Malicious Packer pkr_ce1a

malwarology.substack.com/p/malicious-packer-pkr_ce1a

Malwarology LLC

First Stage

Summary

This packer has been observed delivering a wide variety of malware families including SmokeLoader and Vidar among many others. It has been observed in the wild going back a number of years potentially to 2017. The particular variant of the packer analyzed here contains two sets of bytes with no apparent use which occur on either side of values that are integral to the decoding and unpacking process. These two byte strings are stable across many months of observed samples of this packer. What follows is a detailed analysis of the first stage of one sample of this packer which delivers a SmokeLoader payload. Until a widely recognized name or identifier can be determined for this packer, it has the designation `pkr_ce1a`.

Identification

This sample has two known filenames. The first, `6523.exe`, is observed in the wild in the path component of the URL used to distribute the file.1 The second, `povgwaoci.iwe`, is located in the `RT_VERSION` resource within the `VS_VERSIONINFO` structure in a field named `InternationalName`. This field along with others in the same StringTable structure are not parsed by Exiftool or Cerbero Suite. This indicates that the structure is malformed or non-standard.

According to AV detection results, the correct identification of the unpacked file, *SmokeLoader*, does appear. There is also one detection based on dynamic analysis: *Zenpack*.

The import hash of this sample is shared by a group of other files.2 However, the large majority of imported functions are called in dead code located after opaque predicates. Therefore, the usefulness of this import hash is almost nothing. Closer analysis of the opaque predicates in this sample can be found below.

Behavioral and Code Analysis Findings

Build Analysis

According to the File Header, the timestamp of compilation is 2022-02-16T10:14:32Z. The linker version found in the PE32 Optional Header is 9.0.

```
00400108  struct PE32_Optional_Header __pe32_optional_header =
00400108  {
00400108      enum pe_magic magic = PE_32BIT
0040010a      uint8_t majorLinkerVersion = 0x9
0040010b      uint8_t minorLinkerVersion = 0x0
0040010c      uint32_t sizeOfCode = 0x16c00
00400110      uint32_t sizeOfInitializedData = 0x82c00
00400114      uint32_t sizeOfUninitializedData = 0x0
00400118      uint32_t addressOfEntryPoint = 0x7140
0040011c      uint32_t baseOfCode = 0x1000
00400120      uint32_t baseOfData = 0x18000
00400124      uint32_t imageBase = 0x400000
```

Linker Version 9.0

The compiler is identified as *Visual Studio 2008 Release* according to function signatures in Ghidra that match a number of library functions in the sample. One example of this detection for the `___security_init_cookie` function is shown in the figure below.

```
                     *************************************************************
                     * Library Function - Single Match                           *
                     *  ___security_init_cookie                                  *
                     *                                                           *
                     * Library: Visual Studio 2008 Release                       *
                     *************************************************************
                     void __cdecl ___security_init_cookie(void)
     void              <VOID>         <RETURN>
     undefined4        Stack[-0x8]:4  local_8                    XREF[2]:     0040b093(RW),
                                                                              0040b0bf(R)
     undefined4        Stack[-0xc]:4  local_c                    XREF[3]:     0040b08f(RW),
                                                                              0040b0b5(*),
                                                                              0040b0c2(R)
     undefined4        Stack[-0x10]:4 local_10                   XREF[1]:     0040b0e7(R)
     undefined4        Stack[-0x14]:4 local_14                   XREF[2]:     0040b0dd(*),
                                                                              0040b0ea(R)
                     ___security_init_cookie                XREF[1]:     entry:00407140(c)
   0040b082 8b ff        MOV        EDI,EDI
   0040b084 55           PUSH       EBP
```

Compiler Detection: Visual Studio 2008 Release

The majority of the library functions found in the sample are from *Microsoft Visual C++ 9.0.21022*. This is identified in the sample's rich signature.

Analysis [Rich Signature]

```
; Rich Signature

product id: 0x0084 minor build version: 21022 count: 44
product id: 0x0095 minor build version: 21022 count: 29
product id: 0x0083 minor build version: 21022 count: 141
product id: 0x007B minor build version: 50727 count: 7
product id: 0x0001 minor build version: 0 count: 129
product id: 0x008A minor build version: 21022 count: 1
product id: 0x0094 minor build version: 21022 count: 1
product id: 0x0091 minor build version: 21022 count: 1
```

Rich Signature: VC++ 9.0.21022

Main Function

```
main:
00406569  a13ccc4100      mov     eax, dword [size_part1]
0040656e  a314b24800      mov     dword [shellcode_size], eax
00406573  b83b2d0b00      mov     eax, 0xb2d3b  // Size part 2
00406578  // Calculate shellcode size
00406578  010514b24800    add     dword [shellcode_size], eax
0040657e  a19cbc4100      mov     eax, dword [scaddr_part1]
00406583  // lpLibFileName: kernel32.dll
00406583  68cc4b4000      push    kernel32_1  {"kernel32.dll"}
00406588  a318b24800      mov     dword [shellcode_addr_part1], eax
0040658d  ff1534104000    call    dword [LoadLibraryW]
00406593  // lpProcName: LocalAlloc
00406593  68e84b4000      push    _LocalAlloc  {"LocalAlloc"}
00406598  50              push    eax {_&kernel32}  // hModule: kernel32
00406598  // hModule: kernel32
00406599  a3d0d54200      mov     dword [data_42d5d0], eax
0040659e  ff1568104000    call    dword [GetProcAddress]
004065a4  // uBytes: 63072
004065a4  ff3514b24800    push    dword [shellcode_size] {uBytes}
004065aa  a344d54200      mov     dword [data_42d544], eax
004065af  6a00            push    0x0  // uFlags: LMEM_FIXED
004065b1  ffd0            call    eax  // LocalAlloc -> &shellcode
004065b3  a340d54200      mov     dword [_&shellcode], eax
004065b8  e80bf3ffff      call    change_protection
004065bd  e8bbf3ffff      call    unpack_shellcode
004065c2  a140d54200      mov     eax, dword [_&shellcode]
004065c7  a3d4d54200      mov     dword [_&shellcode_oep], eax
004065cc  ffd0            call    eax  // Call &shellcode_oep
004065ce  33c0            xor     eax, eax  {0x0}
004065d0  c21000          retn    0x10 {__return_addr}
```

