SmokeLoader Triage

research.openanalysis.net/smoke/smokeloader/loader/config/yara/triage/2022/08/25/smokeloader.html

OALABS Research August 25, 2022

Overview

Samples

- Packed parent cef4f5f561b5c481c67e0a9a3dd751d18d696b61c7a5dab5ebb29535093741b4
- Unpacked SmokeLoader 041a05dd902a55029449bf412cedbe59a593f8d4e67d4ae37cf7a928c92f22ca

SmokeLoader Background

This SmokeLoader sample is on MalwareBazaarand through sandbox runs we know that is was used to download Vidar. From JoeSandbox public report we know we should find the following config in this loader

```
{
  "C2 list": [
    "http://piratia.su/tmp/",
    "http://piratia-life.ru/tmp/",
    "http://diewebseite.at/tmp/",
    "http://faktync.com/tmp/",
    "http://mupsin.ru/tmp/",
    "http://aingular.com/tmp/",
    "http://mordo.ru/tmp/"
  ]
}
```

References

- Deep Analysis of SmokeLoader
- Historical Changes and Trends (Marcos Alvares)

Stage 2

Opaque predicate deobfuscation

From this blog we have a simple jmp fix script.

```
import idc

ea = 0
while True:
    ea =  min(idc.find_binary(ea, idc.SEARCH_NEXT | idc.SEARCH_DOWN, "74 ? 75 ?"),  #
JZ / JNZ
              idc.find_binary(ea, idc.SEARCH_NEXT | idc.SEARCH_DOWN, "75 ? 74 ?"))  #
JNZ / JZ
    if ea == idc.BADADDR:
        break
    idc.patch_byte(ea, 0xEB)    # JMP
    idc.patch_byte(ea+2, 0x90)  # NOP
    idc.patch_byte(ea+3, 0x90)  # NOP
``
```

Once we fix the jmps we need to nop out the junk code between the code to allow IDA to convert this into a function

```python
import idaapi


start = 0x00402DDD
end = 0x00402EBF
ptr = start
while ptr <= end:
    next_ptr = next_head(ptr)
    junk_bytes = next_ptr - ptr
    if ida_bytes.get_bytes(ptr, 1) == b'\xeb':
        idaapi.patch_bytes(ptr, junk_bytes * b'\x90')
    ptr = next_ptr
```

Or, we could use this excellent script from @anthonyprintup

```
import ida_ua
import ida_name
import ida_bytes


def decode_instruction(ea: int) -> ida_ua.insn_t:
    instruction: ida_ua.insn_t = ida_ua.insn_t()
    instruction_length = ida_ua.decode_insn(instruction, ea)
    if not instruction_length:
        return None
    return instruction


def main():
    begin: int = ida_name.get_name_ea(idaapi.BADADDR, "start")
    end: int = begin + 0xE2

    instructions: dict[int, ida_ua.insn_t] = {}

    # Undefine the current code
    ida_bytes.del_items(begin, 0, end)

    # Follow the control flow and create instructions
    instruction_ea: int = begin
    while instruction_ea <= end:
        if instruction_ea not in instructions.keys():
            instruction: ida_ua.insn_t = ida_ua.insn_t()
            instruction_length: int = ida_ua.create_insn(instruction_ea, instruction)
        else:
            instruction: ida_ua.insn_t = decode_instruction(instruction_ea)
            instruction_length: int = instruction.size
        if not instruction_length:
            print(f"Failed to create an instruction at address {instruction_ea=:#x}")
            return

        # Append the current instruction address to the list
        instructions[instruction.ip] = instruction

        # Handle unconditional jumps
        current_instruction_mnemonic: str = instruction.get_canon_mnem()
        next_instruction: ida_ua.insn_t | None = decode_instruction(instruction_ea +
instruction.size)
        if next_instruction is not None:
            next_instruction_mnemonic: str = next_instruction.get_canon_mnem()
            if (current_instruction_mnemonic == "jnz" and next_instruction_mnemonic
== "jz") or \
                    (current_instruction_mnemonic == "jz" and
next_instruction_mnemonic == "jnz"):
                # Unconditional jump detected
                assert instruction.ops[0].type == ida_ua.o_near
                instruction_ea = instruction.ops[0].addr
```

```
                ida_ua.create_insn(next_instruction.ip)
                instructions[next_instruction.ip] = next_instruction
                continue

        if current_instruction_mnemonic == "jmp":
            assert instruction.ops[0].type == ida_ua.o_near
            instruction_ea = instruction.ops[0].addr
        else:
            instruction_ea += instruction.size

    # NOP the remaining instructions
    for ea in range(begin, end):
        skip: bool = False
        for _, instruction in instructions.items():
            if ea in range(instruction.ip, instruction.ip + instruction.size):
                skip = True
                break
        if skip:
            continue

        # Patch the address
        ida_bytes.patch_bytes(ea, b"\x90")


if __name__ == "__main__":
    main()
```

After this we can see that the next function address is built using some stack/ret manipulation.

