Evolving Tactics of SLOW#TEMPEST: A Deep Dive Into Advanced Malware Techniques

By Mark Lim

Published: 2025-07-11 · Archived: 2026-04-06 02:51:09 UTC

Executive Summary

In late 2024, we discovered a malware variant related to the SLOW#TEMPEST campaign. In this research article, we explore the obfuscation techniques employed by the malware authors. We deep dive into these malware samples and highlight methods and code that can be used to detect and defeat the obfuscation techniques.

Understanding these evolving tactics is essential for security practitioners to develop robust detection rules and strengthen defenses against increasingly sophisticated threats.

We focus on the following techniques used by the threat actors for the SLOW#TEMPEST campaign:

- Control flow graph (CFG) obfuscation using dynamic jumps
- Obfuscated function calls

Palo Alto Networks customers are better protected from the threats discussed in this article through the following products and services:

- Advanced WildFire
- Cortex XDR and XSIAM

If you think you might have been compromised or have an urgent matter, contact the Unit 42 Incident Response team.

Related Unit 42 Topics	**Anti-analysis, DLL sideloading**

Background

In this article, we analyze a more recent variant of the malware sample (SHA256 hash: a05882750f7caac48a5b5ddf4a1392aa704e6e584699fe915c6766306dae72cc) from the SLOW#TEMPEST campaign. The attackers distribute the malware as an ISO file, which is a common technique used to bundle multiple files to potentially evade initial detection. This ISO file contains 11 files; two are malicious, and the remainder are benign.

The first malicious file is zlibwapi.dll, which we'll refer to as loader DLL, decrypts and executes the embedded payload. The payload is not integrated into the loader DLL in a typical manner. Instead, it is appended to the end of another DLL named ipc_core.dll.

The loader DLL is executed via DLL side-loading by a legitimate signed binary named DingTalk.exe. DLL side-loading is a technique where attackers use a legitimate program to load a malicious DLL file, causing the legitimate program to execute the attacker's code. Separating the payload from the loader DLL complicates detection, as the malicious code will only execute if both the loader and payload binaries are present.

In the following sections, we dive deeper into the anti-analysis techniques used by the malware authors to obfuscate the code in the loader DLL.

Control Flow Graph (CFG) Obfuscation Using Dynamic Jumps

CFG obfuscation alters the execution order of program instructions, making static and dynamic analysis more difficult. This makes it harder to understand the program’s logic, identify malicious functionality and create effective signature-based detection.

For static analysis, traditional tools relying on linear sequences or predictable control flow become ineffective. Dynamic analysis also becomes more challenging because misleading execution paths obscure actual malicious operations. CFG obfuscation breaks the mapping between the original source or compiled code and runtime execution, making it significantly more difficult to create reliable detection rules.

To demonstrate this technique, we analyze the application of CFG obfuscation, specifically using dynamic jumps to the loader DLL's main function. Figure 1 illustrates the CFGs of this function, both with and without obfuscation. Once we remove the obfuscation, the continuous code flow becomes apparent, marked by two colored lines:

