

# Guloder Deobfuscation using Ghidra

 [irfan-eternal.github.io/guloder-deobfuscation-using-ghidra/](https://github.com/irfan-eternal/guloder-deobfuscation-using-ghidra/)

July 23, 2023



[irfan\\_eternal](#) included in [Malware Analysis](#)

2023-07-23 946 words 5 minutes



Contents

## Introduction

---

Hi all, Today we will be Analyzing Guloader Shellcode using Ghidra. Our Objective is to Identify some Anti-analysis and Obfuscation techniques used by Guloader and Defeat it using Automation. People who would like to follow along can download the sample from [here](#). The File was seen on 2023-05-11

## Anti-Analysis / Obfuscation

---

### API Hashing

---

The Shellcode is using API hashing to hide API's being called. For Each API Resolving it first resolves [LdrLoadDll](#) add 5 to it's address to avoid any Hooking done by EDR. Adding 5 to the API address is to avoid the classic 5 Byte Hook. Most EDR replaces first first bytes with a

jump to EDR's code. It then use this address to Load the DLL. after Loading the DLL . it resolves the hash of the API it needs to call

```

dword __fastcall resolveAPIFunc(dword DllName,dword ApI_hash)
{
    undefined4 in_EAX;
    int extraout_ECX;
    undefined4 unaff_EBX;
    int unaff_EBP;
    int unaff_ESI;
    undefined4 unaff_EDI;
    undefined2 *DllName2;
    dword dllName;

    *(undefined4 *) (unaff_EBP + 0x212) = unaff_EBX;
    *(undefined4 *) (unaff_EBP + 0x22d) = in_EAX;
    *(dword *) (unaff_EBP + 0x1df) = ApI_hash;
    *(undefined4 *) (unaff_EBP + 0x1c2) = 0x5c618f9c;
    *(uint *) (unaff_EBP + 0x1c2) = *(uint *) (unaff_EBP + 0x1c2) ^ 0xe289b05a;
    *(uint *) (unaff_EBP + 0x1c2) = *(uint *) (unaff_EBP + 0x1c2) ^ 0x26e75f13;
    *(int *) (unaff_EBP + 0x1c2) = *(int *) (unaff_EBP + 0x1c2) + 0x67f0a464;
    if (*(int *) (unaff_EBP + 0x22d) == *(int *) (unaff_EBP + 0x1c2)) {
        /* ntdll */
        resolveAPILibrary(0x9cda770c);
        /* ldrload.dll */
        resolveHashedApi's(0x55db1585);
        /* anti-hooking for LdrLoadDll */
        *(int *) (unaff_EBP + 0xc) = unaff_ESI + 5;
    }
    DllName2 = *(undefined2 **) (unaff_EBP + 0x164);
    *(dword *) (DllName2 + 2) = DllName;
    returnstrlength(DllName);
    *DllName2 = (short)extraout_ECX;
    *(undefined4 *) (unaff_EBP + 0x235) = unaff_EDI;
    DllName2[1] = (short)(extraout_ECX + 2);
    *(undefined2 **) (unaff_EBP + 0x17f) = DllName2;
    dllName = *(dword *) (unaff_EBP + 0x17f);
    *(int *) (unaff_EBP + 0x24f) = extraout_ECX + 2;
    LdrLoadDll(0,0,(char)dllName,(char)unaff_EBP + 'h');
    resolveHashedApi's(ApI_hash);
    return dllName;
}

