Unpacking the packer ‘pkr_mtsi’

By Robert SimmonsRobert Simmons

Published: 2026-01-06 · Archived: 2026-04-06 00:49:16 UTC

This blog post presents an in-depth technical analysis of *pkr_mtsi*, a malicious Windows packer first observed in the wild on April 24, 2025, and continuously deployed through the time of writing. The packer is actively leveraged in large-scale malvertising and SEO-poisoning campaigns to distribute trojanized installers for legitimate software, enabling initial access and flexible delivery of follow-on payloads. In observed campaigns, *pkr_mtsi* has been used to deliver a diverse set of malware families, including Oyster, Vidar, Vanguard Stealer, Supper, and more, underscoring its role as a general-purpose loader rather than a single-payload wrapper.

This analysis highlights *pkr_mtsi*’s evolution across campaigns over the past eight months, including the use of increasingly sophisticated obfuscation, anti-analysis techniques, and evasive API resolution strategies. Despite this evolution, the *pkr_mtsi* packer retains consistent structural and behavioral characteristics that enable reliable identification, behavioral detection, and signature development when analyzed holistically. This report discusses those characteristics — and provides a YARA rule to detect all identified versions of the packer.

Identification Strategies

The *pkr_mtsi* packer is commonly distributed under the guise of a legitimate software installer. It has been observed masquerading as installers for widely used utilities such as PuTTY, Rufus, and Microsoft Teams, among others. Distribution of the malware is not the byproduct of supply chain compromises of the legitimate vendors. Rather, it is typically facilitated by fake software download websites that pose as legitimate sources and achieve prominent placement in search engine results via malvertising campaigns and SEO poisoning techniques.

Antivirus detections of *pkr_mtsi* (that are not purely generic) frequently include the substrings "oyster" or "shellcoderunner." In addition, a single public YARA rule exists that identifies a limited subset of *pkr_mtsi* samples under the name "TextShell." However, this rule does not comprehensively cover all observed variants. To address this gap, a more complete YARA rule that matches all identified samples of this packer is provided in Appendix A. Results of a recent retro hunt in Spectra Analyze using this rule are shown below (Figure 1).

Figure 1: YARA hunting results showing “oyster” and “shellcoderunner” common threat types.

Core Technical Features

Samples attributable to *pkr_mtsi*, share a consistent set of core technical features. The first non-library function invoked by main always allocates a region of memory into which the next execution stage is written. Earlier variants perform this allocation via a direct call to VirtualAlloc, while more recent variants employ an obfuscated call to *ZwAllocateVirtualMemory*.

Following memory allocation, execution proceeds through a sequence of functions responsible for reconstructing the next-stage payload. The payload is divided into chunks ranging from one to eight bytes, stored as immediate values. In later variants, the chunks are passed through a decoding routine. And in all variants, written to specific offsets within the allocated memory region. The unusually large number of these small "loader" instructions is a strong indicator of this packer's presence.

Early variants of *pkr_mtsi* that we analyzed resolved DLLs and API functions from plaintext strings. Later variants have shifted to resolving both DLLs and APIs via hashed identifiers combined with Process Environment Block (PEB) traversal. A further distinguishing feature across variants is the pervasive use of junk calls to GDI API functions, which serve no functional purpose and are intended to frustrate static and behavioral analysis. Together, these characteristics form the foundation for reliable identification and are captured in the detection logic of the YARA rule included at the end of this report (Appendix A).

Identifying Characteristics

The *pkr_mtsi* packer has been observed in both executable (EXE) and dynamic-link library (DLL) forms. While the overall unpacking logic is shared between these formats, the DLL variants support multiple execution contexts. One execution path reliably triggers on DLL load and is responsible for unpacking the next stage and final payload. Additional execution paths may be invoked during DLL unload events, allowing alternative entry points into the unpacked payload to be executed.

In several DLL samples, the packer exports *DllRegisterServer*, enabling the malware to be loaded via *regsvr32.exe*. This provides a convenient mechanism for persistence through registry-based COM registration while leveraging a trusted Windows utility for execution.