- Green for True branches
- Red for False branches

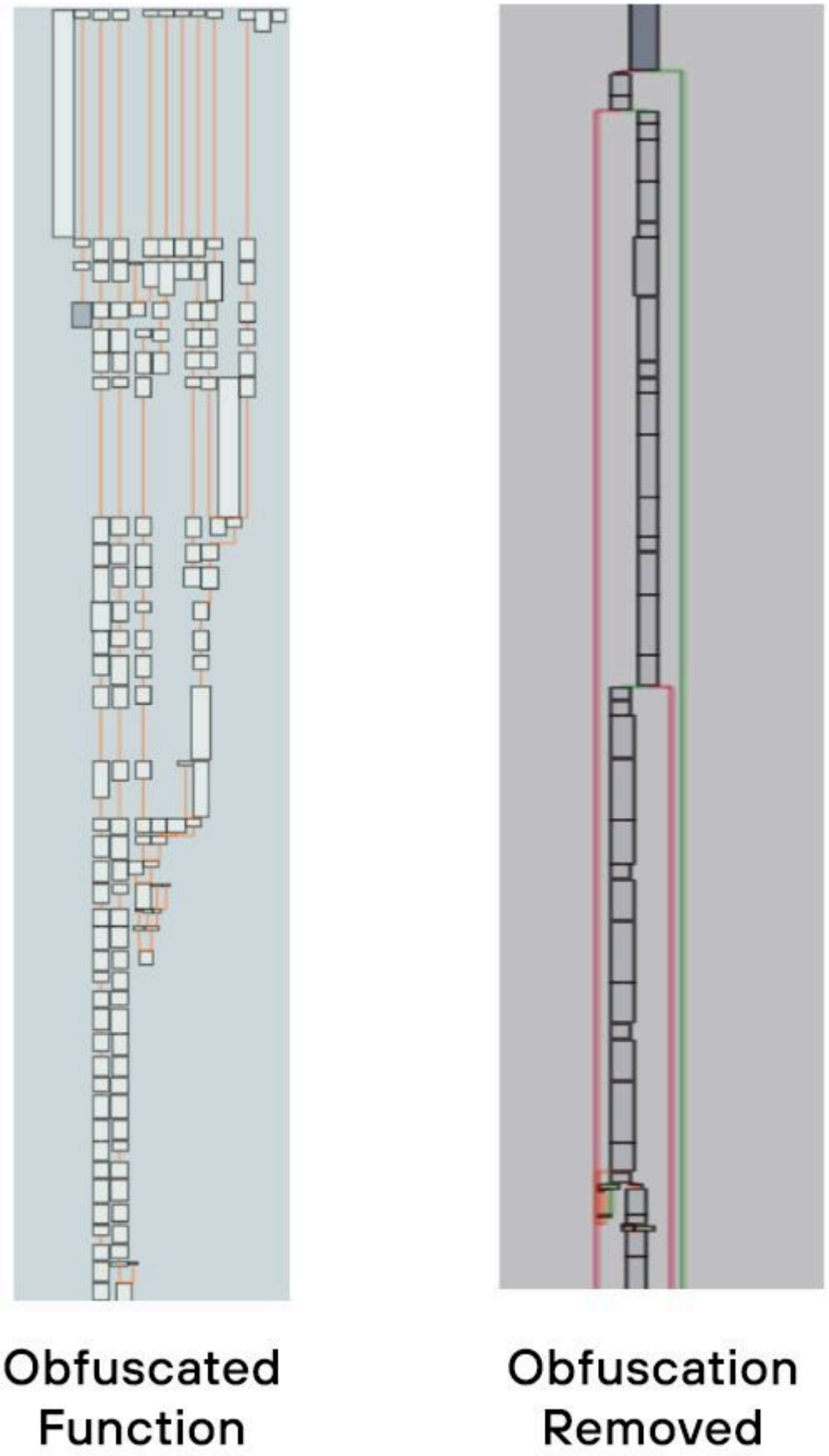

Figure 1. CFGs of the main function for the loader DLL with and without obfuscation.

This function is extensive, comprising over 17,000 lines of assembly instructions. Given the size of the main function for the loader DLL, we turned to the Hex-Rays decompiler to speed up analysis. However, the Hex-Rays decompiler was only able to generate 10 lines of pseudocode for the same main function, as shown in Figure 2.

```
1 // Hidden C++ exception states: #wind=34
2 void __fastcall sub_180009800(__int64 a1, int a2)
3 {
4   __int64 v2; // rax
5
6   v2 = 8;
7   if ( a2 == 1 )
8     v2 = 88;
9   __asm { jmp     rax }
10 }
```

Figure 2. Pseudocode generated by Hex-Rays of the main function for the loader DLL.

The reason for the incomplete Hex-Rays decompiler output is because the sample used dynamic jump instructions. Dynamic jumps are where target addresses in code are computed at runtime.

Unlike direct jumps to fixed addresses, dynamic jumps make it impossible for the decompiler to determine the execution flow without actually running the program. This lack of a clear, predetermined path severely hinders the decompiler's ability to reconstruct the original high-level code, often leading to incomplete or inaccurate decompilation results.

Figure 3 shows one of the dynamic jumps using the JMP RAX instruction near the entry point of the main function of the loader DLL. The JMP instruction will cause the execution flow to be diverted to a different target address.

The target address depends on factors like memory contents, register values and the results of conditional checks performed during execution. In this case, it is computed at runtime by a sequence of preceding instructions and stored in the CPU register RAX.

```
0000000180009821    mov     [rbp+1170h+var_40], 0FFFFFFFFFFFFFFFEh
000000018000982C    mov     [rbp+1170h+var_50], r8
0000000180009833    mov     [rbp+1170h+var_54], edx
0000000180009839    mov     [rbp+1170h+var_60], rcx
0000000180009840    mov     eax, [rbp+1170h+var_54]
0000000180009846    sub     eax, 1
0000000180009849    mov     ecx, 88
000000018000984E    mov     eax, 8
0000000180009853    cmovz   rax, rcx
0000000180009857    mov     rcx, cs:off_18019BFD0
000000018000985E    add     rcx, rax
0000000180009861    mov     rax, 0F5846A57F5E942D3h
000000018000986B    mov     rax, [rax+rcx]
000000018000986F    mov     rcx, 0C8B25E2735BA799Ah
0000000180009879    add     rax, rcx
000000018000987C    jmp     rax
000000018000987E ; ---------------------------------------------------------------------------
000000018000987E    mov     rax, cs:off_18019A2D0
0000000180009885    mov     rcx, 51EB356284032967h
000000018000988F    add     rax, rcx
0000000180009892    lea     rcx, [rbp+1170h+var_C8]
```