Main Function

The instructions highlighted in yellow in the figure above are examples of junk code insertion. These are dummy instructions that are placed between relevant instructions with the goal of making signature development more difficult. The instructions at the end of the function highlighted in white are not executed. These highlight colorings are used throughout the screenshots of this analysis.

The first instructions in the main function calculate the size of the encoded data that contains the shellcode which is the next stage of the packer. The first part is read from a constant in the `.data` section at address `0x41cc3c`. The second part is hardcoded in the main function at address `0x406573`. The size is calculated in the next instruction by adding the two parts together. The resulting size is written to a variable in the `.data` section. Highlighted in the figure below are the bytes before and after the first constant. These two sets of bytes are not read during the execution of the packer, and their purpose is unknown. However, they are stable across builds of this packer going back for months at least.

```
0041cc10  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
0041cc20  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
0041cc30  00 00 00 00 00 00 00 00-69 9a f9 74              ........i..t
  // Distance to encoded shellcode: 0x5c4
0041cc3c  int32_t size_part1 = -0xa36db

0041cc40  96 aa cb 46 00 00 00 00-00 00 00 00 00 00 00 00  ...F............
0041cc50  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
0041cc60  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
```

Stable Surrounding Bytes

The address where the encoded data is located goes through a similar process. The first part is read from the `.data` section then written to a variable in the same section. The addition to the second part does not occur until later in the `unpack_shellcode` function. The bytes surrounding this part are shown in the figure below.

```
0041bc70  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
0041bc80  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
0041bc90  00 00 00 00 00 00 00 00-94 48 8d 6a              .........H.j

0041bc9c  int32_t scaddr_part1 = 0x36a4c5

0041bca0  f2 16 0b 68 00 00 00 00-00 00 00 00 00 00 00 00  ...h............
0041bcb0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
0041bcc0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
```

Stable Surrounding Bytes

The next set of instructions handles loading `kernel32.dll` and resolving the address of `LocalAlloc`.3 These instructions taken together are unique to this packer and shared across hundreds of variants. The stray instruction at address `0x406588` is part of the previous logical grouping of instructions. This is an example of interleaving code that is meant to make signature development more difficult.

```
00406583  68cc4b4000      push    kernel32_1  {"kernel32.dll"}  // lpLibFileName: kernel32.dll
00406588  a318b24800      mov     dword [shellcode_addr_part1], eax
0040658d  ff1534104000    call    dword [LoadLibraryW]
00406593  68e84b4000      push    _LocalAlloc  {"LocalAlloc"}  // lpProcName: LocalAlloc
00406598  50              push    eax {_&kernel32}  // hModule: kernel32
00406598  // hModule: kernel32
00406599  a3d0d54200      mov     dword [data_42d5d0], eax
0040659e  ff1568104000    call    dword [GetProcAddress]
```

Resolve `LocalAlloc`

The call to `LocalAlloc` is obfuscated by calling it from the `eax` register. This is a type of function call obfuscation. The `uBytes` parameter to the function is 63072 bytes which is the result of the calculation described above.

```
004065a4  ff3514b24800       push    dword [shellcode_size] {uBytes}  // uBytes: 63072
004065aa  a344d54200         mov     dword [data_42d544], eax
004065af  6a00               push    0x0  // uFlags: LMEM_FIXED
004065b1  ffd0               call    eax  // LocalAlloc -> &shellcode
004065b3  a340d54200         mov     dword [_&shellcode], eax
```

Obfuscated Call to `LocalAlloc`

The final instructions in the main function are calls to other malicious functions and finally a call to the entry point of the decoded shellcode. The next stage of the packer starts after that call.