The intermediate stage produced by the packer is a modified UPX-packed module. Recent versions of UPX are used, but with selective removal of identifying components to evade detection. Across observed samples, portions of the UPX structure, including headers, magic values, and ancillary metadata, are stripped as long as execution remains viable. This deliberate degradation of the UPX module complicates both static identification and automated unpacking.

In earlier samples, the image bases used were MSVC defaults for PE32+ EXEs: *0x140000000* and DLLs: *0x180000000*. And in later samples, these image bases changed to non-standard, very high numbers such as *0x7ff662c10000*. The precise motivation for this change is unclear. One plausible explanation is that it is an attempt to resemble a system DLL or an ASLR-relocated image even when relocation support is disabled.

Behavioral and Code Analysis Findings

The *pkr_mtsi* packer exhibits a clear evolutionary trajectory across campaigns. Earlier variants preserve more recognizable UPX artifacts and rely on simpler API resolution mechanisms, while later variants remove detectable features and introduce obfuscation layers. These changes indicate an adversary that is actively iterating on the packer to counter detection and analysis.

For example, in earlier *pkr_mtsi* samples, execution begins with a distinctive sequence of adversary-controlled function calls invoked in rapid succession. The first of these functions allocates memory that will hold the

unpacked next-stage payload. Subsequent functions reconstruct the payload by writing small fragments of data into the allocated region. The total number of these payload-writing functions varies according to the size of the embedded payload, but their repetitive structure and volume form a recognizable behavioral pattern.

This execution pattern results in a dense cluster of functions whose sole purpose is to write chunks of data which range in size from one to eight bytes into memory. These chunks are embedded as immediate values within the instruction stream and may be passed through lightweight decoding routines before being written. The high frequency of such write operations, combined with the absence of meaningful control flow or computation within these functions, distinguishes this packer from more conventional loaders.

From a behavioral perspective, the packer's early-stage activity is dominated by memory allocation followed by intensive memory writes to a single contiguous region. This activity occurs prior to any meaningful interaction with the unpacked payload logic. Later variants preserve this overall execution model but introduce additional obfuscation layers intended to frustrate static analysis. Despite these changes, the fundamental behavior of early allocation followed by staged payload reconstruction remains consistent across samples and campaigns.

***Figure 2: first set of functions in main in older vs recent samples of* pkr_mtsi.**

Anti-Debugging Features

RL's analysis found that the *pkr_mtsi* packer employs a number of different anti-analysis features which are highlighted in various locations in the report below according to where the feature is found in the samples. However, anti-debugging is used throughout many variants of this packer and those anti-debugging techniques are described here. Two debugger detection API calls are utilized: *IsDebuggerPresen*t and *CheckRemoteDebuggerPresent*. Many instances of these function calls simply cause the process to exit. However, some instances use a different type of trap after the call that leads to an infinite loop with a jump instruction that leads to itself. This pattern (Figure 3) is easily detected using YARA.

Figure 3: Anti-Debugging trap path to infinite loop shown with red arrow.

Memory Allocation Functions

The *allocate_memory* function in older samples contained an indirect call to *VirtualAlloc* and was not obfuscated. The exact bytes of this function, other than relative displacements and input parameter instruction order, are stable over all observed variants of the packer that call *VirtualAlloc* in the *allocate_memory* function. These byte patterns are used for identification of the variants that contain them. All function names for adversary functions herein are assigned during research and are not necessarily the adversary's name for the functions in their source code.

***Figure 4: Function to allocate memory for next stage that utilizes* VirtualAlloc.**

More recent campaigns reveal changes to the allocate_memory function. Specifically, they use an obfuscated call to *ZwAllocateVirtualMemory* rather than *VirtualAlloc*. The transition to this obfuscated function was observed in samples starting in August 2025. The constant values used as input parameters to *ZwAllocateVirtualMemory* are stable but appear in different orders and interspersed with differing numbers of junk instructions depending on the build. These features are randomized by the packer build process.