```

API Hashing function is the same as we see in the wild. it goes to the Export Directory of the DLL Loaded and performs Hashing of all API name's till the hashes match .if the hashes match it stores the Address of the API

```

uint resolveHashedApi's(int Api_Hash)
{
    int in_EAX;
    IMAGE_EXPORT_DIRECTORY *exportDir;
    int iVar1;
    uint uVar2;
    int iVar3;
    int iVar4;
    int unaff_EBP;
    int *piVar5;

    *(int *) (unaff_EBP + 4) = in_EAX;
    iVar4 = *(int *) (in_EAX + *(int *) (in_EAX + 0x3c) + 0x78);
    exportDir = (IMAGE_EXPORT_DIRECTORY *) (*(int *) (unaff_EBP + 4) + iVar4);
    *(DWORD *) (unaff_EBP + 8) = exportDir->NumberOfNames;
    *(DWORD *) (unaff_EBP + 0x14) = exportDir->AddressOfFunctions;
    *(DWORD *) (unaff_EBP + 0x10) = exportDir->AddressOfNameOrdinals;
    piVar5 = (int *) (exportDir->AddressOfNames + *(int *) (unaff_EBP + 4));
    iVar3 = 0;
    do {
        iVar1 = *piVar5;
        *(int *) (unaff_EBP + 0x26f) = iVar4;
        iVar4 = *(int *) (unaff_EBP + 0x26f);
        *(int *) (unaff_EBP + 0x17c) = iVar1 + *(int *) (unaff_EBP + 4);
        *(int *) (unaff_EBP + 0x27a) = iVar3;
        iVar1 = dbj2+XOR(*(undefined4 *) (unaff_EBP + 0x17c));
        *(int *) (unaff_EBP + 0x23a) = iVar3;
        if (iVar1 == Api_Hash) {
            uVar2 = (uint) *(ushort *)
                (*(int *) (unaff_EBP + 0x10) + *(int *) (unaff_EBP + 4) +
                *(int *) (unaff_EBP + 0x23a) * 2);
            *(int *) (unaff_EBP + 8) =
                *(int *) (*(int *) (unaff_EBP + 0x14) + *(int *) (unaff_EBP + 4) + uVar2 * 4) +
                *(int *) (unaff_EBP + 4);
            return uVar2;
        }
        piVar5 = piVar5 + 1;
        iVar3 = *(int *) (unaff_EBP + 0x23a) + 1;
        *(int *) (unaff_EBP + 0x1aa) = iVar1;
    } while (iVar3 != *(int *) (unaff_EBP + 8));
    return *(uint *) (unaff_EBP + 0x1aa);
}

```

Decompile: resolveHash... x 00 Bytes: thirdstage x 0101 DAT Defined Strings x

Hashing Algorithm : In the past Guloder was using just DJB2 hash . Now it xors the result of djb2 hash with a hardcoded value to perform API Hashing

```
undefined * __fastcall dbj2+XOR(uint param_1_00,undefined param_2,byte *param_1)
{
    undefined *puVar1;
    int unaff_EBX;
    int unaff_EBP;

    puVar1 = &LAB_00001502+3;
    while( true ) {
        *(int *) (unaff_EBP + 0x212) = unaff_EBX;
        if ((*param_1 == 0x40) || (*(char *) (unaff_EBP + 0x1e8) = (char)param_1_00, *param_1 == 0x3f)) {
            return (undefined *)0x0;
        }
        *(undefined **) (unaff_EBP + 0x1e8) = puVar1;
        unaff_EBX = *(int *) (unaff_EBP + 0x1e8);
        if (*param_1 == 0x24) {
            return (undefined *)0x0;
        }
        *(undefined *) (unaff_EBP + 0x1c2) = param_2;
        param_2 = *(undefined *) (unaff_EBP + 0x1c2);
        if (0xa3 < *param_1) break;
        param_1_00 = (uint)*param_1;
        puVar1 = (undefined *) (unaff_EBX * 0x21 + param_1_00 ^ 0x8131a1);
        param_1 = param_1 + 1;
        if (*param_1 == 0) {
            return puVar1;
        }
    }
    return (undefined *)0x0;
}
```

