Bandit Stealer Garbled

research.openanalysis.net/bandit/stealer/garble/go/obfuscation/2023/07/31/bandit-garble.html

OALABS Research July 31, 2023

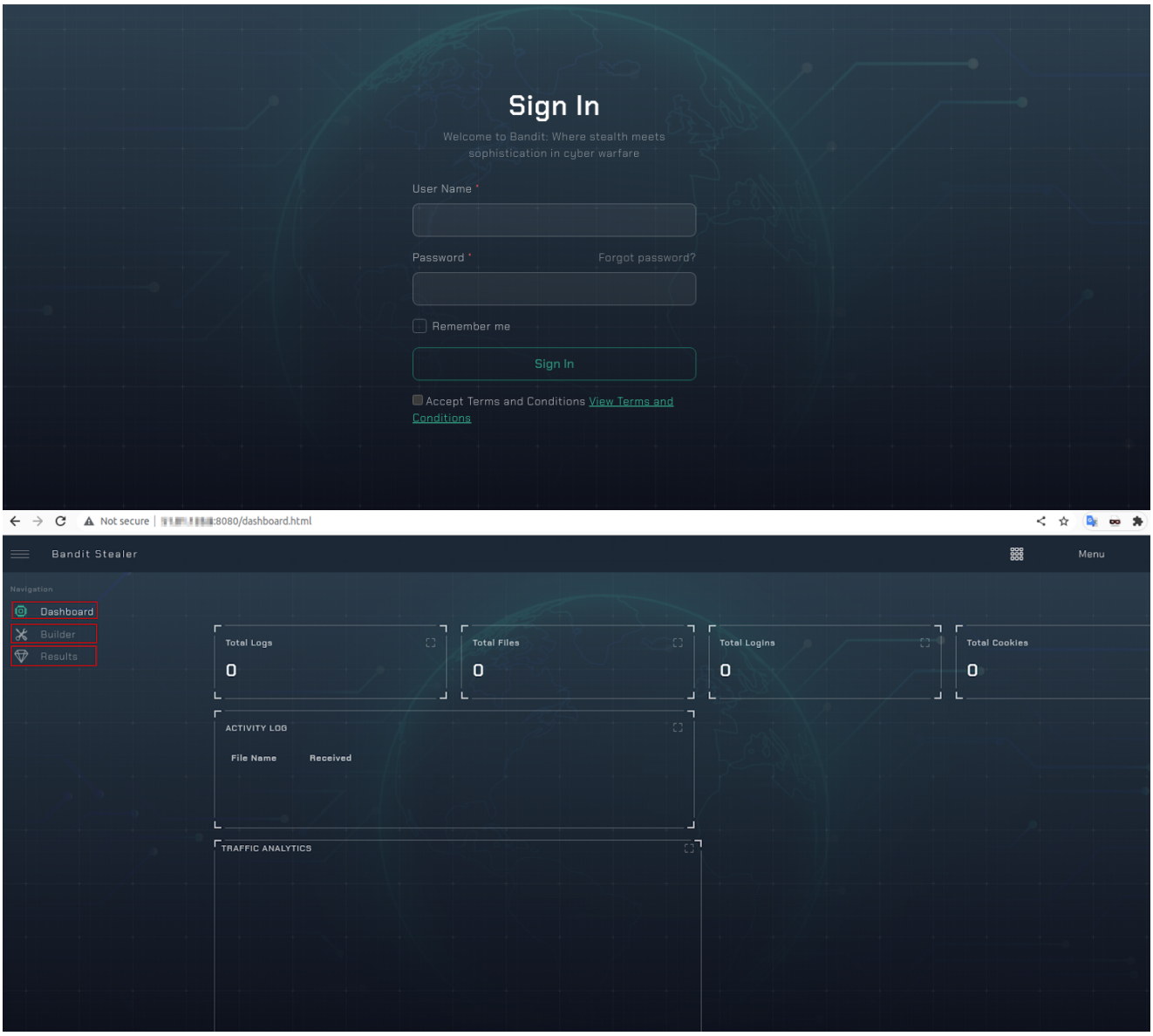
Overview

This is a new infostealer written in GO that primarily targets browser credentials and crypto wallets. The collected information is uploaded to Telegram with the operator's telegram ID and channel ID hard coded in the binary. Some variants of the stealer use Garble an open source GO obfuscator.