Figure 3. One dynamic jump in the main function for the loader DLL.

We countered CFG obfuscation employing dynamic jumps by first identifying all instances of these jumps using the IDAPython script shown in Figure 4.

```
idaapi.msg_clear()
ea = idaapi.get_screen_ea()
func_obj = idaapi.get_func(ea)

start_ea = func_obj.start_ea
end_ea = func_obj.end_ea

ea = start_ea
while ea < end_ea:
    ea = idaapi.next_head(ea)
    if not idc.is_code(idc.get_full_flags(ea)):
        continue
    insn = idaapi.insn_t()
    length = idaapi.decode_insn(insn, ea)
    if length == 0:
        continue
    if insn.itype >= idaapi.NN_ja and insn.itype <= idaapi.NN_jmpshort and insn.itype != idaapi.NN_jmp:
        disasm_line = idc.generate_disasm_line(ea,0)
        print(f'0x{ea:x}',disasm_line)
```

Figure 4. Code to locate dynamic jumps.

Using the above script, we identified 10 dynamic jumps in the main function of the loader DLL.

The dynamic jump target is determined by a preceding sequence of nine instructions, termed a “dispatcher,” before each JMP RAX instruction. These 10 dispatchers, found within the loader DLL's main function, share a similar structure. However, each dispatcher uses a distinct set of instructions to compute the jump's destination address, effectively hiding the program's control flow.

Imagine the program is a complex dance routine. Normally, the dancers move predictably from one step to the next. However, in this case, the program has hidden "jump points." Before each jump, there's a mini-routine, like a secret handshake, that decides exactly where the next jump will land. These secret handshakes are all a bit

different, making it very hard to predict the dance's true path, almost like the dancers are improvising where they'll go next, even though it's all pre-programmed.

Each dispatcher implements a two-way branching mechanism. The code path taken depends on the state of the Zero Flag (ZF) or the carry flag (CF) when the dispatcher is entered. These flags, which are set by previous instructions to indicate the result of an operation (e.g., zero or overflow), determine which branch is taken. Each dispatcher has a pair of conditional move (CMOVNZ) or set (SETNL) instructions and an indirect jump (JMP RAX). This creates a dynamic control flow that depends on runtime conditions and memory contents, making static analysis difficult.

ZF and CF are CPU status flags that reflect arithmetic and logical operation outcomes. These flags act as internal switches, enabling dynamic program execution based on prior computation results. Each conditional move or set instruction has two possible target addresses: one for a true condition and the other for a false condition. For example, the conditional move if not zero (CMOVNZ) instruction will only move data if the ZF is 0, indicating that the previous operation did not result in 0. If the ZF is 1 (meaning the previous result was 0), the CMOVNZ instruction will not move the data, and execution will continue to a different target address.

Figure 5 shows one of the dispatchers. We annotated the instructions to explain how the destination addresses are computed.

```
.text:000000018000B83C   mov     ecx, 18h                    ; Load 0x18 into ecx
.text:000000018000B841   mov     eax, 48h                    ; Load 0x48 into eax
.text:000000018000B846   cmovnz  rax, rcx                    ; Conditional move: if ZF=0, rax = rcx (0x18)
.text:000000018000B84A   mov     rcx, cs:off_18019BFD0       ; Load value from offset into rcx  [off_18019BFD0] = 0xA7B95A98A30799D
.text:000000018000B851   add     rcx, rax                    ; Add selected offset (0x18 or 0x48)
.text:000000018000B854   mov     rax, 0F5846A57F5E942D3h     ; Load constant into rax
.text:000000018000B85E   mov     rax, [rax+rcx]              ; Dereference address computed by rax+rcx
.text:000000018000B862   mov     rcx, 0C8B25E2735BA799Ah     ; Load constant into rcx
.text:000000018000B86C   add     rax, rcx                    ; Add constant to produce final jump address
.text:000000018000B86F   jmp     rax                         ; Jump to computed address
```

Figure 5. CPU instructions of one dispatcher.