Change Protection Function

```
change_protection:
004058c8  55                push    ebp {__saved_ebp}
004058c9  8bec              mov     ebp, esp {__saved_ebp}
004058cb  51                push    ecx {flNewProtect}
004058cc  51                push    ecx {flOldProtect}
004058cd  // lpLibFileName: kernel32.dll
004058cd  68f0494000        push    kernel32_2  {"kernel32.dll"}
004058d2  ff1534104000      call    dword [LoadLibraryW]
004058d8  // lpProcName: VirtualProtect
004058d8  6860d34200        push    _VirtualProtect
004058dd  50                push    eax {_&kernel32}  // hModule: kernel32
004058dd  // hModule: kernel32
004058de  a3d0d54200        mov     dword [data_42d5d0], eax
004058e3  c60560d3420056    mov     byte [_VirtualProtect], 0x56  // V
004058ea  c60561d3420069    mov     byte [char_i], 0x69  // i
004058f1  c60562d3420072    mov     byte [char_r_1], 0x72  // r
004058f8  c60567d3420050    mov     byte [char_P], 0x50  // P
004058ff  c6056dd3420074    mov     byte [char_t_1], 0x74  // t
00405906  c6056ed3420000    mov     byte [char_null], 0x0
0040590d  c60563d3420074    mov     byte [char_t_2], 0x74  // t
00405914  c60564d3420075    mov     byte [char_u], 0x75  // u
0040591b  c60565d3420061    mov     byte [char_a], 0x61  // a
00405922  c60566d342006c    mov     byte [char_l], 0x6c  // l
00405929  c60568d3420072    mov     byte [char_r_2], 0x72  // r
00405930  c60569d342006f    mov     byte [char_o], 0x6f  // o
00405937  c6056ad3420074    mov     byte [char_t_3], 0x74  // t
0040593e  c6056bd3420065    mov     byte [char_e], 0x65  // e
00405945  c6056cd3420063    mov     byte [char_c], 0x63  // c
0040594c  ff1568104000      call    dword [GetProcAddress]
00405952  a338d54200        mov     dword [_&VirtualProtect], eax
00405957  // flNewProtect part 1
00405957  c745fc20000000    mov     dword [ebp-0x4 {flNewProtect_part1}], 0x20
0040595e  // flNewProtect part 2
0040595e  8345fc20          add     dword [ebp-0x4], 0x20  {0x40}
00405962  8d45f8            lea     eax, [ebp-0x8 {flOldProtect}]
00405965  // lpflOldProtect: &flOldProtect
00405965  50                push    eax {flOldProtect} {_&flOldProtect}
00405965  // lpflOldProtect: &flOldProtect
00405966  // flNewProtect: PAGE_EXECUTE_READWRITE
00405966  ff75fc            push    dword [ebp-0x4]  {0x40}
00405969  // dwSize: 63072
00405969  ff3514b24800      push    dword [shellcode_size] {dwSize}
0040596f  // lpAddress: &shellcode
0040596f  ff3540d54200      push    dword [_&shellcode] {lpAddress}
00405975  // flOldProtect -> PAGE_READWRITE
00405975  ff1538d54200      call    dword [_&VirtualProtect]
0040597b  c9                leave    {__saved_ebp}
0040597c  c3                retn     {__return_addr}
```

Change Protection Function

This function is a wrapper around an obfuscated call to `VirtualProtect`. Starting at address `0x4058e3` and continuing until the call to `GetProcAddress`, the name of the function `VirtualProtect` is written character-by-character, out of order, to a variable in the `.data` section. Building the function name in this way is an example of variable recomposition. The address of this string is then used as the `lpProcName` parameter to `GetProcAddress`. Finally, a call is made to `VirtualProtect` to change the protection from `PAGE_READWRITE` (`0x4`) to `PAGE_EXECUTE_READWRITE` (`0x40`) thus enabling execution in the newly allocated memory.

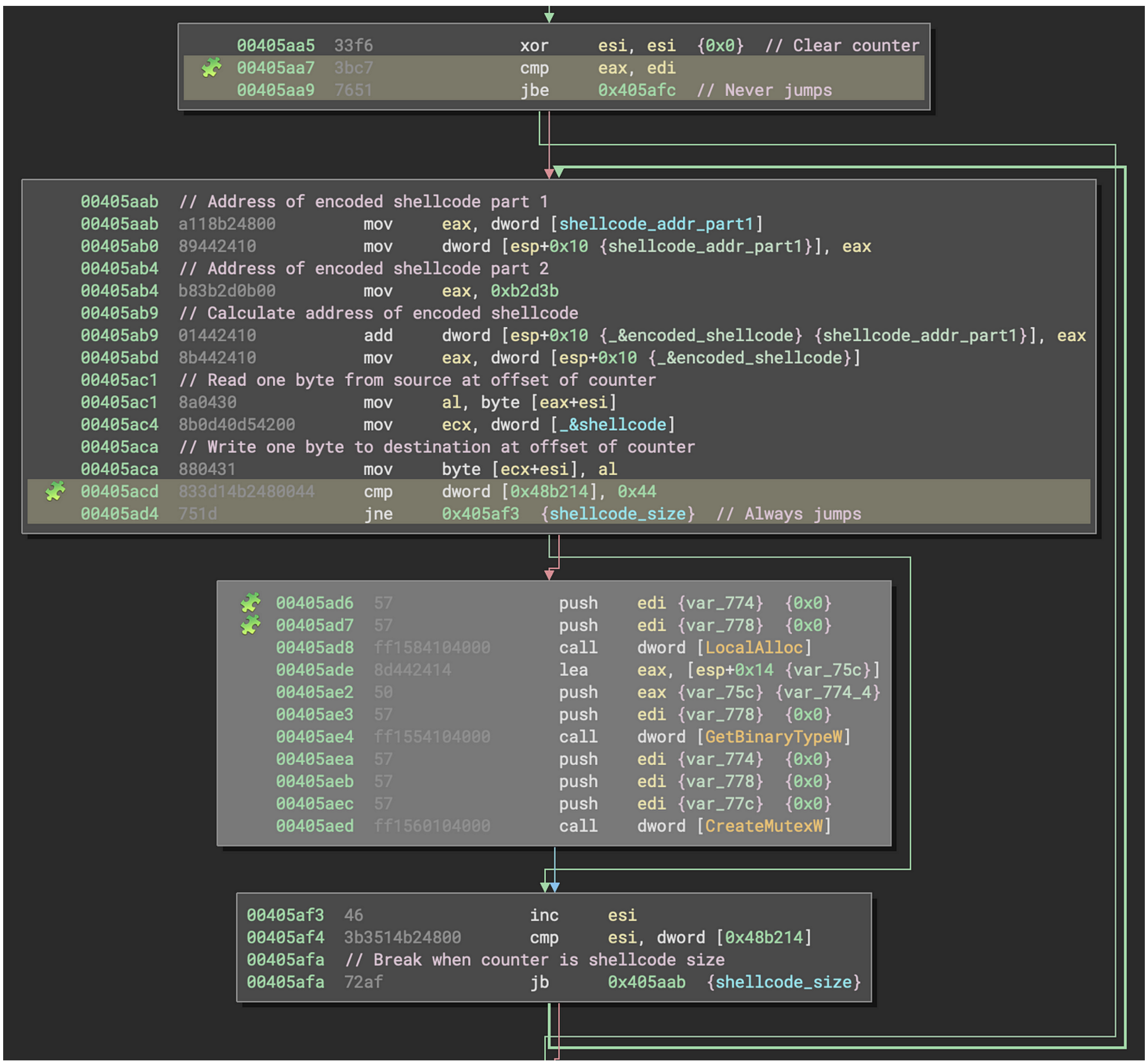
The `flNewProtect` parameter to the `VirtualProtect` function is also obfuscated by adding together `0x20` and `0x20`. This hides the `PAGE_EXECUTE_READWRITE` flag of `0x40` from being located near the call to `VirtualProtect`. This is a form of argument obfuscation.

```
00405957  c745fc20000000    mov     dword [ebp-0x4 {flNewProtect_part1}], 0x20  // flNewProtect part 1
0040595e  8345fc20          add     dword [ebp-0x4], 0x20  {0x40}  // flNewProtect part 2
```

