PikaBot Is Back With a Vengeance - Part 2

research.openanalysis.net/pikabot/debugging/string decryption/emulation/memulator/2023/11/19/new-pikabot-strings.html

OALABS Research November 19, 2023

Overview

This is a continuation of our work on the new Pikabot core module. Our initial analysis can be found here.

Sample

`39d6f7865949ae7bb846f56bff4f62a96d7277d2872fec68c09e1227e6db9206` UnpacMe

String Decryption

- Strings are inline
- The string data is built in a stack string (pushed on to the stack as DWORDs)
- The keys are cstrings in the `.data` section
- The first layer of encryption is RC4 (inline)
- The decrypted strings are base64 encoded
- The base64 encoded strings are then decrypted usuing AES CBC
    - The AES key and IV are themselves base64 strings but only the first 32/16 bytes are used and the strings are not decoded

Locating Strings

The RC4 key setup loops are good markers.

The first time we have a compare with 256 this is the entry to the RC4 decryption The second time we have a compare with 256 this is the key setup loop

```
3D 00 01 00 00                          cmp     eax, 100h
81 FE 00 01 00 00                       cmp     esi, 100h
81 FF 00 01 00 00                       cmp     edi, 100h
```

After the first compare with 256 but before the second compare with 256 when we have a div this is the key length.

```
F7 75 F0                                div     [ebp+var_key_length]
```

After the first two compare with 256 the first xor byte with memory address is the start of the encrypted data on the stack.

```
32 44 0D B0                             xor     al, byte ptr [ebp+ecx+var_data_enc]
```

Once we have found the data xor the first compare is the end of the RC4 decryption... also the string length.

```
83 F9 40                                cmp     ecx, 40h ; '@'
0F 82 22 FF FF FF                       jb      loc_408E37
```

Decryption

We are going to use a custom emulator that only handles memory operations no control flow. Using the above rules we will implement "dump points" for the key, and encrypted data.