Function Decryption

Some functions are encrypted. We can find the first one by following the obfuscated control flow until the first `call`. This call calls into a function which then calls the decryption function. The decryption function takes a size and a offset to the function that needs to be decrypted. The size is placed in the `ecx` register, and the function offset follows the call.

```
mov     ecx, 0E7h ; 'ç
```

The decryption itself is a single byte xor but the decryption key is moved into the `edx` register as a full DWORD (we only used the LSB).

```
mov     edx, 76186250h
```

From this blog we have a simple deobfuscation script updated for our sample. This script didn't perform well for some reason so we ended up manually decrypting the functions!

```
import idc
import idautils

def xor_chunk(offset, n):
    ea = 0x400000 + offset
    for i in range(n):
        byte = ord(idc.get_bytes(ea+i, 1))
        byte ^= 0x50
        idc.patch_byte(ea+i, byte)


def decrypt(xref):
    call_xref = list(idautils.CodeRefsTo(xref, 0))[0]
    while True:
        if idc.print_insn_mnem(call_xref) == 'push' and
idc.get_operand_type(call_xref, 0) == idaapi.o_imm:
            n = idc.get_operand_value(call_xref, 0)
            break
        if idc.print_insn_mnem(call_xref) == 'mov' and
idc.get_operand_type(call_xref, 1) == idaapi.o_imm:
            n = idc.get_operand_value(call_xref, 1)
            break
        call_xref = prev_head(call_xref)
    n = idc.get_operand_value(call_xref, 0)
    offset = (xref + 5) - 0x400000
    xor_chunk(offset, n)
    idc.create_insn(offset+0x400000)
    ida_funcs.add_func(offset+0x400000)



xor_chunk_addr = 0x00401118 # address of the xoring function
decrypt_xref_list = idautils.CodeRefsTo(xor_chunk_addr, 0)

for xref in decrypt_xref_list:
    decrypt(xref)
```

API Hashing

According to this blog we are expecting to see some API hashing using the **djb2** algorithm. We can try to find this function by searching for the constant 0x1505.

Though the djb2 algorithm is used for the API hashing the API hashes have some further obfuscation that prevents them from being resolved directly .... **TODO**

```
import requests

hash = 812437175

def hash_djb2(s):
    hash = 5381
    for x in s:
        hash = (( hash << 5) + hash) + x
    return hash & 0xFFFFFFFF


TEST_1 = 1555243728
test = b'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789'

print(hash_djb2(test))

api_name = b'CopyFileW\x00'
api_hash = 0x306cceb7



print(hex(hash_djb2(api_name)))

205455184
0x306cceb7
```

Extract Stage 3

There is a 32-bit and a 64-bit version of stage 3 stored consecutivly in the binary. In older versions of the loader these stages where simply compressed with LZN1 but in this version there appears to be another step that is missing... **TODO**

```
data_string =
'5058187647a21889fa5d103a51605c8eb0621a575b63149f64673087b5753166bb323066f2465570508c1


data_string =
'5036187647a21889fa5d1e12d6615c8ea0623a565b60149f1a8e1e471479b89d409f9877506c1a7a74460


def unhex(hex_string):
    import binascii
    if type(hex_string) == str:
        return binascii.unhexlify(hex_string.encode('utf-8'))
    else:
        return binascii.unhexlify(hex_string)


data = unhex(data_string)

import struct
def decrypt_dw(data, dw):
    out = b''
    for i in range(0,(len(data)//4)*4,4):
        tmp = struct.unpack('<I', data[i:i+4])[0]
        out += struct.pack('<I', tmp ^ dw)
    return out


out = decrypt_dw(data, 0x76186250)

out2 = []
for c in out:
    out2.append(c^ 0x50)


import malduck
ptxt_data = malduck.lznt1(bytes(out2))
print(ptxt_data)
#open('/tmp/out.bin','wb').write(ptxt_data)

b''
```

Stage 3

@doomedraven

> i need to leave in 3rd stage there is a cnc table, just search in 3rd stage for immediate value of 1000 and you will be pretty close to the table