Obfuscated New Protect Value

Unpack Shellcode Function

This function performs three actions that are interspersed with anti-analysis code. The first action is moving the encoded data from its starting location in the `.data` section to the newly allocated memory.

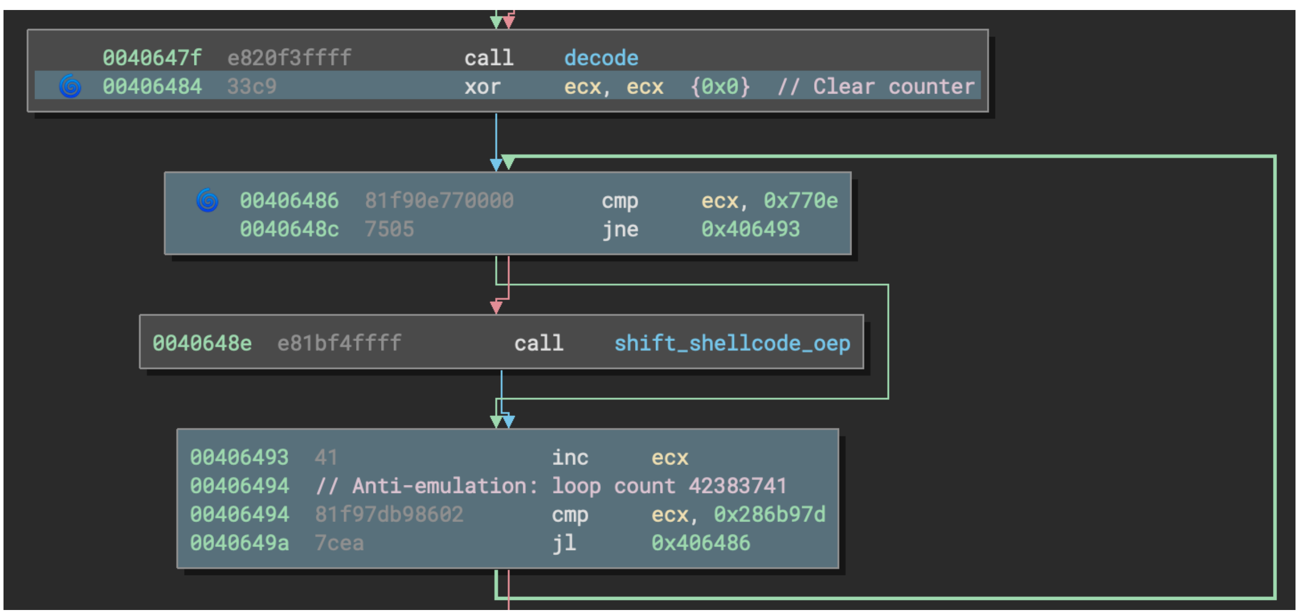


Move Encoded Data

Note the instructions at addresses `0x405aa7` and `0x405acd`. These are both opaque predicates. An opaque predicate appears to be a conditional jump, but the conditions can only be met in one way making the jump effectively unconditional. The one at the top is basically a fake: it never jumps. The one in the middle always jumps. This one additionally encloses a block of dead code. This packer very frequently couples dead code insertion with opaque predicates that always jump over the dead code. The function calls in the dead code are included in the import table making identification via import hash of little utility.

Later on in this function is the call to the `decode` function. After that is a call to a function which shifts the pointer to the decoded shellcode. The shift changes the offset from the start of this data to the location of the shellcode original execution point (OEP). Because the shellcode is position independent, this OEP is also offset zero of the shellcode. Both of these functions are analyzed in more detail below.