This idea and tool is based on the initial "memulator" concept from @mishap pxor_string_decrypt_wip.py.

```
import pefile
import re
from typing import List
from capstone import *
from capstone.x86 import *

from dataclasses import dataclass

@dataclass
class MemorySegment:
    address: int
    size: int
    data: bytes



class ProcessMemory:
    memory: List[MemorySegment]

    def __init__(self):
        self.memory = []

    def _section_align(self, size: int) -> int:
        return ((size + 0x1000 - 1) // 0x1000) * 0x1000

    def allocate(self, address, size):
        # Check if the memory is already allocated or if it overlaps with another
allocation
        for m in self.memory:
            if m.address <= address < m.address + m.size:
                raise Exception(f"Memory already allocated: {hex(address)}")
            if address <= m.address < address + size:
                raise Exception(f"Memory already allocated: {hex(address)}")
        # Align the size
        size = self._section_align(size)
        # Allocate the memory
        self.memory.append(MemorySegment(address, size, b'\x00' * size))
        # Return the address and new size
        return address, size

    def read(self, address, size):
        # Get the memory segment
        for m in self.memory:
            if m.address <= address < m.address + m.size:
                break
        else:
            raise Exception(f"Memory not allocated: {hex(address)}")
        # Check if we are reading out of bounds
        if address + size > m.address + m.size:
            raise Exception(f"Reading out of bounds: {hex(address)}")
        # Return the data
        return m.data[address - m.address:address - m.address + size]
    def write(self, address, data):
        # Get the memory segment
        for m in self.memory:
            if m.address <= address < m.address + m.size:
                break
        else:
            raise Exception(f"Memory not allocated: {hex(address)}")
        # Check if we are writing out of bounds
        if address + len(data) > m.address + m.size:
            raise Exception(f"Writing out of bounds: {hex(address)}")
        # Write the data
        m.data = m.data[:address - m.address] + data + m.data[address - m.address +
len(data):]
        # Return number of bytes written
        return len(data)

    def dump_section(self, address):
        # Get the memory segment
        for m in self.memory:
            if m.address <= address < m.address + m.size:
                break
        else:
            raise Exception(f"Memory not allocated: {hex(address)}")
        # Return the data
        return m.data

    def clear_section(self, address):
        # Get the memory segment
        for m in self.memory:
            if m.address <= address < m.address + m.size:
                break
        else:
            raise Exception(f"Memory not allocated: {hex(address)}")
        # Clear the data
        m.data = b'\x00' * m.size


ADDRESS_MASK = 0xFFFFFFFF

def op_mem(instr, op, *, aligned=False):
    mem_address = 0
    base = op.mem.base
    if base != X86_REG_INVALID:
        name = base
        value = regs[name]
        mem_address += value

    index = op.mem.index
    if index != X86_REG_INVALID:
        name = index
        value = regs[name]
        mem_address += value * op.mem.scale
    disp = op.mem.disp # TODO: negative value handling?
    mem_address += disp
    mem_address &= ADDRESS_MASK
    if aligned:
        alignment = op.size
        if mem_address & (alignment - 1) != 0:
            assert False, f"Address {hex(mem_address)} not aligned to {alignment}"
    return mem_address

def op_read(instr, index, *, aligned=False):
    op: X86Op = instr.operands[index]
    if op.type == CS_OP_REG:
        name = op.value.reg
        # Handle low high and word registers
        if name == X86_REG_AL:
            value = regs[X86_REG_EAX] & 0xFF
        elif name == X86_REG_AH:
            value = (regs[X86_REG_EAX] >> 8) & 0xFF
        elif name == X86_REG_AX:
            value = regs[X86_REG_EAX] & 0xFFFF
        elif name == X86_REG_BL:
            value = regs[X86_REG_EBX] & 0xFF
        elif name == X86_REG_BH:
            value = (regs[X86_REG_EBX] >> 8) & 0xFF
        elif name == X86_REG_BX:
            value = regs[X86_REG_EBX] & 0xFFFF
        elif name == X86_REG_CL:
            value = regs[X86_REG_ECX] & 0xFF
        elif name == X86_REG_CH:
            value = (regs[X86_REG_ECX] >> 8) & 0xFF
        elif name == X86_REG_CX:
            value = regs[X86_REG_ECX] & 0xFFFF
        elif name == X86_REG_DL:
            value = regs[X86_REG_EDX] & 0xFF
        elif name == X86_REG_DH:
            value = (regs[X86_REG_EDX] >> 8) & 0xFF
        elif name == X86_REG_DX:
            value = regs[X86_REG_EDX] & 0xFFFF
        elif name == X86_REG_SI:
            value = regs[X86_REG_ESI] & 0xFFFF
        elif name == X86_REG_DI:
            value = regs[X86_REG_EDI] & 0xFFFF
        elif name == X86_REG_BP:
            value = regs[X86_REG_EBP] & 0xFFFF
        elif name == X86_REG_SP:
            value = regs[X86_REG_ESP] & 0xFFFF
        else:
            value = regs[name]
        print(f"\t\t\t\tReading {hex(value)} from reg {name}:{instr.reg_name(name)}")
        return value
    elif op.type == CS_OP_MEM:
        mem_address = op_mem(instr, op, aligned=aligned)
        data = memory.read(mem_address, op.size)
        return int.from_bytes(data, "little")
    elif op.type == CS_OP_IMM:
        # TODO: sign extend?
        return op.value.imm
    else:
        raise NotImplementedError()

def op_write(instr, index, value, *, aligned=False):
    op: X86Op = instr.operands[index]
    size = op.size
    if op.type == CS_OP_REG:
        name = op.value.reg
        print(f"\t\t\t\tWriting {hex(value)} to reg {name}:{instr.reg_name(name)}")
        # Handle low high and word registers
        if name == X86_REG_AL:
            regs[X86_REG_EAX] &= 0xFFFFFF00
            regs[X86_REG_EAX] |= value & 0xFF
        elif name == X86_REG_AH:
            regs[X86_REG_EAX] &= 0xFFFF00FF
            regs[X86_REG_EAX] |= (value & 0xFF) << 8
        elif name == X86_REG_AX:
            regs[X86_REG_EAX] &= 0xFFFF0000
            regs[X86_REG_EAX] |= value & 0xFFFF
        elif name == X86_REG_BL:
            regs[X86_REG_EBX] &= 0xFFFFFF00
            regs[X86_REG_EBX] |= value & 0xFF
        elif name == X86_REG_BH:
            regs[X86_REG_EBX] &= 0xFFFF00FF
            regs[X86_REG_EBX] |= (value & 0xFF) << 8
        elif name == X86_REG_BX:
            regs[X86_REG_EBX] &= 0xFFFF0000
            regs[X86_REG_EBX] |= value & 0xFFFF
        elif name == X86_REG_CL:
            regs[X86_REG_ECX] &= 0xFFFFFF00
            regs[X86_REG_ECX] |= value & 0xFF
        elif name == X86_REG_CH:
            regs[X86_REG_ECX] &= 0xFFFF00FF
            regs[X86_REG_ECX] |= (value & 0xFF) << 8
        elif name == X86_REG_CX:
            regs[X86_REG_ECX] &= 0xFFFF0000
            regs[X86_REG_ECX] |= value & 0xFFFF
        elif name == X86_REG_DL:
            regs[X86_REG_EDX] &= 0xFFFFFF00
            regs[X86_REG_EDX] |= value & 0xFF
        elif name == X86_REG_DH:
            regs[X86_REG_EDX] &= 0xFFFF00FF
            regs[X86_REG_EDX] |= (value & 0xFF) << 8
        elif name == X86_REG_DX:
            regs[X86_REG_EDX] &= 0xFFFF0000
            regs[X86_REG_EDX] |= value & 0xFFFF
        elif name == X86_REG_SI:
            regs[X86_REG_ESI] &= 0xFFFF0000
            regs[X86_REG_ESI] |= value & 0xFFFF
        elif name == X86_REG_DI:
            regs[X86_REG_EDI] &= 0xFFFF0000
            regs[X86_REG_EDI] |= value & 0xFFFF
        elif name == X86_REG_BP:
            regs[X86_REG_EBP] &= 0xFFFF0000
            regs[X86_REG_EBP] |= value & 0xFFFF
        elif name == X86_REG_SP:
            regs[X86_REG_ESP] &= 0xFFFF0000
            regs[X86_REG_ESP] |= value & 0xFFFF
        else:
            regs[name] = value


    elif op.type == CS_OP_MEM:
        mem_address = op_mem(instr, op, aligned=aligned)
        data = value.to_bytes(size, "little")
        # TODO: handle invalid memory access
        memory.write(mem_address, data)
    else:
        raise NotImplementedError()

def print_stack(ptr, size):
    # Print the stack -255  bytes with addresses in hex as DWORDS
    for i in range(0, 4*6 + size, 4):
        address = ptr - size + i
        data = memory.read(address, 4)
        value = int.from_bytes(data, "little")
        if address == ptr:
            print(f"{address:08x} {value:08x} <--- POINTER")
        else:
            print(f"{address:08x} {value:08x}")



def rc4(data, key):
    S = list(range(256))
    j = 0
    out = b''
    # KSA Phase
    for i in range(256):
        j = (j + S[i] + key[i % len(key)]) % 256
        S[i], S[j] = S[j], S[i]
    # PRGA Phase
