edi

```

Figure 10. The function's virtual address is found using IDA.

Later, this virtual address can be used to calculate the distance to a different function within the same DLL. For example, the virtual address of CreateProcessA is 0x7dd71072. By taking the difference between these two addresses, the offset from VirtualProtect to CreateProcessA is -0x32bd bytes.

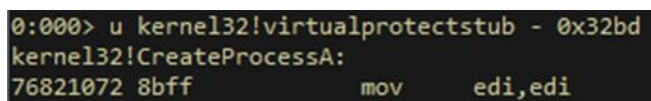


```

000000007DD71072 ; Exported entry 167. CreateProcessA
000000007DD71072
000000007DD71072
000000007DD71072 ; Attributes: bp-based frame
000000007DD71072
000000007DD71072 ; BOOL __stdcall CreateProcessA(LPCSTR lpApplicationName,
000000007DD71072 public _CreateProcessA@40
000000007DD71072 _CreateProcessA@40 proc near
000000007DD71072
000000007DD71072 lpApplicationName= dword ptr 8
000000007DD71072 lpCommandLine= dword ptr 0Ch
000000007DD71072 lpProcessAttributes= dword ptr 10h
000000007DD71072 lpThreadAttributes= dword ptr 14h
000000007DD71072 bInheritHandles= dword ptr 18h
000000007DD71072 dwCreationFlags= dword ptr 1Ch
000000007DD71072 lpEnvironment= dword ptr 20h
000000007DD71072 lpCurrentDirectory= dword ptr 24h
000000007DD71072 lpStartupInfo= dword ptr 28h
000000007DD71072 lpProcessInformation= dword ptr 2Ch
000000007DD71072
000000007DD71072 mov     edi, edi

```

Figure 11. IDA reveals CreateProcessA's virtual address.



```

0:000> u kernel32!virtualprotectstub - 0x32bd
kernel32!CreateProcessA:
76821072 8bff      mov     edi,edi

```

Figure 12. WinDbg shows that the calculated difference between the two virtual address is correct.

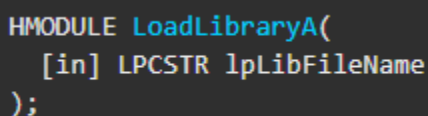
Using this method, the address of a function has been found without the use of an existing pointer within the binary.

#### 4.1.4.2 LoadLibrary and GetProcAddress

When it is possible to do so, the preferred method to locate the address of a new function is to utilize the LoadLibrary and GetProcAddress functions. A function from the LoadLibrary family of functions (e.g. LoadLibraryA, LoadLibraryExA, LoadLibraryW, LoadLibraryExW) takes a string parameter containing the name of the desired library. If the corresponding library is found, it will be loaded as a module if necessary and a handle to the module will be returned into EAX. This handle is the base address of the module in question. The returned handle can then be used in conjunction with GetProcAddress to get the address of any function found within the library. Widely used in malware and exploitation, these functions can also sometimes be problematic if Windows Defender Exploit Guard, the successor to EMET, is used and employing specific mitigations that protect against sensitive APIs.

While this method does require the use of two auxiliary API calls every time a new function needs to be called, the use of LoadLibrary and GetProcAddress provides many opportunities within shellcodeless JOP exploitation. With access to only two functions, a plethora of APIs immediately become available to the exploit author, allowing for payloads of many different types to be executed. As long as it is possible for the exploit to perform a call to these two functions and the third retrieved function without losing the JOP control flow, it is likely possible for the exploit to maliciously call several APIs from different libraries near-indefinitely—that is, until payload space becomes a limiting factor.

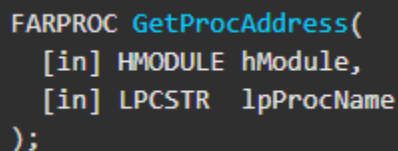
Additionally, the setup process for LoadLibrary and GetProcAddress is simple. The versions of LoadLibrary not containing the “Ex” identifier only take one parameter: a string containing the name of the desired library.



```
HMODULE LoadLibraryA(  
    [in] LPCSTR lpLibFileName  
);
```

Figure 13. MSDN documentation for LoadLibraryA.

After returning from the function and regaining the JOP control flow, the setup for GetProcAddress only requires two parameters: the handle returned from LoadLibrary, and a string containing the name of the desired function.



```
FARPROC GetProcAddress(  
    [in] HMODULE hModule,  
    [in] LPCSTR lpProcName  
);
```

Figure 14. MSDN documentation for GetProcAddress.