***Figure 5: Obfuscated call to* ZwAllocateVirtualMemory *in a more recent sample of* pkr_mtsi.**

Loading The NTDLL Handle

The *pkr_mtsi* packer utilizes three general methods for getting a handle for *ntdll.dll* for later use resolving API functions. The method observed in the earliest samples is the most straightforward, loading a string containing the name of the DLL as an input parameter for a call to *LoadLibraryA*. The first evolution after that replaced the call to *LoadLibraryA* with a custom resolver function (Figure 6). The return value of the function is a pointer to *ntdll.dll* in rax. That pointer is then stored in a global variable for use by other functions later.

***Figure 6: DLL handle resolver function call with return value stored in global:* ntdll_handle.**

The first step in the custom resolver function is to load the pointer to the *Process Environment Block (PEB)* from the *Thread Environment Block (TEB)* located at offset 0x60 in the gs segment. The offset 0x60 is observed in two forms, one obfuscated and one not. The obfuscated variant uses a calculation combined with a series of instructions that is similar to stack strings. However, some of the stack values are written and then immediately stomped on with new values over and over. The last value in the series of stomps is the one used in the subsequent calculation. Figure 7 shows the algorithm input values (highlighted in green and marked with No. 1); the series of bytes that are stomped on one-after-another until the last one which is used in the calculation of the offset in gs (green arrow); two inputs to the calculation (highlighted in yellow and marked with No. 2) where they are moved from the stack to two registers; the subtraction calculation and the result (highlighted in red and marked with No. 3); and the instruction where the pointer to the PEB is loaded from the gs segment (highlighted in blue and marked No. 4).

Figure 7: Obfuscated offset in gs segment using stack string stomping showing algorithm input values (1), inputs to the calculation (2), subtraction calculation and result (3), instruction where the pointer to the PEB is loaded from the gs segment (4).

From the PEB, the *InMemoryOrderModuleList* is walked, comparing each module name in the list to the name of the needed DLL, in this case *ntdll.dll*. This is done in two different ways in different samples: either a direct string comparison or checking against a hash of the DLL name (Figure 8).

The main steps in this process are:

1. Walking the linked list.
2. Copying the DLL name to a location on the stack
3. Calculating the hash of the DLL name (optional)
4. Comparing the result to the hash that was the function input parameter (optional)

In some samples, the second and third steps are separated out into their own dedicated subroutines called from this function. In the step that copies the DLL name to the stack, any non-ASCII character is replaced with a question mark character. This is a feature that makes this algorithm detectable by leaving a 0x3f immediate value in the disassembly that can be used to detect the function containing this step.

Additionally, there is a check for the length of the module name at 64 characters. This is significant in that it is an anti-analysis trick to catch filenames in a malware sandbox that are the *SHA256* of the submitted file. If the

filename is a *SHA256*, the packer exits. This is interesting because this anti-analysis check works for EXEs, but not always for DLLs — making it a strange choice for a malware feature that is found in both DLLs and EXEs. The reason it doesn't always work for DLLs is the DLL's own filename is not located early in the linked list. Rather, it is the module name of the EXE that loaded the DLL. This filename may not be a *SHA256*. It is more likely to be *rundll32.exe* or *regsvr32.exe*.

Figure 8: DLL handle resolver algorithm steps.

Resolving API Function Pointers

There are two general locations in *pkr_mtsi* where function pointers are resolved from either function name strings or from hashes. The first is in the *allocate_memory* function following the *resolve_ntdll_handle* function call described above. The handle to that DLL is stored in a global data variable for later use in this function as well as elsewhere. All the steps in this function set up an obfuscated call to *ZwAllocateVirtualMemory* which allocates memory where subsequent functions will write the next stage. In some samples, the size of the memory to allocate is located in one value, but in later samples, the value is calculated from a number of different instructions (Figure 9). This is an anti-analysis feature to make static analysis more difficult. This specifically makes automated unpacking more challenging when the size of the next stage cannot be lifted directly from a single location in the binary.

Figure 9: Instructions used to spread the payload size across multiple locations as anti-analysis.

After the handle to NTDLL has been resolved, the first two bytes of that location are checked for the DOS Header magic number, MZ. If that is found, it parses the rest of the headers to locate the address of the export directory. Then in the export directory, it loops through each export and finds the name of the exported function. That string of characters is run through a hash algorithm. This algorithm in some samples is located in its own subroutine and in others (Figure 10) it is inlined in the *allocate_memory* or *resolve_api_hash* functions.