    i = j = 0
    for char in data:
        i = (i + 1) % 256
        j = (j + S[i]) % 256
        S[i], S[j] = S[j], S[i] # swap
        out += bytes([char ^ S[(S[i] + S[j]) % 256]])
    return out
```

```
#file_data = open('/tmp/FakeSearchProtocolHost.bin', 'rb').read()
file_data =
open('/tmp/pika_known/a79c4a29075098abda0558c40cfc2250ab3dbae6598b7b967a43856532cbce05
 'rb').read()
pe = pefile.PE(data=file_data)

section_data = None
entry_point = pe.OPTIONAL_HEADER.AddressOfEntryPoint
pe_base = pe.OPTIONAL_HEADER.ImageBase


memory = ProcessMemory()

# Allocate memory for each of the sections and write them to the memory
for section in pe.sections:
    address = pe_base + section.VirtualAddress
    size = section.Misc_VirtualSize
    data = section.get_data()
    memory.allocate(address, size)
    memory.write(address, data)


# Allocate memory for the stack
stack_address = 0x1000
stack_size = 0x200000
memory.allocate(stack_address, stack_size)


# Scan all instructions
# For ret reset the stack and set the ESP/EBP registers
# When we hit cmp 0x100 test the strings on the stack
test_start = 0x00408C55
test_end = 0x004090EE

test_start = 0x004020A0
test_end = 0x0040458D

# Run for full code section
test_start = pe_base + pe.sections[0].VirtualAddress
test_end = test_start + pe.sections[0].Misc_VirtualSize

code = memory.read(test_start, test_end - test_start)

cs = Cs(CS_ARCH_X86, CS_MODE_32)
cs.detail = True
cs.skipdata = True

regs = [0]*X86_REG_ENDING

regs[X86_REG_ESP] = stack_address + stack_size//2
```

```
regs[X86_REG_EBP] = stack_address + stack_size//2

string_start = None
keys = []
flag_watch_cmp = False
flag_enter_key_loop = False
flag_enter_rc4 = False
string_start_candidate = None
key_flag = False
tmp_key_addrs = []
tmp_keys = []
key_len = 0

out_strings = {}

for instr in cs.disasm(code, test_start):
    esp = regs[X86_REG_ESP]
    ebp = regs[X86_REG_EBP]
    print(f"{instr.address:08x} ({hex(esp - ebp)}) \t{instr.mnemonic}
{instr.op_str}")
    try:
        if instr.id == X86_INS_MOV:
            #print(f"\tMOV")
            # Move from operand 1 to operand 0
            value = op_read(instr, 1)
            print(f"\t\t\t\tMoving {hex(value)}")
            op_write(instr, 0, value)
            if key_flag:
                    # If the second operand is an address on the stack
                    if instr.operands[1].type == CS_OP_MEM:
                        key_start = op_mem(instr, instr.operands[1])
                        print(f"\t\t\t\tPossible key start: {hex(key_start)}")
                        tmp_key_addrs.append(key_start)
                        key_flag = False

            if flag_enter_key_loop:
                # If the size is 1 byte and the second operand is a memory address
                if instr.operands[0].size == 1 and instr.operands[1].type ==
CS_OP_MEM:
                    string_start_candidate = op_mem(instr, instr.operands[1])

        elif instr.id == X86_INS_MOVZX:
            #print(f"\tMOVZX")
            try:
                if key_flag:
                    # If the second operand is a memory address
                    if instr.operands[1].type == CS_OP_MEM:
                        key_start = op_mem(instr, instr.operands[1])
                        print(f"\t\t\t\tPossible key start: {hex(key_start)}")
                        tmp_key_addrs.append(key_start)
                        key_flag = False
                # Move from operand 1 to operand 0 with zero extension
                value = op_read(instr, 1)
                op_write(instr, 0, value)
            except:
                pass

        elif instr.id == X86_INS_DIV:
            #print(f"\tDIV")
            # If operand 0 is memory
            if flag_enter_rc4 and instr.operands[0].type == CS_OP_MEM:
                key_flag = True
                value0 = op_read(instr, 0)
                key_len = value0
                print(f"\t\t\t\tKey length: {hex(key_len)}")

        elif instr.id == X86_INS_AND:
            #print(f"\tAND")
            # AND operand 0 and operand 1
            value0 = op_read(instr, 0)
            value1 = op_read(instr, 1)
            value = value0 & value1
            op_write(instr, 0, value)

        elif instr.id == X86_INS_OR:
            #print(f"\tOR")
            # OR operand 0 and operand 1
            value0 = op_read(instr, 0)
            value1 = op_read(instr, 1)
            value = value0 | value1
            op_write(instr, 0, value)

        elif instr.id == X86_INS_ADD:
            #print(f"\tADD")
            # Add operand 0 and operand 1
            value0 = op_read(instr, 0)
            value1 = op_read(instr, 1)
            value = (value0 + value1) & ADDRESS_MASK
            op_write(instr, 0, value)

        elif instr.id == X86_INS_SUB:
            #print(f"\tSUB")
            # Subtract operand 1 from operand 0
            value0 = op_read(instr, 0)
            value1 = op_read(instr, 1)
            value = (value0 - value1) & ADDRESS_MASK
            op_write(instr, 0, value)

        elif instr.id == X86_INS_MOVSB:
            #print(f"\tMOVSB")
            # Read byte from DS:ESI and write to ES:EDI
            value = memory.read(regs[X86_REG_ESI], 1)
            memory.write(regs[X86_REG_EDI], value)
            # Increment ESI and EDI
```

```
            regs[X86_REG_ESI] += 1
            regs[X86_REG_EDI] += 1

        elif instr.id == X86_INS_MOVSW:
            #print(f"\tMOVSW")
            # Read word from DS:ESI and write to ES:EDI
            value = memory.read(regs[X86_REG_ESI], 2)
            memory.write(regs[X86_REG_EDI], value)
            # Increment ESI and EDI
            regs[X86_REG_ESI] += 2
            regs[X86_REG_EDI] += 2

        elif instr.id == X86_INS_MOVSD:
            #print(f"\tMOVSD")
            # Read byte by byte from memeory at DS:ESI until null byte
            out = b''
            for i in range(0, 256):
                value = memory.read(regs[X86_REG_ESI] + i, 1)
                if value == b'\x00':
                    break
                out += value
            if out.isascii() and len(out) > 4:
                print(f"\t\t\t\tPotential key: {out}")
                keys.append(out)

            # Read dword from DS:ESI and write to ES:EDI
            value = memory.read(regs[X86_REG_ESI], 4)
            memory.write(regs[X86_REG_EDI], value)
            # Increment ESI and EDI
            regs[X86_REG_ESI] += 4
            regs[X86_REG_EDI] += 4

        elif instr.id == X86_INS_LEA:
            #print(f"\tLEA")
            # Load effective address from operand 1 to operand 0
            value = op_mem(instr, instr.operands[1])
            op_write(instr, 0, value)

        elif instr.id == X86_INS_PUSH:
            #print(f"\tPUSH")
            # Push operand 0
            value = op_read(instr, 0)
            size = instr.operands[0].size
            regs[X86_REG_ESP] -= size
            # Write value to stack and decrement ESP
            print(f"\t\t\t\tPushing {hex(value)}")
            memory.write(regs[X86_REG_ESP], value.to_bytes(size, "little"))


        elif instr.id == X86_INS_POP:
            #print(f"\tPOP")
            # Read value from stack and increment ESP
            size = instr.operands[0].size
            value_data = memory.read(regs[X86_REG_ESP], size)
            value = int.from_bytes(value_data, "little")
            regs[X86_REG_ESP] += size
            # Write value to operand 0
            print(f"\t\t\t\tPopping {hex(value)}")
            op_write(instr, 0, value)

        elif instr.id == X86_INS_CMP:
            #print(f"\tCMP")
            # Compare operand 0 and operand 1
            value0 = op_read(instr, 0)
            value1 = op_read(instr, 1)
            # If watch flag is set and
            if flag_watch_cmp:
                print(f"\t\t\t\tCMP {hex(value1)}")
                # This is the end of our string decryption loop
                # Read string from stack
                string_data = memory.read(string_start, value1)
                print(f"\t\t\t\tString: {string_data.hex()}")
                # Try to decrypt the string with all the keys
                out_string = None
                for key in keys:
                    try:
                        decrypted = rc4(string_data, key)
                        print(f"\t\t\t\tTEST: {decrypted}")
                        if decrypted.isascii():
                            print(f"\t\t\t\tDecrypted: {decrypted}")
                            out_string = decrypted
                            break
                    except:
                        pass
                if out_string is None:
                    # Try with tmp keys
                    print("\t\t\t\tNo strings found attempting with tmp keys")
                    for key in tmp_keys:
                        try:
                            decrypted = rc4(string_data, key)
                            print(f"\t\t\t\tTEST: {decrypted}")
                            if decrypted.isascii():
                                print(f"\t\t\t\tDecrypted: {decrypted}")
                                out_string = decrypted
                                break
                        except:
                            pass
                if out_string is None:
                    # Try with candidate string instead
                    print("\t\t\t\tNo strings found attempting with candidate")
                    # This is the end of our string decryption loop
                    # Read string from stack
                    string_data = memory.read(string_start_candidate, value1)
                    print(f"\t\t\t\tString: {string_data.hex()}")
            # Try to decrypt the string with all the keys
            out_string = None
            for key in keys:
                try:
                    decrypted = rc4(string_data, key)
                    print(f"\t\t\t\tTEST: {decrypted}")
                    if decrypted.isascii():
                        print(f"\t\t\t\tDecrypted: {decrypted}")
                        out_string = decrypted
                        break
                except:
                    pass
        if out_string is None:
            # Try with tmp keys
            print("\t\t\t\tNo strings found attempting with tmp keys")
            # This is the end of our string decryption loop
            # Read string from stack
            string_data = memory.read(string_start_candidate, value1)
            print(f"\t\t\t\tString: {string_data.hex()}")
            # Try to decrypt the string with all the keys
            out_string = None
            for key in tmp_keys:
                try:
                    decrypted = rc4(string_data, key)
                    print(f"\t\t\t\tTEST: {decrypted}")
                    if decrypted.isascii():
                        print(f"\t\t\t\tDecrypted: {decrypted}")
                        out_string = decrypted
                        break
                except:
                    pass
        if out_string is not None:
            out_strings[instr.address] = out_string

        # Reset the flags and tmp_keys but keep the keys
        flag_watch_cmp = False
        flag_enter_key_loop = False
        flag_enter_rc4 = False
        string_start_candidate = None
        key_flag = False
        tmp_keys = []
        key_len = 0
        tmp_key_addrs = []

        print("\t\t\t\tEND for RC4 - Reset flags")
elif value1 == 0x100 and not flag_enter_rc4:
    flag_enter_rc4 = True
    print("\t\t\t\tSTART for RC4 - Set flag")
elif value1 == 0x100 and flag_enter_rc4:
    flag_enter_key_loop = True
    print("\t\t\t\tSTART for key loop - Scanning for XOR")
    # For each tmp key canidate attempt to read it
```

```
                for key_addr in tmp_key_addrs:
                    try:
                        key_data = memory.read(key_addr, key_len)
                        print(f"\t\t\t\tTmp Key: {key_data.hex()}")
                        if key_data.isascii() and b'\x00\x00' not in key_data:
                            print(f"\t\t\t\tAdding tmp key: {key_data}")
                            tmp_keys.append(key_data)
                    except:
                        pass
            elif value1 == 0x100 and flag_enter_key_loop:
                raise Exception("\t\t\t\tToo many cmp 0x100")


        elif instr.id == X86_INS_INC:
            #print(f"\tINC")
            # Increment operand 0
            value = op_read(instr, 0)
            value += 1
            op_write(instr, 0, value)


        elif instr.id == X86_INS_XOR:
            #print(f"\tXOR")
            # If operand 0 is a single byte and operand 1 is on the stack this is the
string xor
            if flag_enter_key_loop and instr.operands[0].size == 1 and
instr.operands[1].type == CS_OP_MEM:
                print(f"\t\t\t\tECX: {hex(regs[X86_REG_ECX])}")
                print(f"\t\t\t\tEBP: {hex(regs[X86_REG_EBP])}")
                # Read the string from the stack
                string_address = op_mem(instr, instr.operands[1])
                # If string address is on the stack break
                if stack_address <= string_address < stack_address + stack_size:
                    print(f"\t\t\t\tString address: {hex(string_address)}")
                    string_start = string_address
                    flag_watch_cmp = True
            # Otherwise xor operand 0 and operand 1
            value0 = op_read(instr, 0)
            value1 = op_read(instr, 1)
            value = value0 ^ value1
            print(f"\t\t\t\tXOR {value0} {value1} = {value}")
            op_write(instr, 0, value)


        elif instr.id == X86_INS_RET or instr.id == X86_INS_RETF or instr.id ==
X86_INS_IRET or instr.id == X86_INS_IRETD or instr.id == X86_INS_IRETQ:
            #print(f"\tRET")
            # Clear registers
            regs = [0]*X86_REG_ENDING
            # Clear stack memory
            memory.clear_section(stack_address)
            # Reset the stack
            regs[X86_REG_ESP] = stack_address + stack_size//2
            regs[X86_REG_EBP] = stack_address + stack_size//2
            # Reset everything
            flag_watch_cmp = False
            flag_enter_key_loop = False
            flag_enter_rc4 = False
            string_start_candidate = None
            key_flag = False
            tmp_keys = []
            key_len = 0
            tmp_key_addrs = []


        else:
            print(f"\t\t\t\tUnknown instruction: {instr.mnemonic} {instr.op_str}")
    except Exception as e:
        print(f"ERROR: {e}")

# Print the strings
print("Strings:")
for addr,string in out_strings.items():
    print(f"{hex(addr)}: {string}")
```

```
from Crypto.Cipher import AES
import base64

def printable_ascii(data):
    for char in data:
        if (char < 0x20 or char > 0x7E) and char != 0x0A and char != 0x0D and char !=
0x09:
            return False
    return True


# Deduplicate the strings to a temporary set
tmp_strings = set()
for _,string in out_strings.items():
    tmp_strings.add(string)

# Get the longest string in the set
longest_string = None
for string in tmp_strings:
    if longest_string is None or len(string) > len(longest_string):
        longest_string = string

candidate = base64.b64decode(longest_string.replace(b'_', b'='))

# Brute force the AES key and IV by attempting to to decrypt the longest string with
all possible combinations of the first 16 bytes of a string and the first 32 bytes of
a string
end_flag = False
for tmp_key in tmp_strings:
    if end_flag:
        break
    for tmp_iv in tmp_strings:
            try:
                key = tmp_key[:32]
                iv = tmp_iv[:16]
                # Try to decrypt the string with the key and IV
                cipher = AES.new(key, AES.MODE_CBC, iv)
                decrypted = cipher.decrypt(candidate)
                # Remove padding if it exists
                if decrypted[-1] <= 0x10:
                    decrypted = decrypted[:-decrypted[-1]]

                # If the decrypted string is ascii print it
                if decrypted != b'' and printable_ascii(decrypted):
                    print(f"Key string: {tmp_key}")
                    print(f"Key: {key.hex()}")
                    print(f"IV string: {tmp_iv}")
                    print(f"IV: {iv.hex()}")
                    print(f"Decrypted: {decrypted}")
                    end_flag = True
                    break
            except:
                pass
rejects = []

final_strings = []

# Decrypt all strings with the key and IV
for addr,string in out_strings.items():
    try:
        # Convert from address to offset
        offset = pe.get_offset_from_rva(addr - pe_base)

        cipher = AES.new(key, AES.MODE_CBC, iv)
        decrypted = cipher.decrypt(base64.b64decode(string.replace(b'_', b'=')))
        # Remove padding if it exists
        if decrypted[-1] <= 0x10:
            decrypted = decrypted[:-decrypted[-1]]
        # If the decrypted string is ascii print it
        if decrypted != b'' and printable_ascii(decrypted):
            print(f"Decrypted: {decrypted}")
            final_strings.append({
              "offset":offset,
              "value":decrypted.decode("ascii"),
            })
        else:
            rejects.append(string)
    except:
        rejects.append(string)

# Print the rejected strings
print("Rejected:")
for string in rejects:
    print(string)

str_dict = {"strings": final_strings}
print(str_dict)
import json
# Write the strings to a json file
with open('strings.json', 'w') as outfile:
    json.dump(str_dict, outfile)
```

```
Key string: b'l9SpFBoXEyglbY0ginoTUBd=pP=y6rVcQG8tP/zV4iqr06yZKEb+VCg1yQJ5jUNE'