As the handle for the corresponding library should already be contained within EAX following a successful LoadLibrary call, the setup for GetProcAddress may be rather short as well. After getting the address of the GetProcAddress function itself, the figures below show one possible way to set up the function call:

Address	Gadget
0x112226f1	pop ecx; jmp edx;
0x11223649	push eax; jmp edx;
0x112236be	push ecx; jmp edx;

Stack (Before Gadgets Execute)		
Address	Value	Notes
0x0018fac0	0x11111111	Junk
0x0018fac4 ( <b>esp</b> )	0x11221658	Return Address
0x0018fac8	0x0018fb72	ProcName String Pointer

Stack (After Gadgets Execute)		
Address	Value	Notes
0x0018fac0 ( <b>esp</b> )	0x11221658	Return Address
0x0018fac4	0x75e20000	Handle to DLL
0x0018fac8	0x0018fb72	ProcName String Pointer

Figure 15. Pushing two parameters onto the stack to set up GetProcAddress.

Values are provided in the exploit such that the string pointer for GetProcAddress is already set up on the stack after returning from LoadLibrary. The first gadget in the sequence pops the return address value into ECX. Next, EAX and ECX are pushed onto the stack. EAX can be pushed without popping any values into the register since LoadLibrary already put the library handle into it. ECX is pushed afterwards to satisfy the requirements of function calls: return address first, then the rest of the parameters. After these two pushes execute, GetProcAddress is ready to be called.

There are equivalent functions to LoadLibrary and GetProcAddress found within ntdll.dll as well. These are LdrLoadDll and LdrGetProcedureAddress, respectively. The parameters necessary for these functions are very similar to those found within LoadLibrary and GetProcAddress, and it may be possible to use the ntdll.dll functions as an alternative, invoking them through a Windows syscall or other means. The use of syscalls directly in code-reuse attacks may help reduce potential detection from endpoint telemetry. It does take additional skill and time to set up syscalls, and the portability of these can be limited as well, as the syscall values that are placed inside EAX tend to change with different releases of Windows. Similar usage of syscalls via ROP may be found in Forrest Orr's exploit code for CVE-2020-0674 [15].

## 4.2 Additional General Problems

We have already discussed several issues that may arise during the development of a shellcodeless JOP exploit. Those have been specific to the unique mechanics of setting up shellcodeless JOP, and supplying all the needed parameters. However, there are other more general problems that can arise that are not necessarily specific to shellcodeless JOP, but can affect all manner of JOP. A few problems are notable: (1.) bad Bytes; (2.) function calls breaking JOP control flow; (3.) reuse of dynamically generated values.

### 4.2.1 Bad Bytes

Often, an exploit payload will not be compatible with certain bytes. The particular bytes and explanation may vary, but this issue is often related to string parsing. In most instances, string parsers read in bytes until they encounter a character which signifies the end of the string. This is usually the null byte `\x00`. Other whitespace characters such as carriage return `\x0d` or line feed `\x0a` may signal a parser to stop reading the string as well. If the payload is read into memory as a string, the inclusion of these characters in the payload will ruin the exploit, preventing it from being read in fully. Instances of other characters breaking exploits for different reasons are possible and exist as well.

To overcome this problem, JOP gadgets can be used to programmatically generate bad byte values without supplying the literal values within the payload directly. A useful way to do this is via the *xor* instruction.

Address	Gadget
0xDEADC0DE	POP EAX; POP EBX; JMP ECX # Load EAX and XOR key
0xDEADC1DE	XOR EAX, EBX; JMP ECX # XOR results in 0x40

## STACK

Address	Value
0x11223340	0x55555515
0x11223344	0x55555555

Figure 16. XORing a key with an encoded values gives a result that contains null bytes.

In the example above, an encoded value is loaded into EAX, and an XOR key is stored in EBX. After the values are loaded, the *xor* instruction puts the value 0x40 into EAX. Since the DWORD form of 0x40 is 0x00000040, this value could not have been put directly into the payload without the `\x00` bytes triggering a bad byte issue. This method is useful as XOR allows for a high degree of flexibility regarding possible values reached from starting values. Additionally, simple gadgets such as *xor eax, eax* allow for null bytes to be loaded with a single gadget. Since any value XORed with itself is 0, these gadgets essentially clear a register, allowing for 0 to often be supplied with ease. Other gadgets with *add*, *sub*, *neg*, and more can all be used to avoid bad bytes as well.