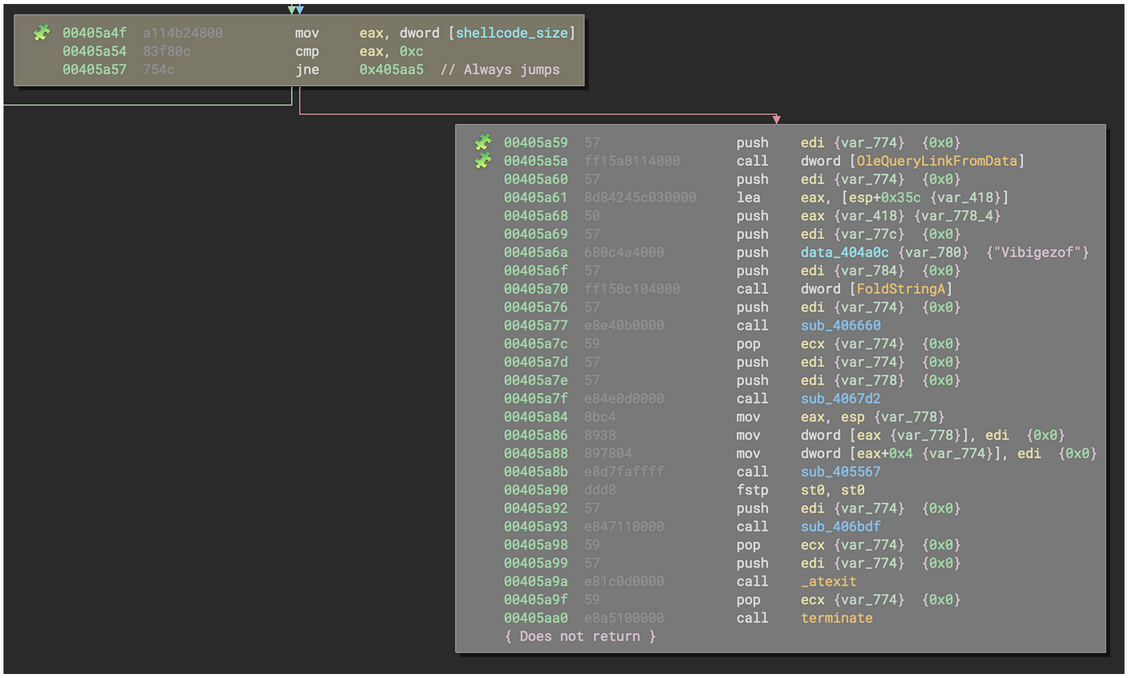
The `shift_shellcode_oep` function is additionally wrapped in an anti-emulator loop which has a very high number of iterations. This slows processing in an emulator which can potentially cause a timeout and an analysis failure.



Decode and Shift Shellcode OEP Functions

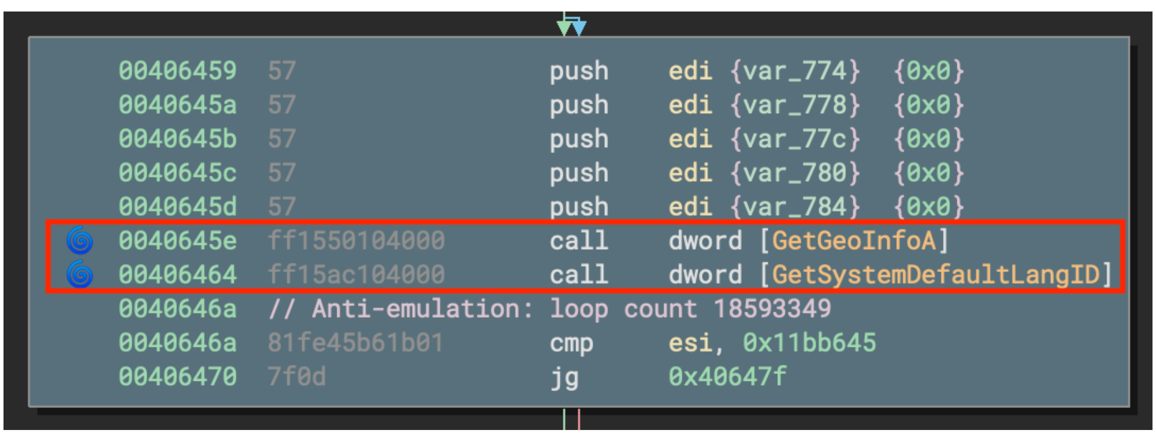
Note the comparison instruction at address `0x406486`. During the anti-emulation loop, the call to `shift_shellcode_oep` is made once when the counter reaches `0x770e`. This is a type of anti-emulation circumvention countermeasure. A basic method for circumventing extremely long loops that target emulators is to patch out the loop. Another is to detect the loop in the emulator and then modify the counter to leave the loop. Because the shift function is called once at a point in the loop, either of these circumventions could end up not executing this function and would leave the emulator unable to execute the shellcode correctly.

The `unpack_shellcode` function overall contains ten opaque predicates primarily of the same type shown above. One of them is slightly different in that it creates a dead end filled with dead code. The last instruction in the dead end is a call to `terminate`. Therefore, this appears to be a location where the execution of the packer ends.