The particular variant of the hash algorithm shown in Figure 10 multiplies by 257 (0x101) with a signed add. This algorithm is similar to hash algorithms listed in HashDB, but this exact one is not in that database. Across many samples of *pkr_mtsi*, the hashing algorithm in any particular sample is the same general algorithm, but many have different constants other than 257. A YARA rule for detecting this algorithm in any malware sample, not just this packer, is provided at the end of the blog. Note: this particular YARA rule should not be used to determine maliciousness of a sample. It simply identifies the presence of this hashing algorithm.

If the calculated hash matches the hard-coded hash, in this case for *ZwAllocateVirtualMemory*, then the pointer to that function is called in an obfuscated way. The input parameters for this call are in different orders in different samples and sometimes are interleaved with junk API calls. Both of those are anti-analysis tricks to make writing a signature more difficult. All of the steps described above are shown in the next figure.

Figure 10: Get a function pointer in NTDLL based on a hash of the function name.

In addition to inline capabilities like shown above, a dedicated function is also sometimes used to resolve API functions from hashes. This function is structured similarly to the one above, but it is used to resolve arbitrary API

function hashes, not just ones from NTDLL that are hard-coded. Again, in some samples the hash calculation is located in a dedicated subroutine. And in other samples, the hash calculation is inline (Figure 11).

Figure 11: Function to resolve API hash to function pointer.

Loading Other DLL Handles

A separate function for resolving DLL handles is used on all DLLs other than *ntdll.dll*. This resolver function uses a numeric, hard-coded key along with pointers to the API functions *RtlInitUnicodeString*, *LdrGetDllHandle*, and *LdrLoadDll* as input parameters. The difference between a DLL hash and a numeric key is that the key does not use a hashing function inside this type of resolver function. There is only a series of conditional comparisons to hard-coded keys, each of which corresponds to a code block containing the name of the desired DLL obfuscated in a stack string. The number of possible DLL names corresponds to the number of stack string blocks and DLL keys in the resolver function. The more DLL handles that this particular payload requires, the longer the resolver function is. Figure 12 shows the input DLL key in register *ebx* being compared to each of the DLL keys in a series. If the hard-coded value is not equal, it jumps to the next comparison. If it is equal, it jumps to the start of the stack string containing the DLL name.

Figure 12: Comparing the DLL key to each hard-coded value.

Each block of instructions containing the obfuscated DLL name starts with a partial stack string with some locations being stomped similar to the *gs* offset noted earlier.

Figure 13: Partial stack string with some location stomping.

The remainder of the block performs a series of calculations to decode obfuscated bytes and then write the characters to the appropriate offset in the DLL name string. In Figure 14, an example of this is bordered in red (1). This area also contains instructions that load the pointers to APIs called later in the function to the registers they are called from(2).

Figure 14: Writing decoded characters to DLL name and preparing API function calls.

The obfuscated stack strings concealing the DLL names can be observed in a debugger. However, since they are contiguous code blocks, they also lend themselves to the faster way of simply dumping the instructions and emulating them using Binary Refinery's *vstack* unit. The command for this emulation is the following:

```
emit dump.dat | vstack -a x64 -p 1: -n 1: -I -M -v -w 500
```

Figure 15: DLL names in output from Binary Refinery's vstack unit.

The DLLs used in this particular sample are *ADVAPI32.dll*, *KERNEL32.DLL*, *msvcrt.dll*, *NETAPI32.dll*, and *WS2_32.dll* (Figure 15). These DLL names are ASCII, but the API functions to load the DLL's handle take a unicode string structure as input. Therefore, an empty struct is initialized using *RtlInitUnicodeString* which is then populated by a wide string version of the DLL name. The ASCII string is converted to the wide string using SIMD

data rearrangement instructions (Figure 16). This code block makes a very good byte pattern for detection in YARA and is used as the basis for the *pkr_mtsi_UnpackMakeWide_1* rule provided in Appendix A.

Figure 16: SIMD instructions block converting ASCII DLL name to a wide string.

Lastly, in this resolver function, an attempt is made to get the DLL handle for an already loaded DLL. If that fails, the DLL is loaded outright. Finally, the handle is the return value of the function (Figure 17).