Shellcodeless JOP exploits that do not have any bad byte problems have the potential to be immensely powerful. Since many API calls do not need have parameter values generated via JOP, they can be put directly into the payload. All that needs to occur is the retrieval of the function address and one or more stack pivots, allowing the function to be called immediately. If additional function addresses and parameters are known, the return address of one function can lead directly into a call to another function,



assuming parameters are set up correctly on the stack. A binary that supports this type of shellcodeless JOP attack can be very powerful, as it can be easy to develop a complex attack while using a relatively small amount of payload space. Additionally, the normal restrictions of JOP such as limited gadgets and restricted register usage become less of a concern, as complex strings of JOP gadgets are not as important as they would be if bad bytes needed to be avoided. Put more plainly, the absence of restrictions on bad bytes coupled with the ability to do multiple stack pivots to precisely reach the payload can allow for a shellcode JOP attack to be developed with great ease. It would require just a knowledge of WinAPI functions, without the need for more advanced manual JOP usage.

#### 4.2.2 Function Calls Breaking JOP Control Flow

Most Windows API functions are highly complex and use most of the registers in their operation. Some involve calling multiple other functions. Occasionally, the registers reserved for the dispatch table and dispatcher gadgets will remain intact after a function's completion, but most of the time they will be either cleared or overwritten with other values. If there are no other functions to be called afterwards, this fact may not be a major issue; however, many shellcodeless JOP payloads will utilize several function calls to perform malicious calls.

While at first this may seem like an impassable obstacle, the solution is relatively straightforward. To remedy this problem, similar steps to those outlined in Section 4.1.1 can be used. Instead of setting the function's return address to the next part of the exploit, set it to a gadget or series of gadgets capable of restoring the correct values into the control flow registers. Since the registers had already been set up prior to this point in the exploit, it is nearly certain that a suitable solution exists. Additionally, not all function calls will overwrite every register. Even if a solution for both control flow registers does not exist, there may still be an available gadget to fix a singular register.

```
##### URLDownloadToFileA RETURNS TO gadget (pop edx; pop ecx; pop ebx; jmp ecx)
stackChain3 = struct.pack('<L', 0x1122165b) #dispatcher addr for POP ECX
stackChain3 += struct.pack('<L', 0x1122165b) #dispatcher addr for POP EDX
stackChain3 += struct.pack('<L', 0x1123d05c) #loadLibraryExW() ptr for POP EBX
```

Figure 17. Exploit code showing JOP registers being repaired. Here, the dispatch table address is put into ECX, and the dispatcher gadget is put into EDX.

#### 4.2.3 Reuse of Dynamically Generated Values

Values that are generated at the runtime of the exploit often need to be used multiple times, and the steps taken to load them may need to be repeated as well. This may be tolerable for small sequences, but it is not desirable for more involved processes. For example, retrieving the same function address through the use of LoadLibrary and GetProcAddress should not be repeated many times if avoidable.

Instead of repeating all the steps to generate a value, the value can simply be written to memory at a known location. The value can be retrieved using whatever methods available via JOP each time it needs to be reused. A simple way of doing this would be to use a stack pivot in conjunction with a *pop* instruction to grab the stored value out of memory. Alternatively, a gadget such as *mov eax, dword ptr [ebx]* could be used to load the value via a dereference.



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

Figure 18. A small series of gadgets stores the address of LoadLibrary in memory for later use.

In the above example, it is shown that storing a value in memory does not have to be a complex process. Once the address of LoadLibrary is obtained, it is simply pushed onto the stack. As long as no other values are pushed over that address, the value will stay intact in memory. Since some functions use the stack for many reasons, it may help to stack pivot before performing the *push* overwrite if a function will be called before the value is retrieved.

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

Figure 19. Later, the stored LoadLibrary address is retrieved from the stored location.

The figure above shows the retrieval process once the stored value is needed. Again, the gadgets utilized are simple – a *pop* to obtain the stored value, and stack pivots to ensure that the correct value is popped and the exploit returns ESP to its starting location. While the needed gadgets may not always be present to allow for this, if they are, the reusable nature of JOP allows for them to be used repeatedly.

### 4.3 Useful API Examples

In implementing our model of shellcodeless JOP, we instantiated it with various shellcodeless JOP exploits. These were intended to serve as proof of concept, allowing multiple API's from different DLLs to be used together to do something potentially significant in a malicious context. In this section, we highlight some of the API's that could be useful in this regard. While the use of LoadLibrary and GetProcAddress shown in section 4.1.4 does open doors to thousands of potential APIs, many will not be useful in reproducing tasks normally done through shellcode. Certain functions, especially those able to perform arbitrary Windows commands, will be important.