Opaque Predicate with Dead End

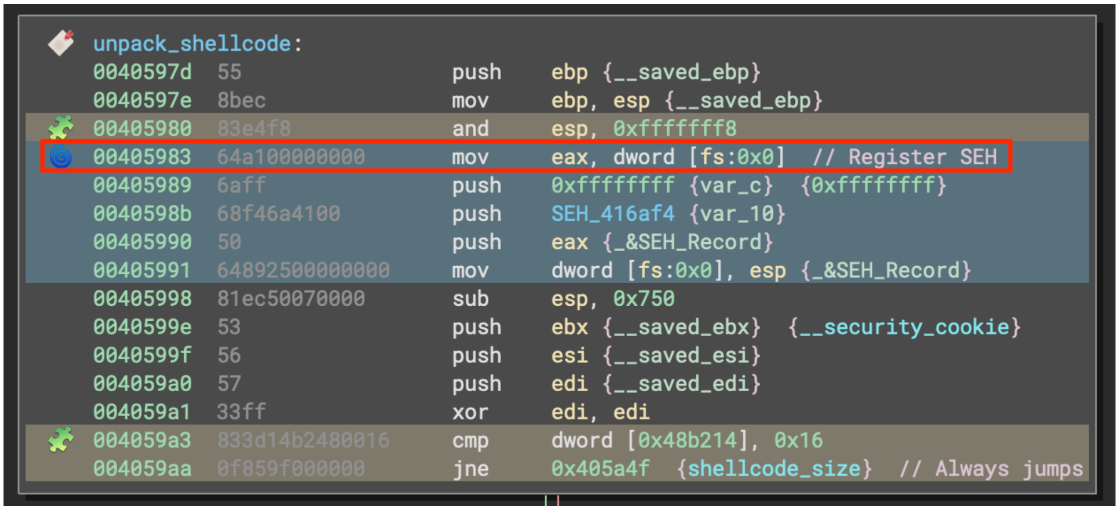
There are a total of four anti-emulation loops similar to the one analyzed above. Two of these wrap opaque predicates which in turn wrap inserted dead code. One, however, in addition to wrapping two opaque predicates, also contains two additional anti-emulator behaviors. Both of these are calls to unusual APIs: `GetGeoInfoA` and `GetSystemDefaultLangID`. During analysis, Qiling emulator halted with an exception because neither of these API calls have been implemented.



Anti-Emulation: Unusual API Calls

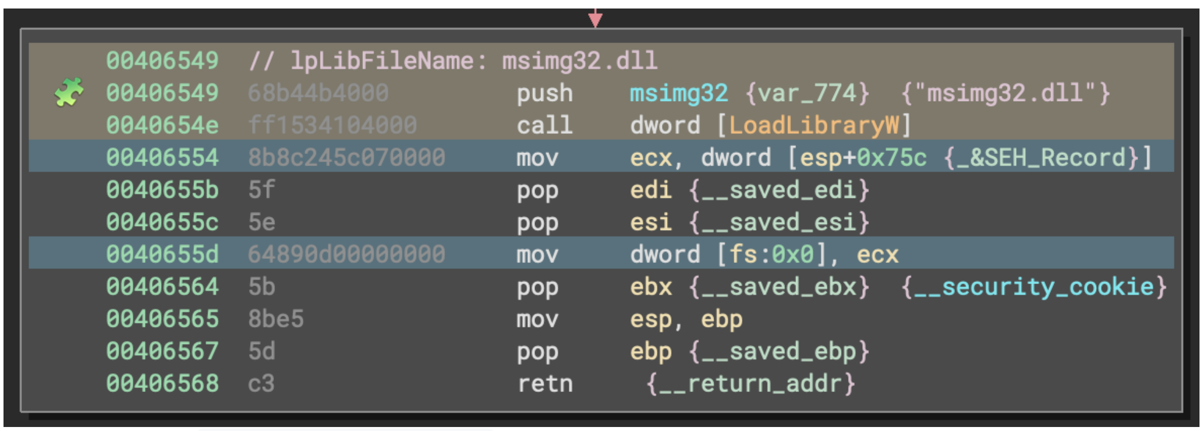
In the very first code block of the `unpack_shellcode` function, a new and subsequently unused exception handler is registered. Chances are about even that this is an anti-emulation behavior or it is just junk code. If it is anti-emulation, it is targeting older emulators based on specific versions of Unicorn Engine which do not implement access to the

Windows Thread Information Block (TIB). Moving the contents of `fs:0x0` as happens at address `0x405983` would fail in that particular environment. This type of anti-emulation is an unimplemented opcode.



Register New SEH

In the last block of the `unpack_shellcode` function, before the return, the SEH is reset back to the previous handler. This occurs right after a junk call to `LoadLibraryW` from which no functions are subsequently resolved.