Key: 6c39537046426f584579676c62593067696e6f545542643d70503d7936725663
IV string: b'FdbAwsDj0FJcgkLPb1J/mqGU7T6e98p9CMnoB'
IV: 466462417773446a30464a63676b4c50
Decrypted: b'{"mdPNC6f8": "%s", "NUn3h77h": "%s", "W381C": "Win %d.%d %d",
"SJ3sWSeKQ": %s, "YlSwktC": "%s", "7smbRjIVo": "%s", "AU3KNl": "%s", "hgcPNL3D": %d,
"RMe4xUeMl": "%s", "K92GpjSw": %d, "04FjIN": "%s", "yVqTFN": "%s", "TrKEIz": "%s",
"gh5jjQAh": "%s", "EoLYU": "%s", "gBmbRM40z": %d, "NUqeH": %d}'
Decrypted: b'CreateMutexW'
Decrypted: b'GetLastError'
Decrypted: b'%s'
Decrypted: b'wsprintfA'
Decrypted: b'&'
Decrypted: b'&tfDgx='
Decrypted: b'whoami.exe /all'
Decrypted: b'&M1LWU='
Decrypted: b'ipconfig.exe /all'
Decrypted: b'&VC76f='
Decrypted: b'netstat.exe -aon'
Decrypted: b'&SBSlO='
Decrypted: b'{"mdPNC6f8": "%s"}'
Decrypted: b'wsprintfA'
Decrypted: b'{"mdPNC6f8": "%s"}'
Decrypted: b'wsprintfA'
Decrypted: b'HydrohemothoraxCoenaesthesis/2bQbdHQI1z9PoD?
SnarlishAllobars=59eYpYysBS&UndoubtableEthnologically=BicentenaryPensil&Superstylishly

Decrypted: b'&'
Decrypted: b'BaylZ'
Decrypted: b'AV89JS'
Decrypted: b'IsWow64Process'
Decrypted: b'GetProductInfo'
Decrypted: b'%d'
Decrypted: b'wsprintfW'
Decrypted: b'unknown'
Decrypted: b'GetComputerNameW'
Decrypted: b'unknown'
Decrypted: b'GetComputerNameExW'
Decrypted: b'unknown'
Decrypted: b'DsGetDcNameW'
Decrypted: b'unknown'
Decrypted: b'EnumDisplayDevicesW'
Decrypted: b'GlobalMemoryStatusEx'
Decrypted: b'GetDesktopWindow'
Decrypted: b'GetWindowRect'
Decrypted: b'%dx%d'
Decrypted: b'wsprintfW'
Decrypted: b'unknown'
Decrypted: b'GetTickCount'
Decrypted: b'OpenProcessToken'
Decrypted: b'GetCurrentProcess'
Decrypted: b'GetTokenInformation'
```

```
Decrypted: b'Kernel32.dll'
Decrypted: b'User32.dll'
Decrypted: b'Wininet.dll'
Decrypted: b'Advapi32.dll'
Decrypted: b'NetApi32.dll'
Decrypted: b'MultiByteToWideChar'
Decrypted: b'WaitForSingleObjectEx'
Decrypted: b'GetTickCount'
Decrypted: b'%s&%s'
Decrypted:
b'UndoubtableEthnologically=antitwilightFluidextract&birefractingUndeceitfulness=huehu

Decrypted: b'UdvGU='
Decrypted: b'wsprintfA'
Decrypted: b'POST'
Decrypted: b'%s&%s'
Decrypted:
b'UndoubtableEthnologically=antitwilightFluidextract&birefractingUndeceitfulness=huehu

Decrypted: b'UdvGU='
Decrypted: b'wsprintfA'
Decrypted: b'POST'
Decrypted: b'{"mdPNC6f8": "%s", "MsDkQb2T": %s, "jVeNAqf": %d, "5ScPjT": "'
Decrypted: b'"}'
Decrypted: b'wsprintfA'
Decrypted: b'fabledOverstridence/h31BYUqJ28W62tz?
nonresister=jYnT8Hj13x&Sixtine=TXEWWGZ&DogvaneHeredia=teAiVe'
Decrypted: b'InternetOpenW'
Decrypted: b'&'
Decrypted: b'HttpOpenRequestW'
Decrypted: b'InternetQueryOptionW'
Decrypted: b'Content-Type: application/x-www-form-urlencoded\r\nAccept:
*/*\r\nAccept-Language: en-US,en;q=0.5\r\nAccept-Encoding: gzip, deflate\r\nUser-
Agent: Mozilla/5.0 (compatible; MSIE 5.0; Windows NT 5.01; Trident/5.1)\r\n'
Decrypted: b'lstrlenW'
Decrypted: b'lstrlenA'
Decrypted: b'HttpSendRequestW'
Decrypted: b'InternetReadFile'
Decrypted: b'InternetCloseHandle'
Decrypted: b'InternetCloseHandle'
Decrypted: b'InternetCloseHandle'
Decrypted: b'InternetCloseHandle'
Decrypted: b'InternetCloseHandle'
Decrypted: b'InternetSetOptionW'
Decrypted: b'InternetOpenW'
Decrypted: b'&'
Decrypted: b'InternetConnectW'
Decrypted: b'HttpOpenRequestW'
Decrypted: b'InternetQueryOptionW'
Decrypted: b'Content-Type: application/x-www-form-urlencoded\r\nAccept:
*/*\r\nAccept-Language: en-US,en;q=0.5\r\nAccept-Encoding: gzip, deflate\r\nUser-
Agent: Mozilla/5.0 (compatible; MSIE 5.0; Windows NT 5.01; Trident/5.1)\r\n'
```

```
Decrypted: b'lstrlenA'
Decrypted: b'HttpSendRequestW'
Decrypted: b'InternetReadFile'
Decrypted: b'InternetCloseHandle'
Decrypted: b'InternetCloseHandle'
Decrypted: b'InternetCloseHandle'
Decrypted: b'InternetCloseHandle'
Decrypted: b'InternetCloseHandle'
Decrypted: b'InternetSetOptionW'
Decrypted: b'InternetCloseHandle'
Decrypted: b'InternetCloseHandle'
Decrypted: b'InternetCloseHandle'
Decrypted: b'InternetReadFile'
Decrypted: b'RegCreateKeyExW'
Decrypted: b'RegSetValueExW'
Decrypted: b'RegCloseKey'
Decrypted: b'RegOpenKeyExW'
Decrypted: b'RegQueryValueExW'
Decrypted: b'RegCloseKey'
Decrypted: b'RegCloseKey'
Decrypted: b'C:\\'
Decrypted: b'GetVolumeInformationW'
Decrypted: b'%s\\%s|%s'
Decrypted: b'wsprintfW'
Decrypted: b'%07lX%09lX%lu'
Decrypted: b'wsprintfW'
Decrypted: b'GetUserDefaultLangID'
Decrypted: b'%appdata%\\Microsoft\\'
Decrypted: b'lotterSig'
Decrypted: b'\\'
Decrypted: b'ExpandEnvironmentStringsW'
Decrypted: b'GetFileAttributesW'
Decrypted: b'Synanthic'
Decrypted: b'.dll'
Decrypted: b'.exe'
Decrypted: b'CreateFileW'
Decrypted: b'WriteFile'
Decrypted: b'CloseHandle'
Decrypted: b'CreateDirectoryW'
Decrypted: b'&'
Decrypted: b'SOFTWARE\\Microsoft\\%s'
Decrypted: b'lotterSig'
Decrypted: b'Subadmini'
Decrypted: b'wsprintfW'
Decrypted: b'{"mdPNC6f8": "%s", "NUn3h77h": "%s", "W381C": "Win %d.%d %d",
"SJ3sWSeKQ": %s, "YlSwktC": "%s", "7smbRjIVo": "%s", "AU3KNl": "%s", "hgcPNL3D": %d,
"RMe4xUeMl": "%s", "K92GpjSw": %d, "04FjIN": "%s", "yVqTFN": "%s", "TrKEIz": "%s",
"gh5jjQAh": "%s", "EoLYU": "%s", "gBmbRM40z": %d, "NUqeH": %d}'
Decrypted: b'GG9TU@T@f0adda360d2b4ccda11468e026526576'
Decrypted: b'wsprintfW'
Decrypted: b'AV89JS'
Decrypted: b'TrichinopolyUncontriving/uiDV6mKfgGakdg?unshelledSplitnut=vEzlHkL'
```

```
Decrypted: b'&'
Decrypted: b'SOFTWARE\\Microsoft\\%s'
Decrypted: b'lotterSig'
Decrypted: b'Subadmini'
Decrypted: b'wsprintfW'
Decrypted: b'CreateToolhelp32Snapshot'
Decrypted: b'Process32FirstW'
Decrypted: b'CloseHandle'
Decrypted: b'Process32NextW'
Decrypted: b'explorer.exe'
Decrypted: b'OpenProcess'
Decrypted: b'CloseHandle'
Decrypted: b'InitializeProcThreadAttributeList'
Decrypted: b'InitializeProcThreadAttributeList'
Decrypted: b'UpdateProcThreadAttribute'
Decrypted: b'DeleteProcThreadAttributeList'
Decrypted: b'NvtocV4e'
Decrypted: b'UpdateProcThreadAttribute'
Decrypted: b'InitializeProcThreadAttributeList'
Decrypted: b'InitializeProcThreadAttributeList'
Decrypted: b'UpdateProcThreadAttribute'
Decrypted: b'CreateProcessW'
Decrypted: b'DeleteProcThreadAttributeList'
Decrypted: b'UpdateProcThreadAttribute'
Decrypted: b'IsWow64Process'
Decrypted: b'CreateToolhelp32Snapshot'
Decrypted: b'Process32FirstW'
Decrypted: b'['
Decrypted: b'"%s:%d:%d:%d:%d:%d:%d"'
Decrypted: b', "%s:%d:%d:%d:%d:%d:%d"'
Decrypted: b']'
Decrypted: b'wsprintfW'
Decrypted: b'Process32NextW'
Decrypted: b'wsprintfW'
Decrypted: b'CloseHandle'
Decrypted: b'CloseHandle'
Decrypted: b'wsprintfW'
Decrypted: b'wsprintfW'
Decrypted: b'CreatePipe'
Decrypted: b'CreateProcessW'
Decrypted: b'CloseHandle'
Decrypted: b'CloseHandle'
Decrypted: b'CloseHandle'
Decrypted: b'CloseHandle'
Decrypted: b'CloseHandle'
Decrypted: b'CloseHandle'
Decrypted: b'WaitForSingleObject'
Decrypted: b'PeekNamedPipe'
Decrypted: b'ReadFile'
Decrypted: b'CloseHandle'
Decrypted: b'CloseHandle'
Decrypted: b'CloseHandle'
```

```
Decrypted: b'&'
Decrypted: b'NvtocV4e'
Decrypted: b'{"mdPNC6f8": "%s", "isjuuMr": "%s", "MsDkQb2T": %s}'
Decrypted: b'wsprintfA'
Decrypted: b'nanoinstructionFrisesomorum/XjqtQzQyycNZVoIQ?
unapplicability=i73MV07GwaCH13Q'
Decrypted: b'BaylZ'
Decrypted: b'ExpandEnvironmentStringsW'
Decrypted: b'&'
Decrypted: b'{"mdPNC6f8": "%s", "isjuuMr": "%s", "MsDkQb2T": %s}'
Decrypted: b'wsprintfA'
Decrypted: b'nanoinstructionFrisesomorum/XjqtQzQyycNZVoIQ?
unapplicability=i73MV07GwaCH13Q'
Decrypted: b'BaylZ'
Decrypted: b'ExpandEnvironmentStringsW'
Decrypted: b'{"mdPNC6f8": "%s", "isjuuMr": "%s", "MsDkQb2T": %s}'
Decrypted: b'wsprintfA'
Decrypted: b'nanoinstructionFrisesomorum/XjqtQzQyycNZVoIQ?
unapplicability=i73MV07GwaCH13Q'
Decrypted: b'&'
Decrypted: b'qqmyS'
Decrypted: b'tK5nVvwh'
Decrypted: b'mGTYP'
Decrypted: b'2bjHya'
Decrypted: b'whoami.exe /all'
Decrypted: b'ipconfig.exe /all'
Decrypted: b'netstat.exe -aon'
Decrypted: b'&'
Decrypted: b'dLmghDRe'
Decrypted: b'ExitProcess'
Decrypted: b'&'
Decrypted: b'&'
Decrypted: b'bustlingly/e9vliMRRWKSd?
DeediestBromes=awIAh8S&bonaght=5vh1psTtP2mk9&stiltyKetohexose=jcAKtvLned5dp8'
Decrypted: b'Software\\Microsoft\\Windows\\CurrentVersion\\Run'
Decrypted: b'Synanthic'
Decrypted: b'rundll32'
Decrypted: b'.dll'
Decrypted: b'wsprintfW'
Decrypted: b'.exe'
Decrypted: b'wsprintfW'
Decrypted: b'&'
Decrypted: b'HxTPXf'
Decrypted: b'yAJsnWxR'
Decrypted: b'4qRRArO'
Decrypted: b'NvtocV4e'
Decrypted: b'qo9g3J'
Decrypted: b'GhPTR'
Decrypted: b'dLmghDRe'
Decrypted: b'9PpreQMX'
Decrypted: b'pKW7fqi2'
Rejected:
```

```
b'{F542086F-F5EF-48C4-8B12-49ED805B0205}'
b'0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz/=+'
b'l9SpFBoXEyglbY0ginoTUBd=pP=y6rVcQG8tP/zV4iqr06yZKEb+VCg1yQJ5jUNE'
b'FdbAwsDj0FJcgkLPb1J/mqGU7T6e98p9CMnoB'
b'\x19<jcy]X\r]'
b'1.1.15-ghost'
b'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/'
b'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/'
b'%s "%s%s%s", %s'
b'lJK\x03\x190'
```