I wrote a python script to identify the Hashes of the API's used by the past Guloaders to identify the Functionalities of the Code

```

val = 0x1505
APIStrings = ["NtGetContextThread", "RtlAddVectoredExceptionHandler",
"NtAllocateVirtualMemory", "DbgUIRemoteBreakIn", "LdrLoadDll", "DbgBreakPoint",
"EnumWindows", "NtSetInformationThread", "ZwSetInformationThread",
"TerminateProcess", "ExitProcess", "NtSetContextThread", "NtWriteVirtualMemory",
"NtCreateSection", "NtMapViewOfSection", "NtOpenFile", "NtSetInformationProcess",
"NtClose",
"NtResumeThread", "NtProtectVirtualMemory", "CreateProcessInternal",
"GetLongPathNameW", "Sleep", "NtCreateThreadEx", "WaitForSingleObject",
"TerminateThread", "CreateFileW", "WriteFile", "ReadFile", "ShellExecuteW",
"RegCreateKeyExA", "RegSetValueExA", "NtQueryInformationProcess",
"InternetOpenA", "InternetSetOptionA", "InternetOpenUrlA", "InternetReadFile",
"InternetCloseHandle"]

for APIString in APIStrings:
    val = 0x1505
    for ch in instring:
        val += ((val <<5))
        val &= 0xFFFFFFFF
        val += ord(ch)
        val &= 0xFFFFFFFF
        val ^= 0x8131A1
    print(APIString+" : "+hex(val))

```

## Checks before Calling API's

---

Before Calling the API it performs some Checks to make sure it is not being Debugged

```
void __cdecl call_api's(undefined api,undefined param1)
{
    code *UNRECOVERED_JUMPTABLE;
    uint *puVar1;
    ushort in_CX;
    uint *puVar2;
    int unaff_EBP;
0   undefined4 unaff_EDI;
1   uint unaff_retaddr;
2   undefined3 in_stack_00000005;
3
4   *(undefined4 *) (unaff_EBP + 300) = unaff_EDI;
5   *(undefined4 *) (unaff_EBP + 0x154) = 0x1855c;
6   *(uint *) (unaff_EBP + 0x150) = unaff_retaddr;
7   puVar1 = *(uint **) (unaff_EBP + 0x44);
8   do {
9       *puVar1 = *puVar1 ^ unaff_retaddr;
10      puVar1 = puVar1 + 1;
11  } while (puVar1 != (uint *)0x1855c);
12  mw_checkforbreakpintorExcpetionatApi(in_CX, (short)unaff_retaddr, _api);
13  (*_api)();
14  UNRECOVERED_JUMPTABLE = *(code **) (unaff_EBP + 0x150);
15  puVar1 = *(uint **) (unaff_EBP + 0x44);
16  puVar2 = *(uint **) (unaff_EBP + 0x154);
17  do {
18      puVar2 = puVar2 + -1;
19      *puVar2 = *puVar2 ^ (uint)UNRECOVERED_JUMPTABLE;
20  } while (puVar2 != puVar1);
21      /* WARNING: Could not recover jumtable at 0x000185be. Too many branches */
22      /* WARNING: Treating indirect jump as call */
23  (*UNRECOVERED_JUMPTABLE)();
24  return;
25  }
26  }
```

It compares the first few bytes of the API address with a) CC b) INT 3 c) UD2 to check if it's being Debugged. if this is not the case it calls the API

```

void __fastcall mw_checkforbreakpointorExcpetionatApi(ushort param_1,undefined2 param_2,short *api)
{
    char cVar1;
    short sVar2;
    int unaff_EBP;
    code *UNRECOVERED_JUMPTABLE;

0          /* check if api is having any breakpoint or exception
1          cc - INT3
2          cd 03 - INT 3
3          of 0b - UD2 */
4    cVar1 = *(char *)api;
5    *(char *) (unaff_EBP + 0x1c1) = (char)param_1;
6    if (cVar1 != '\xcc') {
7        sVar2 = *api;
8        *(undefined2 *) (unaff_EBP + 0x1be) = param_2;
9        if ((sVar2 != 0x3cd) &&
0            (sVar2 = *api,
1             *(ushort *) (unaff_EBP + 0x1ad) = param_1 & 0xff00 | (ushort)*(byte *) (unaff_EBP + 0x1c1),
2             sVar2 != 0xb0f)) {
3            return;
4        }
5    }
6    /* WARNING: Could not recover jumptable at 0x000185d4. Too many branches */
7    /* WARNING: Treating indirect jump as call */
8    (*UNRECOVERED_JUMPTABLE) ();
9    return;
0}
1