Using Unicorn — a multi-platform, multi-architecture CPU emulator framework — automates the identification of destination jump addresses. We achieve this by executing the nine instructions preceding each JMP RAX in a controlled manner, rather than running the entire binary. This allows us to determine the jump addresses for each dispatcher.

To extract the bytecodes of these instructions, we use the code shown in Figure 6.

```
def setup_emulate(JMP_RAX):
    emu_start = JMP_RAX
    loop = 9
    for i in range(loop):
        emu_start = idc.prev_head(emu_start)    #looking for the entry of the dispatcher
    CODE_HEX_len = JMP_RAX - emu_start
    CODE_HEX = idaapi.get_bytes(emu_start, CODE_HEX_len).hex()
    return (CODE_HEX, emu_start)
```

Figure 6. Code to extract the bytecodes of the dispatchers.

Next, we emulate each dispatcher. Since each dispatcher uses a two-way branching mechanism with two target addresses, the emulation process must be repeated twice for each dispatcher to determine both destination

addresses. Figure 7 shows the code used to emulate the dispatchers.

```
def emulate_code(CODE_HEX, ADDRESS):
    CODE = bytes.fromhex(CODE_HEX)
    TEXT_SEG = 0x180001000

    #read bytes from .data segment
    data_segm = idaapi.get_segm_by_name(".data")
    DATA_SEGMENT = idaapi.get_bytes(data_segm.start_ea, data_segm.end_ea - data_segm.start_ea).hex()

    # Compute destination address when CF and ZF are 1s
    mu = Uc(UC_ARCH_X86, UC_MODE_64)
    mu.mem_map(TEXT_SEG, 0x200000)
    mu.mem_write(ADDRESS, CODE)
    eflags = mu.reg_read(UC_X86_REG_EFLAGS)
    eflags |= 0x1
    eflags |= 0x40
    mu.reg_write(UC_X86_REG_EFLAGS, eflags)
    mu.mem_write(0x18019A000,bytes.fromhex(DATA_SEGMENT)) #create .data segment
    mu.reg_write(UC_X86_REG_RAX, 0)
    # Emulate code
    mu.emu_start(ADDRESS, ADDRESS + len(CODE))
    # Read back results
    rax_1 = mu.reg_read(UC_X86_REG_RAX)

    # Compute destination address when CF and ZF are 0s
    mu = Uc(UC_ARCH_X86, UC_MODE_64)
    mu.mem_map(TEXT_SEG, 0x200000)
    mu.mem_write(ADDRESS, CODE)
    eflags = mu.reg_read(UC_X86_REG_EFLAGS)
    eflags &= ~0x1
    eflags &= ~0x40
    mu.reg_write(UC_X86_REG_EFLAGS, eflags)
    mu.mem_write(0x18019A000,bytes.fromhex(DATA_SEGMENT))
    mu.reg_write(UC_X86_REG_RAX, 0)
    # Emulate code
    mu.emu_start(ADDRESS, ADDRESS + len(CODE))
    # Read back results
    rax_0 = mu.reg_read(UC_X86_REG_RAX)

    return (rax_0, rax_1)
```

Figure 7. Code to emulate the dispatcher to determine the destination addresses.

After computing the two destination addresses, we replaced the dispatcher instructions with direct jump instructions to those addresses, effectively removing the CFG obfuscation. This allowed us to see the original code flow easily in IDA Pro. Figure 8 shows the code to patch the instructions in the IDA database.

```
def jmp_rax_patch(emu_start, jmp_ea, tar_0_ea, tar_1_ea):
    offset0 = tar_0_ea - emu_start - 6
    print(f"{emu_start:X} offset = {offset0}")
    offset0 = offset0.to_bytes(4,'little')
    jnz_buf = "0F85" + offset0.hex()
    jnz_buf = bytes.fromhex(jnz_buf)
    idaapi.patch_bytes(emu_start,jnz_buf)
    cur_ea = emu_start + len(jnz_buf)
    offset1 = tar_1_ea - cur_ea - 5
    print(f"{cur_ea:X} offset = {offset1}")
    offset1 = offset1.to_bytes(4,'little')
    jmp_buf = "E9" + offset1.hex()
    jmp_buf = bytes.fromhex(jmp_buf)
    idaapi.patch_bytes(cur_ea,jmp_buf)
    cur_ea = cur_ea + len(jmp_buf)
    while cur_ea != jmp_ea + 2:
        idaapi.patch_byte(cur_ea,0x90)
        cur_ea = idc.next_addr(cur_ea)
```

Figure 8. Code to patch the dispatchers with de-obfuscated jump instructions.