Figure 17: Attempt to get DLL handle then load on failure.

UPX TLS Callback Fixups

The EXE variants of *pkr_mtsi* are straightforward and pass execution to the UPX unpacker stub directly once. The DLLs, however, can have more than one pathway to execute the next stage. The main function always unpacks the next stage module into newly allocated memory and then makes any necessary adjustments such as TLS callbacks. A series of fixups which add the base address of the allocated memory to the relative offsets from the UPX module and then write the resulting virtual addresses to the module are shown in the next figure.

Figure 18: TLS callback fixups to adjust them to the actual allocated memory base address.

Fortunately, UPX is open source, so one can consult the commented code on Github and figure out exactly what is located where these fixups occur (Figure 20). That set of fixups write virtual addresses in the UPX TLS directory as well as making a callback array that points to the *PETLSC2* function in the UPX stub (Figure 19).

Figure 19: TLS callback structs in UPX module.

Figure 20: TLS callback support in UPX source code.

UPX Import Fixups

In the main function, the DLL resolvers and the API function resolvers work in series. The first few API function hashes are resolved from NTDLL. That handle is loaded from the global data variable written by the *allocate_memory* function. From that DLL, three function hashes are resolved to the three functions used in the generic DLL resolver function. Those three hashes are *LdrGetDllHandle*, *RtlInitUnicodeString*, and *LdrLoadDll*. The DLL handles from the generic resolvers are then used to resolve API function hashes. The resulting function pointers are written to the next stage UPX module's imports. This series of actions is in lieu of UPX being able to perform its own import resolutions because it is not being loaded by a standard loading process. Figure 21 shows a snippet that includes all three kinds of functions working in a series.

Figure 21: API and DLL hash resolvers writing import to the next stage UPX module.

Page Protection Mistakes

The final action the packer takes before calling the UPX entry point in the next stage is to attempt to change the memory page protections on sections in the allocated memory where the payload has been written and adjusted. However, there is a fortuitous bug in the adversary's code here. There are three calls to *NtProtectVirtualMemory*

used in this packer. This function uses the same protection constants as *VirtualProtect*. Look carefully at the constants marked in red in the next figure. The 0x6 in all three is 0x02 | 0x4 which are *PAGE_READONLY* and *PAGE_READWRITE*. This results in an invalid page protection input value for the function.

***Figure 22: Invalid page protection input values to* NtProtectVirtualMemory.**

This programming flaw provides a detection opportunity. These three calls will generate three errors in a row that can be used to develop behavioral detections in EDR telemetry. The error return value of *C0000045 STATUS_INVALID_PAGE_PROTECTION* is shown in the debugger in Figure 23.

Figure 23: Invalid page protection errors in the debugger.

UPX Execution From Main

In the DLL variants, when the main function of the packer is run, the Windows x64 loader parameter *fdwReason* is used by the packer to make a decision as to whether or not to call the UPX unpacker stub (Figure 24). This is probably to prevent a second unnecessary unpacking process if main is triggered by DLL load and unload events that are not specifically "1" (*DLL_PROCESS_ATTACH*).

Figure 24: Conditional call to the next stage UPX module entry point at the unpacker stub.

Note the three input parameters in the call to the next stage. The data from those parameters is passed all the way through the UPX unpacker stub into the original entry point of the payload. Figures 25, 26 and 27 show the unpacker stub function prologue and epilogue as well as the prologue of the payload showing where these three data values are passed in through.

Figure 25: Unpacker UPX stub prologue.

Figure 26: Unpacker UPX stub epilogue.

Figure 27: Payload OEP prologue.

UPX Execution From DLL Export

Also, the packer's own DLL export appears to pass four values to the function that it calls in the payload (Figure 28).

Figure 28: Four parameters passed to payload function of same name as packer's export.

What's interesting about these four values is that they are clobbered immediately in the payload function that is called (Figure 29). This indicates that this packer is coded to deliver a variety of payloads, and pass data to them on execution. But this particular payload does not have that capability.

Figure 29: Clobbered import values in the payload's exported function.