References

Samples

- 050DBD816C222D3C012BA9F2B1308DB8E160E7D891F231272F1EACF19D0A0A06
- 51f357928b0829743b01733840ad190d6cbb0ac593df23bf8029b6d86ffc9251
- 623a5f4c57cf5b3feb6775508cd6492f89d55ce11f62e0b6fb1020fd730b2e8f (obfuscated)



Analysis

According to CloudSek the stealer panel has security flaws that allow some of the data to be accessed without authentication. This example shows the main panel for the stealer `http[:]//142.202.240[.]84:8080/csetayukhv.html`

Binary Analysis

Early builds of the stealer did not use obfuscation, and contained plaintext strings. These versions were simple to reverse engineer once the method names were recovered using GO IDA parser (works well!). Later versions of the stealer attempted to slightly obfuscate the method names and ultimately moved to using Garble a GO obfuscator.

Garble

Garble is able to obfuscate GO method names, obfuscate strings, and modify control flow. Each option can be enabled separately. We will be analyzing the following Bandit Stealer sample obfuscated with Garble `623a5f4c57cf5b3feb6775508cd6492f89d55ce11f62e0b6fb1020fd730b2e8f`.

Method Name Obfuscation

The method names are obfuscated using a hash which is then base64 encoded. The method metadata recovery process is not possible using our favorite IDA script but we can use GoReSym. When this is run we can see method names like...

```
{
            "Start": 5369065792,
            "End": 5369065824,
            "PackageName": "h20dLQEZPVaM",
            "FullName": "h20dLQEZPVaM.CvGFbRy"
        },
        {
            "Start": 5369072416,
            "End": 5369072512,
            "PackageName": "h20dLQEZPVaM",
            "FullName": "h20dLQEZPVaM.IRlPxsFX"
        },
        {
            "Start": 5369072512,
            "End": 5369072608,
            "PackageName": "h20dLQEZPVaM",
            "FullName": "h20dLQEZPVaM.MRLxEQ"
        },
```

Idea it might be possible to brute force these if you had access to an earlier version of the sample which has the full method names

String Obfuscation

The string obfuscation appears to result in dedicated functions for each obfuscated string which consist of some constants and an algorithm used to recreate the string. The function then converts the resulting byte string into a go string and returns it.

```
.text:00000001407BC661                mov     ecx, 0Ah
.text:00000001407BC666                call    runtime_slicebytetostring
.text:00000001407BC66B                mov     rbp, [rsp+38h+var_8]
.text:00000001407BC670                add     rsp, 38h
.text:00000001407BC674                retn
```

It may be possible to attack this by identifying the dedicated string functions and emulating them!

```
from unicorn import *
from unicorn.x86_const import *
import struct
from capstone import *
from capstone.x86 import *


uc = Uc(UC_ARCH_X86, UC_MODE_64)

# Setup the stack
stack_base = 0x00100000
stack_size = 0x00100000
RSP = stack_base + (stack_size // 2)
uc.mem_map(stack_base, stack_size)
uc.mem_write(stack_base, b"\x00" * stack_size)

uc.reg_write(UC_X86_REG_RSP, RSP)

# Setup code

code = bytes.fromhex('49 3B 66 10 0F 86 C8 00 00 00 48 83 EC 38 48 89 6C 24 30 48 8D
6C 24 30 48 BA E5 B1 F0 56 65 EA 73 C9 48 89 54 24 18 66 C7 44 24 20 6F 6E 48 BA 02
00 01 05 01 05 05 07 48 89 54 24 22 48 BA 05 07 05 02 00 07 07 05 48 89 54 24 28 31
C0 EB 1D 44 0F B6 4C 34 18 41 29 D1 41 8D 51 E1 88 54 3C 18 41 8D 50 E1 88 54 34 18
48 83 C0 02 48 83 F8 0E 7D 27 0F B6 54 04 22 0F B6 74 04 23 89 D7 31 F2 01 C2 48 83
FF 0A 73 3C 44 0F B6 44 3C 18 41 29 D0 48 83 FE 0A 72 B8 EB 1B 31 C0 48 8D 5C 24 18
B9 0A 00 00 00 E8 D5 B8 88 FF 48 8B 6C 24 30 48 83 C4 38 C3 89 F0 B9 0A 00 00 00 0F
1F 40 00 E8 3B 02 8A FF 89 F8 B9 0A 00 00 00 E8 2F 02 8A FF 90 E8 A9 DB 89 FF E9 24
FF FF FF CC CC CC CC')


target_base = 0x00400000
target_size = 0x00100000
target_end = target_base + len(code)

uc.mem_map(target_base, target_size, UC_PROT_ALL)
uc.mem_write(target_base, b"\x00" * target_size)
uc.mem_write(target_base, code)


data_base = 0x00600000
data_size = 0x00100000

uc.mem_map(data_base, data_size, UC_PROT_ALL)
uc.mem_write(data_base, b"\x00" * data_size)


cs = Cs(CS_ARCH_X86, CS_MODE_64)
cs.detail = True

def trace(uc, address, size, user_data):
    insn = next(cs.disasm(uc.mem_read(address, size), address))
```

```
    #print(f"{address:#010x}:\t{insn.mnemonic}\t{insn.op_str}")
    if insn.mnemonic == 'call':
        print("Ending on a call!")
        uc.emu_stop()

uc.reg_write(UC_X86_REG_R14, data_base)
uc.hook_add(UC_HOOK_CODE, trace, None)
uc.emu_start(target_base, target_end, 0, 0)

ptr_string = uc.reg_read(UC_X86_REG_RBX)
size = uc.reg_read(UC_X86_REG_RCX)
#print(hex(ptr_string))
string_data = uc.mem_read(ptr_string, size)
string = string_data.decode('utf-8')
print(string)

Ending on a call!
GetVersion
```