```

## Control Flow Obfuscation using Vectored Exception Handling

To Obfuscate the Control Flow it uses RtlAddVectoredExceptionHandler to register a vectored Exception Handler. If the Exception raised is any of below three cases it changes the EIP by an XOR operation

1. EXCEPTION\_ACCESS\_VIOLATION while accessed memory Address is 0
2. EXCEPTION\_SINGLE\_STEP (Single Stepping )
3. EXCEPTION\_BREAKPOINT (Software Break Point - CC)

If it is EXCEPTION\_BREAKPOINT it calculates the value of the EIP by this expression  $EIP = EIP + *(EIP+1) \wedge 6A$  (Changes with sample)

If it is EXCEPTION\_ACCESS\_VIOLATION or EXCEPTION\_SINGLE\_STEP it calculates the value of the EIP by this expression  $EIP = EIP + *(EIP+2) \wedge 6A$  (Changes with sample) .

```

undefined4 vectoredExceptionHandler(_EXCEPTION_POINTERS *param_1)
{
    DWORD DVar1;
    _CONTEXT *iVar2;
    _CONTEXT *iVar3;
    char *pcVar2;
    uint uVar3;

    FUN_00000177();
    DVar1 = param_1->ExceptionRecord->ExceptionCode;
    if (DVar1 == L'\xc0000005') {
        if (param_1->ExceptionRecord->ExceptionInformation[1] != 0) {
            return 0;
        }
    }
    else if (DVar1 != L'\x80000004') {
        if (DVar1 == L'\x80000003') {
            iVar2 = (_CONTEXT *)nw_hardwareBreakpoint_check();
            if (*(char *)iVar2->Eip == '\xcc') {
                uVar3 = (uint)(byte)((char *)iVar2->Eip)[1] ^ 0x6a;
                pcVar2 = (char *) (iVar2->Eip + uVar3);
                do {
                    pcVar2 = pcVar2 + -1;
                    if ((char *) (iVar2->Eip + 2) == pcVar2) {
                        iVar2->Eip = iVar2->Eip + uVar3;
                        return 0xffffffff;
                    }
                } while (*pcVar2 != '\xcc');
            }
        }
    }
    return 0;
}
iVar3 = (_CONTEXT *)nw_hardwareBreakpoint_check();
iVar3->Eip = iVar3->Eip + (uint)(*(byte *) (iVar3->Eip + 2) ^ 0x6a);
return 0xffffffff;
}

```

In the Vectored Exception Handler function it also checks if any Dynamic analysis is being performed by Checking if any hardware breakpoint is set using ContextRecord a member of [\\_EXCEPTION\\_POINTERS](#)

```

int nw_hardwareBreakpoint_check(_EXCEPTION_POINTERS *exception_pointer)
{
    PCONTEXT pCVar1;

    pCVar1 = exception_pointer->ContextRecord;
    if (((pCVar1->Dr0 == 0) && (pCVar1->Dr1 == 0)) && (pCVar1->Dr2 == 0)) &&
        (((pCVar1->Dr3 == 0 && (pCVar1->Dr6 == 0)) && (pCVar1->Dr7 == 0)))) {
0      return (int)pCVar1;
1    }
2    return 0;
3}
4

```

To Understand Vectored Exception in Detail i would suggest to read this [article](#)

## Encrypted Strings

---

All the Important strings including the C2 URL is Encrypted using XOR. The Encrypted strings are created by Performing a lot of Mathematical Expression(not Hardcoded).After Encryption String is Fully Created . The first Dword contains the length of the Encrypted string. then what follows is the Encrypted string. The Address of the Encrypted string is given as parameter to the String Decryption preparation Function. It store the Encrypted String length in a Variable. And changes the First Dword to a Dummy Value . After that increments the Address of Encrypted String by a dword now it points to the Actual start of the String.