{'strings': [{'offset': 5621, 'value': 'CreateMutexW'}, {'offset': 6224, 'value': 'GetLastError'}, {'offset': 6615, 'value': '%s'}, {'offset': 6895, 'value': 'wsprintfA'}, {'offset': 7710, 'value': '&'}, {'offset': 7994, 'value': '&tfDgx='}, {'offset': 8360, 'value': 'whoami.exe /all'}, {'offset': 8623, 'value': '&M1LWU='}, {'offset': 8954, 'value': 'ipconfig.exe /all'}, {'offset': 9355, 'value': '&VC76f='}, {'offset': 9701, 'value': 'netstat.exe -aon'}, {'offset': 10049, 'value': '&SBSlO='}, {'offset': 10521, 'value': '{"mdPNC6f8": "%s"}'}, {'offset': 10800, 'value': 'wsprintfA'}, {'offset': 12213, 'value': '{"mdPNC6f8": "%s"}'}, {'offset': 12584, 'value': 'wsprintfA'}, {'offset': 13364, 'value': 'HydrohemothoraxCoenaesthesis/2bQbdHQI1z9PoD?
SnarlishAllobars=59eYpYysBS&UndoubtableEthnologically=BicentenaryPensil&Superstylishly
 {'offset': 13875, 'value': '&'}, {'offset': 14197, 'value': 'BaylZ'}, {'offset': 14553, 'value': 'AV89JS'}, {'offset': 16191, 'value': 'IsWow64Process'}, {'offset': 16637, 'value': 'GetProductInfo'}, {'offset': 16976, 'value': '%d'}, {'offset': 17202, 'value': 'wsprintfW'}, {'offset': 17813, 'value': 'unknown'}, {'offset': 18333, 'value': 'GetComputerNameW'}, {'offset': 18747, 'value': 'unknown'}, {'offset': 19108, 'value': 'GetComputerNameExW'}, {'offset': 19547, 'value': 'unknown'}, {'offset': 19961, 'value': 'DsGetDcNameW'}, {'offset': 20258, 'value': 'unknown'}, {'offset': 21011, 'value': 'EnumDisplayDevicesW'}, {'offset': 21450, 'value': 'GlobalMemoryStatusEx'}, {'offset': 21836, 'value': 'GetDesktopWindow'}, {'offset': 22234, 'value': 'GetWindowRect'}, {'offset': 22712, 'value': '%dx%d'}, {'offset': 22951, 'value': 'wsprintfW'}, {'offset': 23411, 'value': 'unknown'}, {'offset': 24012, 'value': 'GetTickCount'}, {'offset': 24443, 'value': 'OpenProcessToken'}, {'offset': 24900, 'value': 'GetCurrentProcess'}, {'offset': 25176, 'value': 'GetTokenInformation'}, {'offset': 39521, 'value': 'Kernel32.dll'}, {'offset': 39881, 'value': 'User32.dll'}, {'offset': 40227, 'value': 'Wininet.dll'}, {'offset': 40505, 'value': 'Advapi32.dll'}, {'offset': 40840, 'value': 'NetApi32.dll'}, {'offset': 44196, 'value': 'MultiByteToWideChar'}, {'offset': 47472, 'value': 'WaitForSingleObjectEx'}, {'offset': 49139, 'value': 'GetTickCount'}, {'offset': 54486, 'value': '%s&%s'}, {'offset': 55066, 'value': 'UndoubtableEthnologically=antitwilightFluidextract&birefractingUndeceitfulness=huehue
 {'offset': 55323, 'value': 'UdvGU='}, {'offset': 55560, 'value': 'wsprintfA'}, {'offset': 56072, 'value': 'POST'}, {'offset': 56691, 'value': '%s&%s'}, {'offset': 57297, 'value': 'UndoubtableEthnologically=antitwilightFluidextract&birefractingUndeceitfulness=huehue
 {'offset': 57550, 'value': 'UdvGU='}, {'offset': 57927, 'value': 'wsprintfA'}, {'offset': 58565, 'value': 'POST'}, {'offset': 59043, 'value': '{"mdPNC6f8": "%s", "MsDkQb2T": %s, "jVeNAqf": %d, "5ScPjT": "'}, {'offset': 59335, 'value': '"}'}, {'offset': 59707, 'value': 'wsprintfA'}, {'offset': 60228, 'value': 'fabledOverstridence/h31BYUqJ28W62tz?
nonresister=jYnT8Hj13x&Sixtine=TXEWWGZ&DogvaneHeredia=teAiVe'}, {'offset': 62559, 'value': 'InternetOpenW'}, {'offset': 63169, 'value': '&'}, {'offset': 63834,