Next-Stage UPX Static Analysis

The second stage of *pkr_mtsi* is a UPX module that has been modified to remove parts that could be detected from the outer layer of the first stage. The second stage is broken up into small one to eight byte chunks that are then dispersed across the *write_payloadN* functions.

In earlier builds of this packer, there was no decoding process for any of these payload chunks. So, predictable byte patterns from the second stage would "shine through" the outer packer (Figure 30).

Figure 30: Chunks of plain ASCII from the second stage UPX module.

Many samples have the entire DOS and PE headers of the UPX module missing. Later variants removed the UPX magic number and headers located at the start of the UPX1 section (Figure 31).

Figure 31: Sample to the right has UPX magic number and other headers removed.

And even newer samples have text resources deleted. It seems like this adversary is removing parts of the UPX module to prevent detection and will remove anything as long as the module will still execute and load the payload final stage.

Conclusion

This analysis shows that, despite ongoing adversary iteration, *pkr_mtsi* exposes multiple durable detection and response opportunities that defenders can operationalize immediately. Preventive controls should emphasize behavioral detections centered on early-stage execution patterns, including deterministic memory allocation followed by dense sequences of small immediate-value writes, obfuscated resolution of *ZwAllocateVirtualMemory*, anomalous PEB traversal for API resolution, and excessive nonfunctional use of GDI APIs. The programming flaw involving repeated *NtProtectVirtualMemory* calls with invalid protection flags presents a particularly high-signal opportunity for resilient EDR and telemetry-based detections.

For DFIR practitioners, understanding the packer’s staged architecture, modified UPX intermediary, and alternate execution paths, especially DLL-based execution via *regsvr32.exe*, enables faster triage, more reliable unpacking, and clearer separation of packer behavior from payload functionality. Together, the techniques and detection logic presented in this report allow defenders to disrupt *pkr_mtsi* intrusion chains earlier in the attack lifecycle and investigate active incidents more efficiently and confidently. Complete analysis of the next stage and payloads will be covered in an upcoming RL research post.