#### 4.3.1 URLDownloadToFile

This function takes a URL path to a file and will attempt to download the file from the internet. The file is saved to the path specified. Each parameter can be set to NULL except for the URL and filename string parameters, so effort to setup this function is low.

### 4.3.2 System

The System function simply takes a singular string and executes it as a Windows command. If this function is available, it is extremely flexible and powerful as it is essentially a shell on the machine. With relatively low effort, the command could be called multiple times using shellcodeless JOP to great effect.

### 4.3.3 CreateProcess

While this function does have a long parameter list, almost all are optional and can be set to 0x0, including the values in the two required structures. Since this function spawns a new process using the specified path, this function could easily be used in conjunction with another, such as URLDownloadToFile. This way, a malicious executable could be downloaded from the internet and then executed on the victim machine by CreateProcess.

### 4.3.4 WinExec

Similar to CreateProcess, WinExec takes a command line for an application to be executed. The only other parameter after the command line string is one regarding how the window should be displayed when launching the application. Since the parameter list is shorter, it may be easier to call this function instead of CreateProcess when trying to start an application.

### 4.3.4 LoadLibrary/GetProcAddress

As previously shown, these functions can be invaluable as a set of utility functions, able to retrieve the address of the specified function.

LoadLibrary has many versions; however, they all take a string pointer to the name of the library as a parameter: LoadLibraryA, LoadLibraryExA, LoadLibraryW, and LoadLibraryExW.

It is preferable to use LoadLibraryA if possible since it has the easiest and shortest setup, though the other functions work just as well. As previously mentioned, the LoadLibrary versions containing “Ex” have more options, and the versions containing “W” take wide character strings instead of normal strings.

GetProcAddress simply takes the handle to the DLL containing the function, and a string or ordinal number referring to the desired function. The DLL handle is found by calling LoadLibrary, so the value of EAX will likely need to be written to memory somehow for use as a parameter in GetProcAddress. Alternatively, GetModuleHandle can be used as well, although this will not work if the module is not already loaded.

## 5. Validation

To evaluate our proposed design, we implemented a number of proof of concept (PoC) exploits. These have been executed in the Windows environment, testing them to see if they functioned as intended. We have intentionally avoided using all the features available from Windows Defender Exploit Guard, as these are optional. Our intent with this work is to show the feasibility of this approach in allowing for complex and advanced shellcodeless JOP attacks to be performed. We acknowledge that utilizing every feature of Exploit Guard or some EDR features could prove problematic. Bypassing or avoiding some of these would have additional steps that could be taken to avoid them – or alternatively approaches that could be done in the design of the shellcodeless JOP attack. Given a particular binary, some may not have other options available. However, we reiterate that our attacks demonstrate the feasibility of this. In many cases, some defensive features may not be utilized, or a target may be using an older operating system or environment where certain defensive features may not exist. Many of the

system mitigations enabled by default will not have a bearing on our exploit. CFG can provide some difficult, although as explain in our previous work [11], there are many situations where CFG can be avoided or may not provide full defense against JOP, and that is assuming binaries were compiled to be compliant with CFG – which they may not be. Many of the mitigations that could prove most problematic, such as those that came from the now deprecated EMET, such as StackPivot, CallerCheck, SimExec, etc., are opt-in. For many who are not highly security conscious, we can assume these have not been opted into. While trendy to offer bypasses of all mitigations, that is not a focal point of this research, and we are confident that in many cases, existing ways to bypass or avoid them could be effective.

In this section, we will highlight two PoC exploits written to prove the feasibility and efficacy of the shellcodeless JOP technique. While bad bytes are not a concern in either exploit, both still require the use of JOP to write various function parameters to memory. After initializing JOP control flow registers, both exploits perform a stack pivot into a region of memory compatible with bad bytes. The binary each payload is exploiting contains a pointer to LoadLibraryExW, which the PoCs dereference and use to call the function. The parameters for this function are supplied directly within the payload and do not need to be manually altered. Once complete, the PoCs repair the broken JOP control flow registers through the use of a JOP gadget. Next, they use the address of LoadLibraryExW and a calculated offset to get the address of the GetProcAddress function. Once this is loaded, the exploits write two parameter values to memory and call the function.