'value': 'HttpOpenRequestW'}, {'offset': 64241, 'value': 'InternetQueryOptionW'}, {'offset': 65219, 'value': 'Content-Type: application/x-www-form-urlencoded\r\nAccept: */*\r\nAccept-Language: en-US,en;q=0.5\r\nAccept-Encoding: gzip, deflate\r\nUser-Agent: Mozilla/5.0 (compatible; MSIE 5.0; Windows NT 5.01; Trident/5.1)\r\n'}, {'offset': 65756, 'value': 'lstrlenW'}, {'offset': 66009, 'value': 'lstrlenA'}, {'offset': 66300, 'value': 'HttpSendRequestW'}, {'offset': 66602, 'value': 'InternetReadFile'}, {'offset': 66913, 'value': 'InternetCloseHandle'}, {'offset': 67213, 'value': 'InternetCloseHandle'}, {'offset': 67522, 'value': 'InternetCloseHandle'}, {'offset': 68458, 'value': 'InternetCloseHandle'}, {'offset': 69662, 'value': 'InternetCloseHandle'}, {'offset': 70039, 'value': 'InternetSetOptionW'}, {'offset': 72028, 'value': 'InternetOpenW'}, {'offset': 72547, 'value': '&'}, {'offset': 72912, 'value': 'InternetConnectW'}, {'offset': 73249, 'value': 'HttpOpenRequestW'}, {'offset': 73576, 'value': 'InternetQueryOptionW'}, {'offset': 74589, 'value': 'Content-Type: application/x-www-form-urlencoded\r\nAccept: */*\r\nAccept-Language: en-US,en;q=0.5\r\nAccept-Encoding: gzip, deflate\r\nUser-Agent: Mozilla/5.0 (compatible; MSIE 5.0; Windows NT 5.01; Trident/5.1)\r\n'}, {'offset': 75105, 'value': 'lstrlenA'}, {'offset': 75389, 'value': 'HttpSendRequestW'}, {'offset': 75817, 'value': 'InternetReadFile'}, {'offset': 76126, 'value': 'InternetCloseHandle'}, {'offset': 76442, 'value': 'InternetCloseHandle'}, {'offset': 76744, 'value': 'InternetCloseHandle'}, {'offset': 77400, 'value': 'InternetCloseHandle'}, {'offset': 77936, 'value': 'InternetCloseHandle'}, {'offset': 78659, 'value': 'InternetSetOptionW'}, {'offset': 79290, 'value': 'InternetCloseHandle'}, {'offset': 79590, 'value': 'InternetCloseHandle'}, {'offset': 79999, 'value': 'InternetCloseHandle'}, {'offset': 80574, 'value': 'InternetReadFile'}, {'offset': 84550, 'value': 'RegCreateKeyExW'}, {'offset': 84954, 'value': 'RegSetValueExW'}, {'offset': 85214, 'value': 'RegCloseKey'}, {'offset': 85583, 'value': 'RegOpenKeyExW'}, {'offset': 85963, 'value': 'RegQueryValueExW'}, {'offset': 86401, 'value': 'RegCloseKey'}, {'offset': 86933, 'value': 'RegCloseKey'}, {'offset': 87352, 'value': 'C:\\'}, {'offset': 87628, 'value': 'GetVolumeInformationW'}, {'offset': 87911, 'value': '%s\\%s|%s'}, {'offset': 88165, 'value': 'wsprintfW'}, {'offset': 88545, 'value': '%07lX%09lX%lu'}, {'offset': 88791, 'value': 'wsprintfW'}, {'offset': 89618, 'value': 'GetUserDefaultLangID'}, {'offset': 90269, 'value': '%appdata%\\Microsoft\\'}, {'offset': 90574, 'value': 'lotterSig'}, {'offset': 90965, 'value': '\\'}, {'offset': 91247, 'value': 'ExpandEnvironmentStringsW'}, {'offset': 91840, 'value': 'GetFileAttributesW'}, {'offset': 92090, 'value': 'Synanthic'}, {'offset': 92354, 'value': '.dll'}, {'offset': 92795, 'value': '.exe'}, {'offset': 93038, 'value': 'CreateFileW'}, {'offset': 93289, 'value': 'WriteFile'}, {'offset': 93548, 'value': 'CloseHandle'}, {'offset': 93865, 'value': 'CreateDirectoryW'}, {'offset': 94665, 'value': '&'}, {'offset': 95003, 'value': 'SOFTWARE\\Microsoft\\%s'}, {'offset': 95227, 'value': 'lotterSig'}, {'offset': 95465, 'value': 'Subadmini'}, {'offset': 95954, 'value': 'wsprintfW'}, {'offset': 97691, 'value': '{"mdPNC6f8": "%s", "NUn3h77h": "%s", "W381C": "Win %d.%d %d", "SJ3sWSeKQ": %s, "YlSwktC": "%s", "7smbRjIVo": "%s", "AU3KNl": "%s", "hgcPNL3D": %d, "RMe4xUeMl": "%s", "K92GpjSw": %d, "04FjIN": "%s", "yVqTFN": "%s", "TrKEIz": "%s", "gh5jjQAh": "%s", "EoLYU": "%s", "gBmbRM40z": %d, "NUqeH": %d}'}, {'offset': 98028, 'value': 'GG9TU@T@f0adda360d2b4ccda11468e026526576'}, {'offset': 98269, 'value': 'wsprintfW'}, {'offset': 99603, 'value': 'AV89JS'}, {'offset': 100430, 'value': 'TrichinopolyUncontriving/uiDV6mKfgGakdg?unshelledSplitnut=vEzlHkL'}, {'offset': 101031, 'value': '&'}, {'offset': 101306, 'value': 'SOFTWARE\\Microsoft\\%s'}, {'offset': 101535, 'value': 'lotterSig'}, {'offset': 101787, 'value': 'Subadmini'}, {'offset': 102024, 'value': 'wsprintfW'}, {'offset': 112721, 'value':