Last Block of `unpack_shellcode`

Decode Function

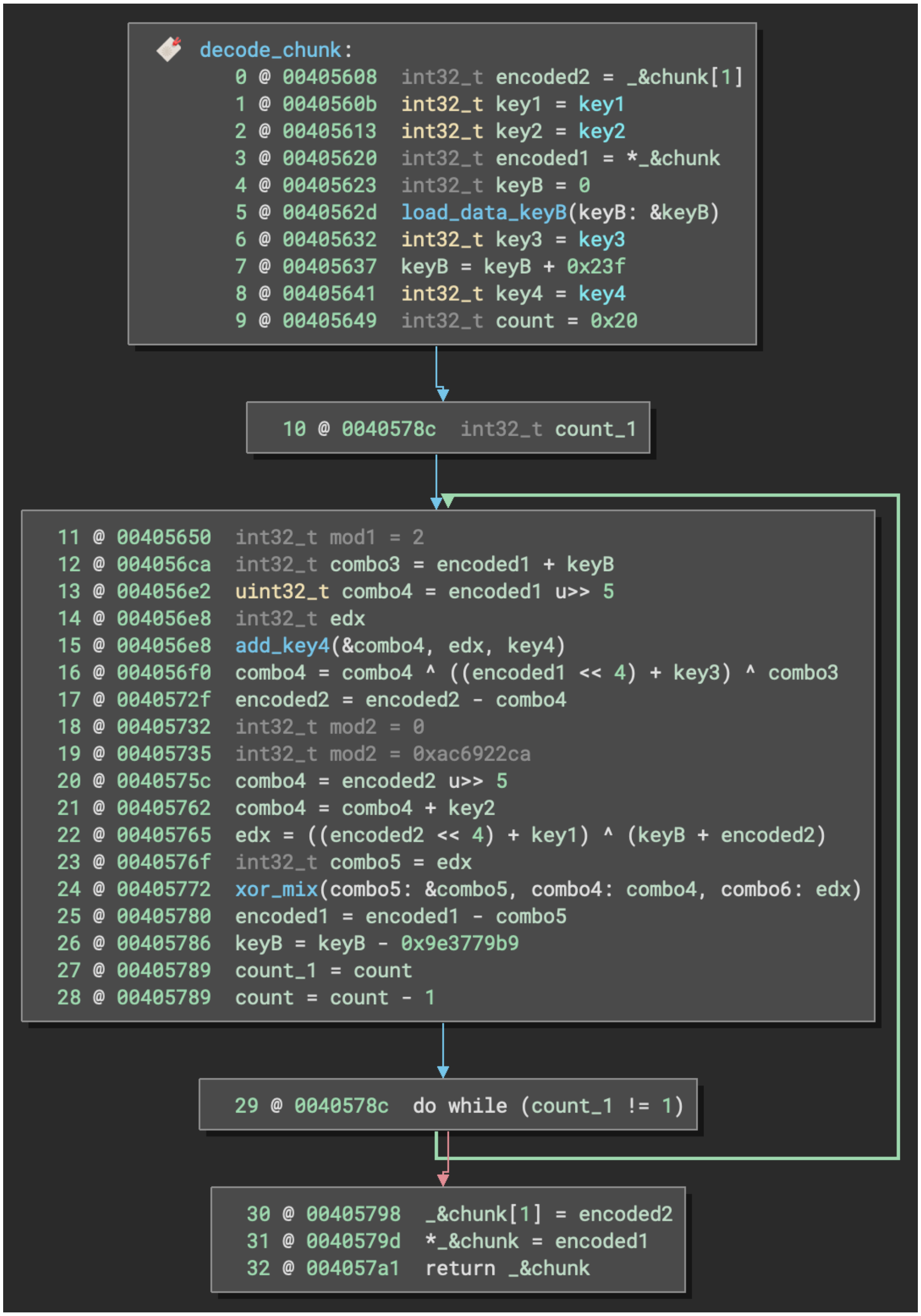
Decode Function (Truncated)

The data is decoded in chunks. So this function has three basic purposes: convert the encoded shellcode size to a chunk count by dividing by 8, wrapping a loop around the call to the `decode_chunk` function, and then calling that function. In the middle of this function is a very large insertion of dead code wrapped by an opaque predicate. This is highlighted in white in the figure above. It has also been truncated for the screenshot. There are two locations in this function with junk code insertion which impedes signature development: at addresses `0x4057c1` and `0x4057cc`.

The chunks are 8 bytes long, therefore at the end of the loop after the call to `decode_chunk`, 8 is added to the chunk pointer then the chunk count is decremented by one.

```
00405894  ff75fc            push    dword [ebp-0x4 {_&chunk}] {var_14}
00405897  e85bfdffff        call    decode_chunk
0040589c  // Shift pointer to next chunk
0040589c  8345fc08          add     dword [ebp-0x4 {_&chunk}], 0x8
004058a0  ff4df8            dec     dword [ebp-0x8 {chunk_count}]
004058a3  0f8535ffffff      jne     0x4057de
```
Chunk Decode Loop Control

Decode Chunk Function



Decompiled Decode Chunk Function

The `decode_chunk` function has four opaque predicates that each wrap dead code, and there are two locations with junk code insertion. In addition to these, the logic of the function is quite convoluted. However, after patching out all of the opaque predicates, dead code, and junk code, a cleaner decompilation graph can be analyzed. The figure above is that graph, fully annotated and cleaned up.

There are three tiny functions called during the decoding algorithm: `load_data_keyB`, `add_key4`, and, `xor_mix`. These simply isolate small pieces of the algorithm to impede analysis and make signature development more difficult.

```
load_data_keyB:
004055ed  // Load key B component 1
004055ed  8100e134efc6        add     dword [eax], 0xc6ef34e1
004055f3  c3                  retn     {__return_addr}

add_key4:
004055f4  0108                add     dword [eax], ecx
004055f6  c3                  retn     {__return_addr}

xor_mix:
004055d0  55                  push    ebp {__saved_ebp}
004055d1  8bec                mov     ebp, esp {__saved_ebp}
004055d3  51                  push    ecx {var_8}
004055d4  8365fc00            and     dword [ebp-0x4 {var_8_1}], 0x0
004055d8  8b450c              mov     eax, dword [ebp+0xc {combo6}]
004055db  8945fc              mov     dword [ebp-0x4 {combo6}], eax
004055de  8b4508              mov     eax, dword [ebp+0x8 {combo4}]
004055e1  3145fc              xor     dword [ebp-0x4 {combo7} {combo6}], eax
004055e4  8b45fc              mov     eax, dword [ebp-0x4 {combo7}]
004055e7  8901                mov     dword [ecx], eax
004055e9  c9                  leave    {__saved_ebp}
004055ea  c20800              retn    0x8 {__return_addr}
```