Stand Alone String Decryption

```
from unicorn import *
from unicorn.x86_const import *
import struct
from capstone import *
from capstone.x86 import *


def decrypt(code_string):
    uc = Uc(UC_ARCH_X86, UC_MODE_64)

    # Setup the stack
    stack_base = 0x00100000
    stack_size = 0x00100000
    RSP = stack_base + (stack_size // 2)
    uc.mem_map(stack_base, stack_size)
    uc.mem_write(stack_base, b"\x00" * stack_size)

    uc.reg_write(UC_X86_REG_RSP, RSP)

    # Setup code
    code = bytes.fromhex(code_string)
    target_base = 0x00400000
    target_size = 0x00100000
    target_end = target_base + len(code)

    uc.mem_map(target_base, target_size, UC_PROT_ALL)
    uc.mem_write(target_base, b"\x00" * target_size)
    uc.mem_write(target_base, code)


    data_base = 0x00600000
    data_size = 0x00100000

    uc.mem_map(data_base, data_size, UC_PROT_ALL)
    uc.mem_write(data_base, b"\x00" * data_size)


    cs = Cs(CS_ARCH_X86, CS_MODE_64)
    cs.detail = True

    def trace(uc, address, size, user_data):
        insn = next(cs.disasm(uc.mem_read(address, size), address))
        #print(f"{address:#010x}:\t{insn.mnemonic}\t{insn.op_str}")
        if insn.mnemonic == 'call':
            #print("Ending on a call!")
            uc.emu_stop()

    uc.reg_write(UC_X86_REG_R14, data_base)
    uc.hook_add(UC_HOOK_CODE, trace, None)
    uc.emu_start(target_base, target_end, 0, 0)

    #print(uc.mem_read(stack_base, stack_size).replace(b'\x00', b''))
```

```
    ptr_string = uc.reg_read(UC_X86_REG_RBX)
    size = uc.reg_read(UC_X86_REG_RCX)
    string_data = uc.mem_read(ptr_string, size)
    string = string_data.decode('utf-8')
    return string



#print(decrypt('49 3B 66 10 0F 86 C8 00 00 00 48 83 EC 38 48 89 6C 24 30 48 8D 6C 24
30 48 BA E5 B1 F0 56 65 EA 73 C9 48 89 54 24 18 66 C7 44 24 20 6F 6E 48 BA 02 00 01
05 01 05 05 07 48 89 54 24 22 48 BA 05 07 05 02 00 07 07 05 48 89 54 24 28 31 C0 EB
1D 44 0F B6 4C 34 18 41 29 D1 41 8D 51 E1 88 54 3C 18 41 8D 50 E1 88 54 34 18 48 83
C0 02 48 83 F8 0E 7D 27 0F B6 54 04 22 0F B6 74 04 23 89 D7 31 F2 01 C2 48 83 FF 0A
73 3C 44 0F B6 44 3C 18 41 29 D0 48 83 FE 0A 72 B8 EB 1B 31 C0 48 8D 5C 24 18 B9 0A
00 00 00 E8 D5 B8 88 FF 48 8B 6C 24 30 48 83 C4 38 C3 89 F0 B9 0A 00 00 00 0F 1F 40