'CreateToolhelp32Snapshot'}, {'offset': 112957, 'value': 'Process32FirstW'}, {'offset': 113326, 'value': 'CloseHandle'}, {'offset': 113632, 'value': 'Process32NextW'}, {'offset': 113921, 'value': 'explorer.exe'}, {'offset': 114230, 'value': 'OpenProcess'}, {'offset': 114621, 'value': 'CloseHandle'}, {'offset': 118454, 'value': 'InitializeProcThreadAttributeList'}, {'offset': 118832, 'value': 'InitializeProcThreadAttributeList'}, {'offset': 119226, 'value': 'UpdateProcThreadAttribute'}, {'offset': 119806, 'value': 'DeleteProcThreadAttributeList'}, {'offset': 120307, 'value': 'NvtocV4e'}, {'offset': 121017, 'value': 'UpdateProcThreadAttribute'}, {'offset': 124454, 'value': 'InitializeProcThreadAttributeList'}, {'offset': 124806, 'value': 'InitializeProcThreadAttributeList'}, {'offset': 125240, 'value': 'UpdateProcThreadAttribute'}, {'offset': 125508, 'value': 'CreateProcessW'}, {'offset': 125808, 'value': 'DeleteProcThreadAttributeList'}, {'offset': 126893, 'value': 'UpdateProcThreadAttribute'}, {'offset': 132441, 'value': 'IsWow64Process'}, {'offset': 132953, 'value': 'CreateToolhelp32Snapshot'}, {'offset': 133443, 'value': 'Process32FirstW'}, {'offset': 133716, 'value': '['}, {'offset': 134038, 'value': '"%s:%d:%d:%d:%d:%d:%d"'}, {'offset': 134319, 'value': ', "%s:%d:%d:%d:%d:%d:%d"'}, {'offset': 134575, 'value': ']'}, {'offset': 134856, 'value': 'wsprintfW'}, {'offset': 135146, 'value': 'Process32NextW'}, {'offset': 135652, 'value': 'wsprintfW'}, {'offset': 135957, 'value': 'CloseHandle'}, {'offset': 136449, 'value': 'CloseHandle'}, {'offset': 136870, 'value': 'wsprintfW'}, {'offset': 137168, 'value': 'wsprintfW'}, {'offset': 138132, 'value': 'CreatePipe'}, {'offset': 138512, 'value': 'CreateProcessW'}, {'offset': 138862, 'value': 'CloseHandle'}, {'offset': 139129, 'value': 'CloseHandle'}, {'offset': 139377, 'value': 'CloseHandle'}, {'offset': 139622, 'value': 'CloseHandle'}, {'offset': 140399, 'value': 'CloseHandle'}, {'offset': 140751, 'value': 'CloseHandle'}, {'offset': 141454, 'value': 'WaitForSingleObject'}, {'offset': 141835, 'value': 'PeekNamedPipe'}, {'offset': 142183, 'value': 'ReadFile'}, {'offset': 143160, 'value': 'CloseHandle'}, {'offset': 143522, 'value': 'CloseHandle'}, {'offset': 143923, 'value': 'CloseHandle'}, {'offset': 150360, 'value': '&'}, {'offset': 150865, 'value': 'NvtocV4e'}, {'offset': 151420, 'value': '{"mdPNC6f8": "%s", "isjuuMr": "%s", "MsDkQb2T": %s}'}, {'offset': 151686, 'value': 'wsprintfA'}, {'offset': 152132, 'value': 'nanoinstructionFrisesomorum/XjqtQzQyycNZVoIQ?unapplicability=i73MV07GwaCH13Q'}, {'offset': 152389, 'value': 'BaylZ'}, {'offset': 153146, 'value': 'ExpandEnvironmentStringsW'}, {'offset': 154011, 'value': '&'}, {'offset': 154396, 'value': '{"mdPNC6f8": "%s", "isjuuMr": "%s", "MsDkQb2T": %s}'}, {'offset': 155096, 'value': 'wsprintfA'}, {'offset': 155711, 'value': 'nanoinstructionFrisesomorum/XjqtQzQyycNZVoIQ?unapplicability=i73MV07GwaCH13Q'}, {'offset': 155980, 'value': 'BaylZ'}, {'offset': 156993, 'value': 'ExpandEnvironmentStringsW'}, {'offset': 157966, 'value': '{"mdPNC6f8": "%s", "isjuuMr": "%s", "MsDkQb2T": %s}'}, {'offset': 158345, 'value': 'wsprintfA'}, {'offset': 158752, 'value': 'nanoinstructionFrisesomorum/XjqtQzQyycNZVoIQ?unapplicability=i73MV07GwaCH13Q'}, {'offset': 159890, 'value': '&'}, {'offset': 160194, 'value': 'qqmyS'}, {'offset': 160447, 'value': 'tK5nVvwh'}, {'offset': 160722, 'value': 'mGTYP'}, {'offset': 161129, 'value': '2bjHya'}, {'offset': 161691, 'value': 'whoami.exe /all'}, {'offset': 162165, 'value': 'ipconfig.exe /all'}, {'offset': 162490, 'value': 'netstat.exe -aon'}, {'offset': 163444, 'value': '&'}, {'offset': 163844, 'value': 'dLmghDRe'}, {'offset': 164476, 'value': 'ExitProcess'}, {'offset': 164782, 'value': '&'}, {'offset': 165254, 'value': '&'}, {'offset': 166042, 'value': 'bustlingly/e9vliMRRWKSd?DeediestBromes=awIAh8S&bonaght=5vh1psTtP2mk9&stiltyKetohexose=jcAKtvLned5dp8'}, {'offset': 166376, 'value': 'Software\\Microsoft\\Windows\\CurrentVersion\\Run'},

{'offset': 166718, 'value': 'Synanthic'}, {'offset': 167176, 'value': 'rundll32'},
{'offset': 167524, 'value': '.dll'}, {'offset': 167910, 'value': 'wsprintfW'},
{'offset': 169359, 'value': '.exe'}, {'offset': 169611, 'value': 'wsprintfW'},
{'offset': 170996, 'value': '&'}, {'offset': 171344, 'value': 'HxTPXf'}, {'offset':
172030, 'value': 'yAJsnWxR'}, {'offset': 172493, 'value': '4qRRArO'}, {'offset':
172745, 'value': 'NvtocV4e'}, {'offset': 173026, 'value': 'qo9g3J'}, {'offset':
173363, 'value': 'GhPTR'}, {'offset': 173746, 'value': 'dLmghDRe'}, {'offset':
173996, 'value': '9PpreQMX'}, {'offset': 174347, 'value': 'pKW7fqi2'}]}