Finally, we forced IDA Pro to re-analyze the entire function that was patched using the code shown in Figure 9. This was to trigger IDA Pro to update its CFG based on the de-obfuscated instructions.

```
def fix_function(start, end):
    ea = start
    while ea < end:
        ida_bytes.del_items(ea, ida_bytes.DELIT_SIMPLE)
        ea = idc.next_addr(ea)
    ea = start
    while ea < end:
        len = idaapi.create_insn(ea)
        ea = len + ea
    ida_funcs.add_func(start, end)
```

Figure 9. Code to force IDA Pro to re-analyze the patched functions.

The complete script for resolving with CFG obfuscation using dynamic jumps is available at emu_jmp_rax_idapython.py.

After executing this script, the Hex-Rays decompiler successfully decompiled the main function within the loader DLL. Figure 10 shows part of the decompilation output.

```
if ( a2 != 1 )
  return 1;
((void (__fastcall *)(_BYTE *))((char *)off_18019A2D0 + 0x51EB356284032967LL))(v308);
sub_180008490(v307);
sub_180008490(v306);
sub_180008490(v305);
LOBYTE(v3) = 0;
memset(v304, v3, 0x104u);
if ( ((unsigned int (__fastcall *)(__int64, _BYTE *, __int64))((char *)off_18019A2D8 + 0x51EB356284032967LL))(
       v309,
       v304,
       260) )
{
  ((void (__fastcall *)(void *, char *))((char *)off_18019A2F0 + 0x51EB356284032967LL))(
    &unk_18019E794,
    (char *)off_18019BFD8 - 0xA7B95A80A16ADE8LL);
  sub_180002050(v301, (char *)off_18019BFE8 - 0xA7B95A80A16BD2DLL);
  ((void (__fastcall *)(void *, char *))((char *)off_18019A2F8 + 0x51EB356284032967LL))(
    &unk_18019E7B8,
    (char *)off_18019BFD8 - 0xA7B95A80A16AD91LL);
  sub_180002050(v300, (char *)off_18019BFF0 - 0xA7B95A80A16BD2DLL);
  ((void (__fastcall *)(void *, char *))((char *)off_18019A300 + 0x51EB356284032967LL))(
    &unk_18019E7E4,
    (char *)off_18019BFD8 - 0xA7B95A80A16AD39LL);
  sub_180002050(v299, (char *)off_18019BFF8 - 0xA7B95A80A16BD2DLL);
  v302 = sub_180007630((unsigned int)v308, (unsigned int)v299, (unsigned int)v300, 1, (__int64)v301);
  ((void (__fastcall *)(void *, char *))((char *)off_18019A308 + 0x51EB356284032967LL))(
    &unk_18019E80C,
    (char *)off_18019BFD8 - 0xA7B95A80A16ACE3LL);
  sub_180002050(v298, (char *)off_18019C000 - 0xA7B95A80A16BD2DLL);
  ((void (__fastcall *)(void *, char *))((char *)off_18019A310 + 0x51EB356284032967LL))(
    &unk_18019E818,
    (char *)off_18019BFD8 - 0xA7B95A80A16ACAELL);
  sub_180002050(v297, (char *)off_18019C008 - 0xA7B95A80A16BD2DLL);
```

Figure 10. Decompiled output of the main function in the loader DLL.

However, we observed that further obfuscation remained in the code. Specifically, most functions were called dynamically, and we did not observe any direct Windows API calls. This made it challenging to immediately discern the code's purpose, as the actual functionality was obscured by indirect function resolution.

Obfuscated Function Calls

Obfuscated function calls use indirect calls, where the function's address is calculated dynamically at runtime and then called through a pointer, instead of directly invoking the function by its name. Attackers use this technique to hinder static analysis, as the actual target function is not immediately apparent in the code. This makes it more difficult to understand the program's behavior and identify malicious actions.

Analysis of the main function's assembly code reveals the presence of multiple obfuscated function calls. The Call RAX instruction is a key indicator, as it signifies that the function address is being dynamically determined at runtime rather than being directly specified in the code. Similar to dynamic jumps, the target addresses of these function calls were calculated at runtime. We were not able to determine the target addresses without executing the binary. Figure 11 shows some of the obfuscated function calls.