00 E8 3B 02 8A FF 89 F8 B9 0A 00 00 00 E8 2F 02 8A FF 90 E8 A9 DB 89 FF E9 24 FF FF
FF CC CC CC CC'))
#print(decrypt('4C 8D 64 24 C8 4D 3B 66 10 0F 86 57 01 00 00 48 81 EC B8 00 00 00 48
89 AC 24 B0 00 00 00 48 8D AC 24 B0 00 00 00 48 BA 03 DC F5 44 2F 20 21 53 48 89 54
24 1D 48 BA AA 6D 47 01 6F 45 3A 04 48 89 54 24 25 48 BA 01 6F 45 3A 04 01 23 95 48
89 54 24 28 48 BA 72 19 F5 3B 01 EA 20 47 48 89 54 24 30 48 BA A6 18 DF 72 1B 69 53
4E 48 89 54 24 38 48 BA 4D 76 17 0A 06 E4 23 57 48 89 54 24 40 48 BA 09 85 2F E6 28
FC 36 02 48 89 54 24 48 48 BA B2 49 E8 14 19 4D 29 64 48 89 54 24 50 48 BA 2C 2F 26
12 27 1B 32 0B 48 89 54 24 58 48 8D 7C 24 60 48 8D 35 E7 00 A6 00 0F 1F 80 00 00 00
00 48 89 6C 24 F0 48 8D 6C 24 F0 E8 AB D9 FE FF 48 8B 6D 00 31 C0 EB 1A 44 0F B6 4C
34 1D 41 83 C1 E0 44 01 CA 88 54 3C 1D 44 88 44 34 1D 48 83 C0 02 48 83 F8 58 7D 31
0F B6 54 04 58 0F B6 74 04 59 89 D7 31 F2 01 C2 48 83 FF 3B 73 48 44 0F B6 44 3C 1D
41 83 C0 E0 41 01 D0 66 0F 1F 44 00 00 48 83 FE 3B 72 B1 EB 21 31 C0 48 8D 5C 24 1D
B9 3B 00 00 00 E8 47 89 FD FF 48 8B AC 24 B0 00 00 00 48 81 C4 B8 00 00 00 C3 89 F0
B9 3B 00 00 00 E8 AB D2 FE FF 89 F8 B9 3B 00 00 00 0F 1F 40 00 E8 9B D2 FE FF 90 E8
15 AC FE FF E9 90 FE FF FF CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC'))
print(decrypt('49 3B 66 10 0F 86 EC 00 00 00 48 83 EC 48 48 89 6C 24 40 48 8D 6C 24
40 48 BA C8 61 12 01 F6 F6 69 3E 48 89 54 24 1A 48 BA 12 01 F6 F6 69 3E 6E 5F 48 89
54 24 1C 48 BA 41 6C 62 9B 2A 69 96 19 48 89 54 24 24 48 BA 00 10 0E 10 00 0E 0D 07
48 89 54 24 2C 48 BA 00 0E 0D 07 0D 10 03 10 48 89 54 24 30 48 BA 0D 00 02 11 07 05
04 04 48 89 54 24 38 31 C0 EB 1D 44 0F B6 4C 34 1A 41 29 D1 41 8D 51 7D 88 54 3C 1A
41 8D 50 7D 88 54 34 1A 48 83 C0 02 48 83 F8 14 7D 29 0F B6 54 04 2C 0F B6 74 04 2D
89 D7 31 F2 01 C2 48 83 FF 12 73 3A 44 0F B6 44 3C 1A 41 29 D0 48 83 FE 12 72 B8 66
90 EB 1B 31 C0 48 8D 5C 24 1A B9 12 00 00 00 E8 CD 0D FD FF 48 8B 6C 24 40 48 83 C4
48 C3 89 F0 B9 12 00 00 00 E8 37 57 FE FF 89 F8 B9 12 00 00 00 E8 2B 57 FE FF 90 E8
A5 30 FE FF 0F 1F 44 00 00 E9 FB FE FF FF CC CC CC CC CC CC CC CC CC CC CC CC CC CC
CC CC CC CC CC CC CC CC CC CC CC CC CC'))

Caucasian_Albanian
```

TODO

- handle functions that have a call they need to execute
- check for decryption bugs
- write function identification algorithm based on prologue