Appendix A

YARA Rules

```
 import "pe"

rule pkr_mtsi_1
{
    meta:
        author = "Malware Utkonos"
        date = "2025-10-27"
```

```
        description = "Matches pkr_mtsi packed samples."
        revision = 8
    strings:
        // VirtualAlloc call for location to write payload
        $alloc1 = { ba[4] 3?c9 41b800300000 41b940000000 ff15[4] 488905[4] 4883c42? }
        $alloc2 = { ba[4] 3?c9 41b940000000 41b800300000 ff15[4] 488905[4] 4883c42? }
        $alloc3 = { ba[4] 41b800300000 3?c9 41b940000000 ff15[4] 488905[4] 4883c42? }
        $alloc4 = { ba[4] 41b800300000 41b940000000 3?c9 ff15[4] 488905[4] 4883c42? }
        $alloc5 = { ba[4] 41b940000000 41b800300000 3?c9 ff15[4] 488905[4] 4883c42? }
        $alloc6 = { ba[4] 41b940000000 3?c9 41b800300000 ff15[4] 488905[4] 4883c42? }
            // 18008fb70  4883ec28          sub     rsp, 0x28
            // 18008fb74  ba00900400        mov     edx, 0x49000
            // 18008fb79  41b940000000      mov     r9d, 0x40
            // 18008fb7f  41b800300000      mov     r8d, 0x3000
            // 18008fb85  33c9              xor     ecx, ecx  {0x0}
            // 18008fb87  ff153bc50200      call    qword [rel VirtualAlloc]
            // 18008fb8d  4889056cd40200    mov     qword [rel data_1800bd000], rax
            // 18008fb94  4883c428          add     rsp, 0x28
            // 18008fb98  c3                retn     {__return_addr}

        // ZwAllocateVirtualMemory call for location to write payload
        $alloc7  = { c74424??40000000 [0-40] c74424??00300000 [0-40] 488d?424?? [0-40] ff }
        $alloc8  = { c74424??40000000 [0-40] 488d?424?? [0-40] c74424??00300000 [0-40] ff }
        $alloc9  = { c74424??00300000 [0-40] c74424??40000000 [0-40] 488d?424?? [0-40] ff }
        $alloc10 = { c74424??00300000 [0-40] 488d?424?? [0-40] c74424??40000000 [0-40] ff }
        $alloc11 = { 488d?424?? [0-40] c74424??00300000 [0-40] c74424??40000000 [0-40] ff }
        $alloc12 = { 488d?424?? [0-40] c74424??40000000 [0-40] c74424??00300000 [0-40] ff }
            // 1800371e6  4c8d4c2460        lea     r9, [rsp+0x60]
            // 1800371eb  c744242840000000  mov     dword [rsp+0x28], 0x40
            // 1800371f3  4533c0            xor     r8d, r8d  {0x0}
            // 1800371f6  c744242000300000  mov     dword [rsp+0x20], 0x3000
            // 1800371fe  488d542458        lea     rdx, [rsp+0x58]
            // 180037203  48c7c1ffffffff    mov     rcx, 0xffffffffffffffff
            // 18003720a  ffd3              call    rbx

        // Loop used to copy DLL name to stack
        $find = { 0fb????? [0-20] ( b?3f000000 | c64424??3f ) [0-150] 4?ffc? }
            // 140035490  0fb71446          movzx   edx, word [rsi+rax*2]
            // 140035494  41b83f000000      mov     r8d, 0x3f
            // 14003549a  66413bd4          cmp     dx, r12w
            // 14003549e  0fb6ca            movzx   ecx, dl
            // 1400354a1  440f42c1          cmovb   r8d, ecx
            // 1400354a5  4488440430        mov     byte [rsp+rax+0x30], r8b
            // 1400354aa  48ffc0            inc     rax
        $ll = "LoadLibraryA"

        // Instruction that loads an 8 byte chunk of payload
```

```
        $load = { ( 48b8 | 48b9 | 48ba | 48bb | 48bd | 48be | 48bf | 49b9 | 49b8 | 49bf | 49bb | 49bd | 49be | 4
            // 180183805  48b884552e9032fc8c12   mov     rax, 0x128cfc32902e5584
            // 18018380f  48898424c8050000   mov     qword [rsp+0x5c8], rax  {0x128cfc32902e5584}

        // Instructions that write payload chunks to an offset in memory
        $write1 = { 6641c78? }  // Loose word-immediate memory store write via C7 (disp32, REX base)
            // 7ff68bc961fe  6641c7873cd80100d9b2   mov     word [r15+0x1d83c], 0xb2d9  {0xb2d9}
        $write2 = { ~6641c78? }  // Loose dword-immediate memory store write via C7 (disp32, REX base, no 0x66 p
            // 7ff68bc6830f  41c7867de0020080e93d4d   mov     dword [r14+0x2e07d], 0x4d3de980
        $write3 = { ~6641c78? }  // Loose dword-immediate memory store write via C7 (disp32, REX base, no 0x66 p
            // 7ff68bc6830f  41c7867de0020080e93d4d   mov     dword [r14+0x2e07d], 0x4d3de980
        $write4 = { 66c78? }  // Loose word-immediate memory store via 66+C7 (disp32 addressing; no SIB)
            // 1400074c6  66c78674680100dafe   mov     word [rsi+0x16874], 0xfeda  {0xfeda}
        $write5 = { 48898424????0000 488b8424????0000 488b8c24????0000 488908 }  // Block: spill qword to [rsp+0
            // 1400a480d  4889842418050000   mov     qword [rsp+0x518], rax  {-0x3ba224b6c292ac62}