The first PoC uses LoadLibrary and GetProcAddress to obtain the address of the System function found within msvrt.dll. System is able to execute an arbitrary string as a command line command, and the particular command for the exploit can easily be switched out by altering this string in the exploit script. For demonstration purposes, this PoC runs the command “shutdown /l”, which immediately logs the victim user out of their machine. The function is called directly after GetProcAddress returns its address. Since the callable address is returned into EAX and parameters for System are already set up via the exploit script, the return address for GetProcAddress is simply set to a *jmp eax* gadget which calls System.

The second PoC demonstrates the feasibility of many function calls to combine to form a powerful and realistic payload, permitting complex functionality via shellcodeless JOP. LoadLibraryExW and GetProcAddress are still used in a similar fashion to the first PoC, but this time the exploit performs two malicious API calls rather than just one. First, LoadLibrary and GetProcAddress are used to load the urlmon.dll library and find its function URLDownloadToFileA. This function is then called, causing an executable to be downloaded from the given address and saved as shellcodeless.exe. Then, control flow registers are repaired again, with LoadLibrary and GetProcAddress being called another time. LoadLibrary is used to obtain the handle of kernel32.dll which is already loaded, and GetProcAddress grabs the location of CreateProcessA. Finally, CreateProcessA is called to launch the newly downloaded executable. This PoC showcases the capability of shellcodeless JOP to create a realistic attack that performs similar actions to those taken by written elaborate shellcode.

Address	Disassembly	Comment
1122165B	[CALL to CreateProcessA	
00000000	...	ModuleFileName = NULL
0018FD20	↑. y	CommandLine = "shellcodeless.exe"
00000000	...	pProcessSecurity = NULL
00000000	...	pThreadSecurity = NULL
00000000	...	InheritHandles = FALSE
00000000	...	CreationFlags = 0
00000000	...	pEnvironment = NULL
00000000	...	CurrentDir = NULL
0018FD5A	↑. Z y	pStartupInfo = 0018FD5A
0018FDA7	↑. S y	pProcessInfo = 0018FDA7

Figure 20. The call to `CreateProcessA` at the end of the second PoC. This function executes `shellcodeless.exe`, which was previously downloaded via `URLDownloadToFileA`.

The validation efforts demonstrate convincingly that if a binary is vulnerable and susceptible to supporting a JOP exploit, that instead of merely bypassing DEP, we could then instead avoid the need for doing so, allowing for much more advanced functionality. Under the right circumstances, a shellcodeless JOP attack could be highly effective.

## 6. Final Remarks

The novel shellcodeless JOP technique has the ability to equal what can be done in traditional shellcode. If the `LoadLibrary` and `GetProcAddress` functions are available and not restricted, shellcodeless JOP attacks immediately become flexible and powerful, essentially allowing for the construction of any payload possible via the Windows API. At some point the size of the payload could become an issue for very large shellcodeless JOP attacks, but in other cases, the size provided would be sufficient for very elaborate payloads.

While shellcodeless JOP faces many of the same development issues as other JOP exploits, these are not necessarily any more difficult to overcome as other JOP exploits perform API calls in a similar fashion. The fact that shellcodeless JOP exploits are more likely to contain more frequent and complex API calls brings some additional challenges as well, demanding a greater familiarity with WinAPI functions, shellcode, and flexibility with JOP. Many of these problems are easily solved, such as when API calls break JOP control flow registers. Multiple solutions exist when faced with the problem of retrieving function addresses.

Since it does not contain any arbitrary shellcode, shellcodeless JOP also inherently bypasses types of detection looking for common shellcode patterns. Other mitigations that focus on the unique control flow style of JOP still could affect shellcodeless JOP, although some such as Control Flow Guard may still be bypassed or partly bypassed in some scenarios. That is again, even if the binary is compiled to support it, or the user is not an older operating system; in those cases, no bypass, partial or otherwise, would be necessary.

Vulnerable binaries susceptible to JOP attacks that can allow for a series of multiple stack pivots with no bad byte restrictions could make it extremely simple for an skilled attacker to construct a shellcodeless JOP attack. In that case, it would be considerable easier than doing so with ROP. With ROP, we are reliant upon techniques, such as loading function parameters into a function and then calling the vulnerable WinAPI function via the use of a *pushad* gadget. In short, that is considerably more work,

even under the most ideal circumstances, for one API. With JOP, under ideal circumstances, the same work can be completed using relatively little space. That is, much of it could be supplied directly in the payload, with a smaller number of JOP gadgets used to facilitate the process. While JOP may seem the better candidate, it is also important to bear in mind not every vulnerable binary will be susceptible to JOP. After all, it is still necessary that a viable dispatcher gadget exist. And, while we have expanded the capabilities of JOP, by introducing the two-gadget dispatcher and several other variant dispatcher gadgets[12]. These allow for vastly more gadgets to be feasible as dispatcher gadgets, however, that still may not be sufficient for every binary to support a JOP exploit. Thus, we then return to the problem of JOP's reduced attack surface. ROP, on the other hand, has a much larger attack surface.