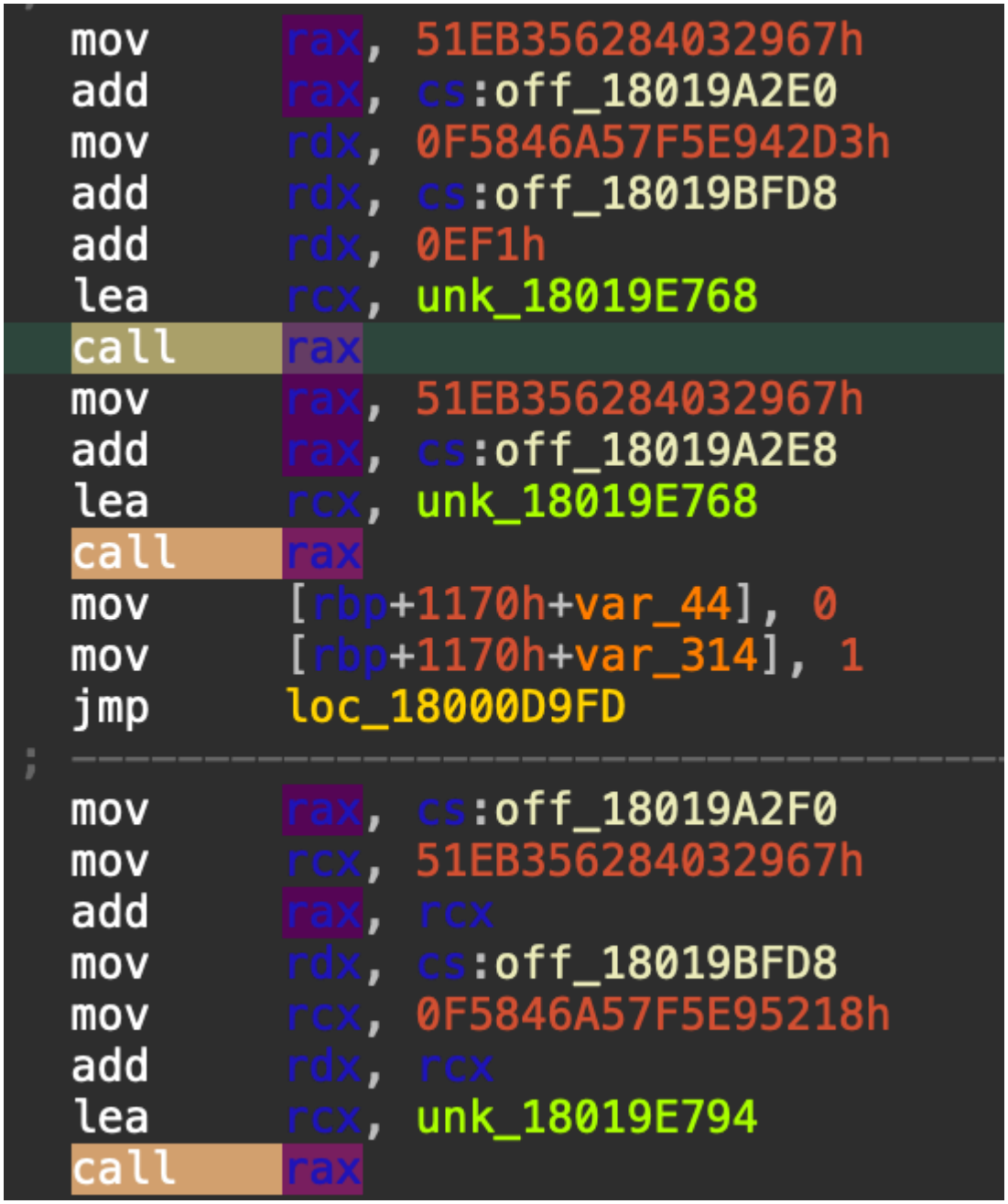

Figure 11. Instructions of multiple obfuscated function calls.

To determine the target address of obfuscated function calls, we applied a similar approach to the one we used for dynamic jumps. This is because both techniques involve calculating target addresses at runtime.