            // 1400a4815  488b842410010000   mov     rax, qword [rsp+0x110]
            // 1400a481d  488b8c2418050000   mov     rcx, qword [rsp+0x518]  {-0x3ba224b6c292ac62}
            // 1400a4825  488908             mov     qword [rax], rcx  {-0x3ba224b6c292ac62}

        // Many calls to CreateSolidBrush with random three byte colors
        $csb = { b9[3]00 ff15[3]00 }
            // 18000106c  b956a76f00         mov     ecx, 0x6fa756
            // 180001071  ff1531c00300       call    qword [rel CreateSolidBrush]
    condition:
        uint16(0) == 0x5a4d and uint32(uint32(0x3c)) == 0x00004550 and
        1 of ($alloc*) and

        // Samples that don't have LoadLibraryA will have the loop that moves a DLL name to the stack.
        ($find or $ll) and

        // The payload is broken into many 8 byte chunks located in immediate values moved into
        // a register by these instructions. There are at least 100 of the and more depending on
        // the size of the payload image.
        (
        #load > 1000 or

        // There are various instructions that write a payload chunk to an offset in a memory location.
        #write1 + #write2 + #write3 + #write4 + #write5 > 1000
        ) and

        // pkr_mtsi always imports more than 20 GDI functions to use as junk code.
        pe.imports("gdi32.dll") > 15 and

        // Many calls to CreateSolidBrush with random three byte colors
        for 20 i in (1..500) : (
            uint32(@csb[i] + 7) ==
            (
```

```
                pe.import_rva("GDI32.dll", "CreateSolidBrush")
            )
            -
            (
                (@csb[i] + !csb[i]) - pe.sections[pe.section_index(@csb[i])].raw_data_offset + pe.sections[pe.se
            )
        )
}

rule pkr_mtsi_UnpackMakeWide_1
{
    meta:
        author = "Malware Utkonos"
        date = "2025-11-15"
        description = "Instructions that unpack ascii strings into wide strings found in pkr_mtsi samples."
    strings:
        $op = { 660f6e4404?? 660f60c0 660f71e008 660fd64445?? 4883c004 483bc1 72 }
            // 18001b1f0  660f6e440430       movd    xmm0, dword [rsp+rax+0x30]
            // 18001b1f6  660f60c0           punpcklbw xmm0, xmm0
            // 18001b1fa  660f71e008         psraw   xmm0, 0x8
            // 18001b1ff  660fd64445a0       movq    qword [rbp+rax*2-0x60], xmm0
            // 18001b205  4883c004           add     rax, 0x4
            // 18001b209  483bc1             cmp     rax, rcx
            // 18001b20c  72e2               jb      0x18001b1f0
    condition:
        uint16(0) == 0x5a4d and uint32(uint32(0x3c)) == 0x00004550 and
        $op
}

rule IsDebuggerPresent_LoopTrap_1
{
    meta:
        author = "Malware Utkonos"
        date = "2025-11-13"
        description = "Matches an infinite loop reached if IsDebuggerPresent is true."
    strings:
        $op = { ff15[4] 85c0 7402 ebfe }
            // 18000f398  ff15c25d0000       call    qword [rel IsDebuggerPresent]
            // 18000f39e  85c0               test    eax, eax
            // 18000f3a0  7402               je      0x18000f3a4
            // 18000f3a2  ebfe               jmp     0x18000f3a2
    condition:
        uint16(0) == 0x5a4d and uint32(uint32(0x3c)) == 0x00004550 and
        for any i in (1..200) : (
            uint32(@op[i] + 2) ==
            (
                pe.import_rva("KERNEL32.dll", "IsDebuggerPresent")
```

```
            )
            -
            (
                (@op[i] + 6) - pe.sections[pe.section_index(@op[i])].raw_data_offset + pe.sections[pe.section_in
            )
        )
}

rule Mul257_Add_Signed_1
{
    meta:
        author = "Malware Utkonos"
        date = "2025-12-10"
        description = "Matches loop that implements hashing algo: multiply 257 with signed add."
        warning = "This rule detects the presence of this hash algorithm in benign and malicious samples."
    strings:
        $op = { 69??01010000 ( 488d??01 | 488d642401 | 4d8d??01 | 4d8d642401 ) ( 0fbe?? | 400fbe?? ) ( 01?? | 03
            // 18001b400  69c001010000       imul    eax, eax, 0x101
            // 18001b406  488d5201           lea     rdx, [rdx+0x1]
            // 18001b40a  0fbec9             movsx   ecx, cl
            // 18001b40d  03c1               add     eax, ecx
            // 18001b40f  0fb60a             movzx   ecx, byte [rdx]
            // 18001b412  84c9               test    cl, cl
            // 18001b414  75ea               jne     0x18001b400
    condition:
        $op
}
```

Source: https://www.reversinglabs.com/blog/unpacking-pkr_mtsi