With JOP ROCKET [9, 10], we have the ability to easily find each and every possible JOP gadget, allowing for skilled exploit authors to move beyond the limitations of ROP. This tool would readily support creating a shellcodeless JOP exploit. Absent such a tool, creating a non-trivial shellcodeless JOP attack would be a monumentally difficult task, requiring considerable work that would amount to creating a similar tool. Shellcodeless JOP, if nothing else, showcases the relevance and usefulness of such a tool.

JOP has been seldom done in the wild. In fact, there were reports it had never been done in the wild before. At least in a Windows environment, its use has been extremely limited, with only a handful of academic articles discussing it. These left more open questions than answers in terms of actual practical implementation, and no working code was provided. It is our hope that with work such as with shellcodeless JOP, that we will see others begin to use JOP for legitimate, red-team purposes or in competitions. The novelty of our shellcodeless JOP attack can also be clear in the absence of other work in this area.

We refer interested readers to our @Hack Saudi Arabia 2021 Briefings presentation, for further information, including live demonstrations.

## 7. References

1. Miller, M.: Understanding Windows Shellcode, <http://hick.org/code/skape/papers/win32-shellcode.pdf>
2. Cooke, B.: CloudMe 1.11.2 - Buffer Overflow ROP (DEP,ASLR), <https://www.exploit-db.com/exploits/48840>
3. Checkoway, S., Davi, L., Dmitrienko, A., Sadeghi, A.R., Shacham, H., Winandy, M.: Return-oriented programming without returns. *Proc. ACM Conf. Comput. Commun. Secur.* 559–572 (2010). <https://doi.org/10.1145/1866307.1866370>
4. Bletsch, T., Jiang, X., Freeh, V.W.: Proceedings of the 6th International Symposium on Information, Computer and Communications Security, ASIACCS 2011. *Proc. 6th Int. Symp. Information, Comput. Commun. Secur. ASIACCS 2011.* (2011)
5. Specter: Sony Playstation 4 (PS4) 5.05 - BPF Double Free Kernel Exploit Writeup, <https://www.exploit-db.com/exploits/45045>
6. m00nbsd: PoC/CVE-2020-7460/, <https://github.com/thezdi/PoC/tree/master/CVE-2020-7460>
7. M00nbsd: CVE-2020-7460: FreeBSD Kernel Privilege Escalation, <https://www.zerodayinitiative.com/blog/2020/9/1/cve-2020-7460-freebsd-kernel-privilege-escalation>
8. Sadeghi, A., Niksefat, S., Rostamipour, M.: Pure-Call Oriented Programming (PCOP): chaining the gadgets using call instructions. *J. Comput. Virol. Hacking Tech.* 14, 139–156 (2018)
9. Brizendine, B.J.: Advanced Code-reuse Attacks : A Novel Framework for JOP, (2019)
10. Brizendine, B., Stroschein, J.: A JOP Gadget Discovery and Analysis Tool. *S. D. Law Rev.* 65, (2020)
11. Brizendine, B., Babcock, A.: Pre-built JOP Chains with the JOP ROCKET: Bypassing DEP without ROP.
12. Brizendine, B., Babcock, A.: A Novel Method for the Automatic Generation of JOP Chain Exploits. In: *National Cyber Summit.* pp. 77–92 (2021)
13. Min, J.W., Jung, S.M., Lee, D.Y., Chung, T.M.: Jump oriented programming on windows platform (on the x86). *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics).* 7335 LNCS, 376–390 (2012). [https://doi.org/10.1007/978-3-642-31137-6\\_29](https://doi.org/10.1007/978-3-642-31137-6_29)
14. Brizendine, B.: JOP ROCKET repository, [https://github.com/Bw3ll/JOP\\_ROCKET/](https://github.com/Bw3ll/JOP_ROCKET/)
15. Orr, F.: Microsoft Internet Explorer 11 32-bit - Use-After-Free, <https://www.exploit-db.com/exploits/49541>