Our script successfully calculated the destination addresses of these obfuscated function calls. However, we observed that IDA Pro failed to identify the arguments of standard Windows APIs even though the destination addresses were correctly resolved, as Figure 12 shows. This is because there was missing function signature information linking the addresses of the Windows APIs with the obfuscated function calls.

```
mov     rax, cs:off_18019A4A0
add     rax, rcx
mov     rcx, rsp
lea     rdx, [rbp+1170h+var_F74]
mov     [rcx+28h], rdx
mov     dword ptr [rcx+20h], 0
lea     r8, sub_180009420
mov     rcx, r9
mov     rdx, r9
call    rax             ; CreateThread
mov     rcx, [rbp+1170h+var_1188]
mov     rdx, [rbp+1170h+var_1178]
mov     [rbp+1170h+var_F70], rax
mov     rax, cs:off_18019A4A8
add     rax, rdx
call    rax             ; GetModuleHandleA
mov     rcx, [rbp+1170h+var_1178]
mov     [rbp+1170h+var_1180], rax
mov     rax, cs:off_18019A4B0
add     rax, rcx
call    rax             ; GetCurrentProcess
mov     rdx, [rbp+1170h+var_1180]
mov     r8, [rbp+1170h+var_1178]
mov     rcx, rax
mov     rax, cs:off_18019A4B8
add     rax, r8
lea     r8, [rbp+1170h+var_F90]
mov     r9d, 18h
call    rax             ; K32GetModuleInformation
```

Figure 12. Destination addresses of the obfuscated function calls resolved.

To enable IDA Pro to correctly identify the function arguments, rename local variables and perform proper analysis, we needed to explicitly set the “callee” address for each obfuscated function call using the code shown in Figure 13. This provides IDA Pro with the necessary information to recognize the function as a known Windows API.

```python
def set_callee_address(call_ea, callee_ea):
    # Use Intel-specific netnode name
    nname = "$ vmm functions"
    n = ida_netnode.netnode(nname)
    # Store callee EA directly (workaround for missing ea2node)
    n.altset_ea(call_ea, callee_ea+1)
    # Reanalyze the instruction to update internal metadata
    ida_auto.plan_ea(call_ea)
    # Add a user cross-reference
    ida_xref.add_cref(call_ea, callee_ea, ida_xref.fl_CN | ida_xref.XREF_USER)
    # Refresh IDA views correctly
    ida_kernwin.refresh_idaview_anyway()
    return True
```

Figure 13. Code to set the “callee” address to each obfuscated function call.

After adding the code shown in Figure 13 to set the callee address, IDA Pro will automatically label function arguments and rename local variables for each obfuscated function call. This significantly improved our ability to read and analyze the code, allowing us to understand the function's purpose more easily. Figure 14 shows some of the function arguments that IDA Pro labeled.

```
xor     edx, edx        ; dwStackSize
mov     ecx, edx        ; lpThreadAttributes
mov     [rbp+1170h+dwStackSize], rcx
call    rax ; sub_180167CA0 ; sub_180167CA0
mov     r9, [rbp+1170h+dwStackSize] ; lpParameter
mov     rcx, [rbp+1170h+var_1178]
mov     [rbp+1170h+var_F68], rax
mov     rax, cs:off_18019A4A0
add     rax, rcx
mov     rcx, rsp
lea     rdx, [rbp+1170h+var_F74]
mov     [rcx+28h], rdx
mov     dword ptr [rcx+20h], 0
lea     r8, sub_180009420 ; lpStartAddress
mov     rcx, r9         ; lpThreadAttributes
mov     rdx, r9         ; dwStackSize
call    rax ; CreateThread ; CreateThread
mov     rcx, [rbp+1170h+dwStackSize] ; lpModuleName
mov     rdx, [rbp+1170h+var_1178]
mov     [rbp+1170h+var_F70], rax
mov     rax, cs:off_18019A4A8
add     rax, rdx
call    rax ; GetModuleHandleA ; GetModuleHandleA
mov     rcx, [rbp+1170h+var_1178]
mov     [rbp+1170h+var_1180], rax
mov     rax, cs:off_18019A4B0
add     rax, rcx
call    rax ; GetCurrentProcess ; GetCurrentProcess
mov     rdx, [rbp+1170h+var_1180] ; hModule
mov     r8, [rbp+1170h+var_1178]
mov     rcx, rax        ; hProcess
mov     rax, cs:off_18019A4B8
add     rax, r8
lea     r8, [rbp+1170h+modinfo] ; lpmodinfo
mov     r9d, 18h        ; cb
call    rax ; K32GetModuleInformation ; K32GetModuleInformation
```

Figure 14. Function arguments and local variables renamed for the obfuscated function calls.

The complete script for resolving obfuscated function calls is at emu_call_rax_idapython.py.

After executing this script, we successfully de-obfuscated both the control flow and the function calls within the loader DLL. With the code now significantly more readable and the Windows API calls properly identified, we could proceed with analyzing its core functionality. In the final section, we examine the main purpose of the loader DLL.

Loader DLL Analysis

After removing the obfuscation using the scripts emu_call_rax_idapython.py and emu_call_rax_idapython.py, we easily located the main functionality of the loader DLL.