Algorithm Isolates

Shift Shellcode OEP Function

```
shift_shellcode_oep:
004058ae  55                  push    ebp {__saved_ebp}
004058af  8bec                mov     ebp, esp {__saved_ebp}
004058b1  51                  push    ecx {var_8}
004058b2  8365fc00            and     dword [ebp-0x4 {oep_offset}], 0x0
004058b6  8145fcf13b0000      add     dword [ebp-0x4], 0x3bf1
004058bd  8b45fc              mov     eax, dword [ebp-0x4]
004058c0  // Shift shellcode pointer to oep
004058c0  010540d54200        add     dword [_&shellcode], eax
004058c6  c9                  leave    {__saved_ebp}
004058c7  c3                  retn     {__return_addr}
```

Shift Shellcode OEP Function

The `shift_shellcode_oep` function is very simple. It adds `0x3bf1` to the address of the start of the decoded shellcode in allocated memory. This shifts the pointer from the start of the decoded data to the offset of the OEP. This is the address that is called in the main function.

Recommendations

This packer is polymorphic. Variants and builds share many of the same functionality, behavior, and code features. However, they are in different order with randomization of opaque predicates, junk code, and dead code. In spite of this, detection can be achieved by focusing on the two stretches of stable bytes which were identified above. The following two YARA rules match these bytes.

```
rule Packer_pkr_ce1a_ShellcodeSizePart
{
    meta:
        author = "Malwarology LLC"
        date = "2022-10-14"
        description = "Detects bytes surrounding the first part of the data size of
the second stage shellcode in Packer pkr_ce1a."
        reference = "https://malwarology.substack.com/p/malicious-packer-pkr_ce1a"
        sharing = "TLP:CLEAR"
        exemplar = "fc04e80d343f5929aea4aac77fb12485c7b07b3a3d2fc383d68912c9ad0666da"
        address = "0x41cc3c"
        packer = "pkr_ce1a"
    strings:
        $a = { 00699AF974[4]96AACB4600 }
    condition:
        $a and
        uint16(0) == 0x5A4D and
        uint32(uint32(0x3C)) == 0x00004550
}

rule Packer_pkr_ce1a_ShellcodeAddrPart
{
    meta:
        author = "Malwarology LLC"
        date = "2022-10-17"
        description = "Detects bytes surrounding the first part of the address of the
second stage shellcode in Packer pkr_ce1a."
        reference = "https://malwarology.substack.com/p/malicious-packer-pkr_ce1a"
        sharing = "TLP:CLEAR"
        exemplar = "fc04e80d343f5929aea4aac77fb12485c7b07b3a3d2fc383d68912c9ad0666da"
        address = "0x41bc9c"
        packer = "pkr_ce1a"
    strings:
        $a = { 0094488D6A[4]F2160B6800 }
    condition:
        $a and
        uint16(0) == 0x5A4D and
        uint32(uint32(0x3C)) == 0x00004550
}
```

Supporting Data and IOCs

File

- `Filename: 6523.exe`
- `Filename: povgwaoci.iwe`
- `MD5: 5663a767ac9d9b9efde3244125509cf3`
- `SHA1: 84f383a3ddb9f073655e1f6383b9c1d015e26524`
- `SHA25: fc04e80d343f5929aea4aac77fb12485c7b07b3a3d2fc383d68912c9ad0666da`
- `Imphash: bc57832ec1fddf960b28fd6e06cc17ba`
- `Timestamp: 2022-02-16T10:14:32Z`
- `File Type: Win32 EXE`
- `Magic: PE32 executable (GUI) Intel 80386, for MS Windows`
- `Size: 238080`
- `First Seen: 2022-10-14T18:37:13Z` [4]

Distribution URL

`hxxp[://]guluiiiimnstrannaer[.]net/dl/6523.exe`

Malware Behavior Catalog

- DEFENSE EVASION::Software Packing [F0001]
- ANTI-BEHAVIORAL ANALYSIS::Emulator Evasion::Undocumented Opcodes [B0005.002]
- ANTI-BEHAVIORAL ANALYSIS::Emulator Evasion::Unusual/Undocumented API Calls [B0005.003]
- ANTI-BEHAVIORAL ANALYSIS::Emulator Evasion::Extra Loops/Time Locks [B0005.004]
- ANTI-STATIC ANALYSIS::Disassembler Evasion::Argument Obfuscation [B0012.001]
- ANTI-STATIC ANALYSIS::Disassembler Evasion::Variable Recomposition [B0012.004]

- ANTI-STATIC ANALYSIS::Executable Code Obfuscation::Dead Code Insertion [B0032.003]
- ANTI-STATIC ANALYSIS::Executable Code Obfuscation::Junk Code Insertion [B0032.007]
- ANTI-STATIC ANALYSIS::Executable Code Obfuscation::Interleaving Code [B0032.014]

Malware Behavior Catalog Proposed Methods

- ANTI-BEHAVIORAL ANALYSIS::Emulator Evasion::Unimplemented Opcodes [B0005]
- ANTI-STATIC ANALYSIS::Executable Code Obfuscation::Opaque Predicate [B0032]
- ANTI-STATIC ANALYSIS::Executable Code Obfuscation::Function Call Obfuscation [B0032]

Analyses

- Intezer
- UnpacMe

1
Sample downloaded from hxxp[://]guluiiiimnstrannaer[.]net/dl/6523.exe

2
`Imphash: bc57832ec1fddf960b28fd6e06cc17ba`

3
LocalAlloc function (winbase.h)

4
VirusTotal