SmokeLoader Triage

Published: 2022-08-25 · Archived: 2026-04-05 19:07:51 UTC

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
```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.siz
        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.

Generic Opaque Predicate Patching

There is also this nice generic patching script from Alex: nopme.py.

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.

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).

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
            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.
            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 malware also encrypts the hashes with a hard coded XOR key. In our sample the key is 0x76186250.

Decryption

There is a 32-bit and a 64-bit version of stage 3 stored consecutivly in the binary. The data is encrypted with a hard coded 4-byte XOR key, the decryption must be a multiple of four. The trailing bytes (if any) are then decrypted with a single byte XOR. In our sample the DWORD key is 0x76186250 and the single byte key is 0x50.

Decompression

Once the stage 3 data is decrypted it is also decompressed with the **LZSA2 algorithm**. We matched this with a blog. The LZSA algorithm is detailed on this github Emmanuel Marty/LZSA.

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