First, we observed an anti-sandbox check that uses the Windows API GlobalMemoryStatusEx to determine the total physical memory available on the system. The loader DLL will only unpack its payload and execute it in

memory if the target machine has at least 6 GB of RAM. Figure 15 shows the pseudocode of the core components of the loader DLL.

```
Buffer.dwLength = 64;
((void (__stdcall *)(LPMEMORYSTATUSEX))((char *)off_18019A330 + 0x51EB356284032967LL))(&Buffer);// GlobalMemoryStatusEx
v292 = Buffer.ullTotalPhys >> 30;
if ( v292 >= 6 )
{

    ..... //code removed for clarity of presentation

    ((void (__stdcall *)(HANDLE, LPVOID, SIZE_T, DWORD, PDWORD))((char *)off_18019A4D0 + 0x51EB356284032967LL))(
                hProcess,
                modinfo.EntryPoint,
                v167,
                0x40u,
                &flOldProtect);                 // VirtualProtectEx
                nSize = ((__int64 (__fastcall *)(_BYTE *))((char *)off_18019A4C8 + 0x51EB356284032967LL))(v186);
                v168 = (LPCVOID)((__int64 (__fastcall *)(_BYTE *))((char *)off_18019A4D8 + 0x51EB356284032967LL))(v186);// sub_180016100
                ((void (__stdcall *)(HANDLE, LPVOID, LPCVOID, SIZE_T, SIZE_T *))((char *)off_18019A4E0 + 0x51EB356284032967LL))(
                hProcess,
                modinfo.EntryPoint,
                v168,
                nSize,
                &NumberOfBytesWritten);          // WriteProcessMemory
                flNewProtect = flOldProtect;
                v169 = ((__int64 (__fastcall *)(_BYTE *))((char *)off_18019A4C8 + 0x51EB356284032967LL))(v186);// sub_180015D70
                ((void (__stdcall *)(HANDLE, LPVOID, SIZE_T, DWORD, PDWORD))((char *)off_18019A4D0 + 0x51EB356284032967LL))(
                hProcess,
                modinfo.EntryPoint,
                v169,
                flNewProtect,
                &flOldProtect);                 // VirtualProtectEx
```

Figure 15. Pseudocode of the core components of the loader DLL.

Conclusion

The SLOW#TEMPEST campaign's evolution highlights malware obfuscation techniques, specifically dynamic jumps and obfuscated function calls. This illustrates the importance for security practitioners to adopt advanced dynamic analysis techniques (e.g., emulation) alongside static analysis to effectively dissect and understand modern malware.

The success of the SLOW#TEMPEST campaign using these techniques demonstrates the potential impact of advanced obfuscation on organizations, making detection and mitigation significantly more challenging. Understanding how threat actors leverage these methods is crucial for developing robust detection rules and strengthening defenses against increasingly complex threats.

Palo Alto Networks customers are better protected from the threats discussed above through the following products:

- Advanced WildFire can detect the malware samples discussed in this article.
- Cortex XDR and XSIAM are designed to prevent the execution of known malicious malware, and also prevent the execution of unknown malware using Behavioral Threat Protection and machine learning based on the Local Analysis module. The Cortex Shellcode AI module can help detect and prevent shellcode attacks.

If you think you may have been compromised or have an urgent matter, get in touch with the Unit 42 Incident Response team or call:

- North America: Toll Free: +1 (866) 486-4842 (866.4.UNIT42)
- UK: +44.20.3743.3660

- Europe and Middle East: +31.20.299.3130
- Asia: +65.6983.8730
- Japan: +81.50.1790.0200
- Australia: +61.2.4062.7950
- India: 00080005045107

Palo Alto Networks has shared these findings with our fellow Cyber Threat Alliance (CTA) members. CTA members use this intelligence to rapidly deploy protections to their customers and to systematically disrupt malicious cyber actors. Learn more about the Cyber Threat Alliance.

Indicators of Compromise

- SHA256 hash: a05882750f7caac48a5b5ddf4a1392aa704e6e584699fe915c6766306dae72cc
- File size: 7.42 MB
- File description: ISO file distributed in the SLOW#TEMPEST campaign
- SHA256 hash: 3d3837eb69c3b072fdfc915468cbc8a83bb0db7babd5f7863bdf81213045023c
- File size: 1.64 MB
- File description: DLL used to load and execute the payload
- SHA256 hash: 3583cc881cb077f97422b9729075c9465f0f8f94647b746ee7fa049c4970a978
- File size: 1.64 MB
- File description: DLL with encrypted payload in the overlay segment

---