Encrypted\_string and the Encrypted String length is given as parameters to the next Function called which is a wrapper around the string decryption function

```
void string_decryption_prep(undefined4 param_1_00, uint *encrypted_data, undefined param_3)
{
    int *piVar1;
    uint data_size;
    uint *puVar2;
    int unaff_EBP;

0  data_size = *encrypted_data;
1  *encrypted_data = 0x9d259cda;
2  *encrypted_data = *encrypted_data + 0x191037b1;
3  *encrypted_data = *encrypted_data ^ 0x9bc5a252;
4  *encrypted_data = *encrypted_data ^ 0x7c388ac3;
5  *(undefined4 *) (unaff_EBP + 0x246) = 0x47e34586;
6  *(int *) (unaff_EBP + 0x246) = *(int *) (unaff_EBP + 0x246) + 0xa7e9750;
7  *(uint *) (unaff_EBP + 0x246) = *(uint *) (unaff_EBP + 0x246) ^ 0xf6d6c2c8;
8  *(int *) (unaff_EBP + 0x246) = *(int *) (unaff_EBP + 0x246) + 0x5b48e1e5;
9  do {
0      puVar2 = encrypted_data;
1      piVar1 = (int *) (unaff_EBP + 0x246);
2      *piVar1 = *piVar1 + -1;
3      encrypted_data = (uint *) ((int)puVar2 + 1);
4  } while (*piVar1 != 0);
5  wrap_wrap_string_decryption((int)puVar2 + 2, data_size);
6  return;
7}
8
```

The wrapper function passes the Encrypted\_string, Encrypted String length and the Key. The key is actually stored in the return address of the wrapper function

```

*****
*
* FUNCTION
*
*****

void __stdcall wrap_wrap_string_decryption(int encrypted-data, int data_size)

void          <VOID>          <RETURN>
int           EAX:4           encrypted-data
int           EBX:4           data_size
undefined4    Stack[-0x4]:4   local_4

wrap_wrap_string_decryption          XREF[1]:  string_decryp

00016ace e8 6c ff ff ff  CALL      wrap-string_decryption
Key of xor is the return address

00016ad3 50 6e e0 40      ddw      40E06E50h
00016ad7 0a 3b 23 49      ddw      49233B0Ah
00016adb 83 4a ca 37      ddw      37CA4A83h
00016adf b8 8e 69 42      ddw      42698EB8h
00016ae3 c7 46 f6 2a      ddw      2AF646C7h
00016ae7 eb 1d 85 e8      ddw      E8851DEBh
00016aeb 46 3a 41 f8      ddw      F8413A46h
00016aef 14 66 39 ac      ddw      AC396614h
00016af3 36 6d be 33      ddw      33BE6D36h
00016af7 c3 80 64 11      ddw      116480C3h
00016afb 02 6e 81 b8      ddw      B8816E02h
00016aff 7a 5a 2b 03      ddw      32B5A7Ah
00016b03 c1 6f 21 37      ddw      37216FC1h
00016b07 86 e3 3e 8a      ddw      8A3EE386h

```

The Decryption Algorithm is a Simple XOR where Encrypted string is calculated by Mathematical Expression and key is stored in the return address of the wrapper function

```
Decompiler: decryption - (thirdstage)
1
2 void decryption(int encrypted_data,int data_size,int 3e,int key)
3
4 {
5     int iVar1;
6     byte *pbVar2;
7     byte *pbVar3;
8
9     iVar1 = -data_size;
10    pbVar3 = (byte *)-3e;
11    do {
12        pbVar2 = pbVar3 + key + 3e;
13        pbVar3 = pbVar2 + (1 - (key + 3e));
14        if (pbVar2 + (1 - (key + 3e)) == (byte *)0x0) {
15            pbVar3 = (byte *)-3e;
16        }
17        *(byte *) (encrypted_data + data_size + iVar1) =
18            *(byte *) (encrypted_data + data_size + iVar1) ^ *pbVar2;
19        iVar1 = iVar1 + 1;
20    } while (iVar1 != 0);
21    return;
22}
23
```

## Automation using Ghidra API's

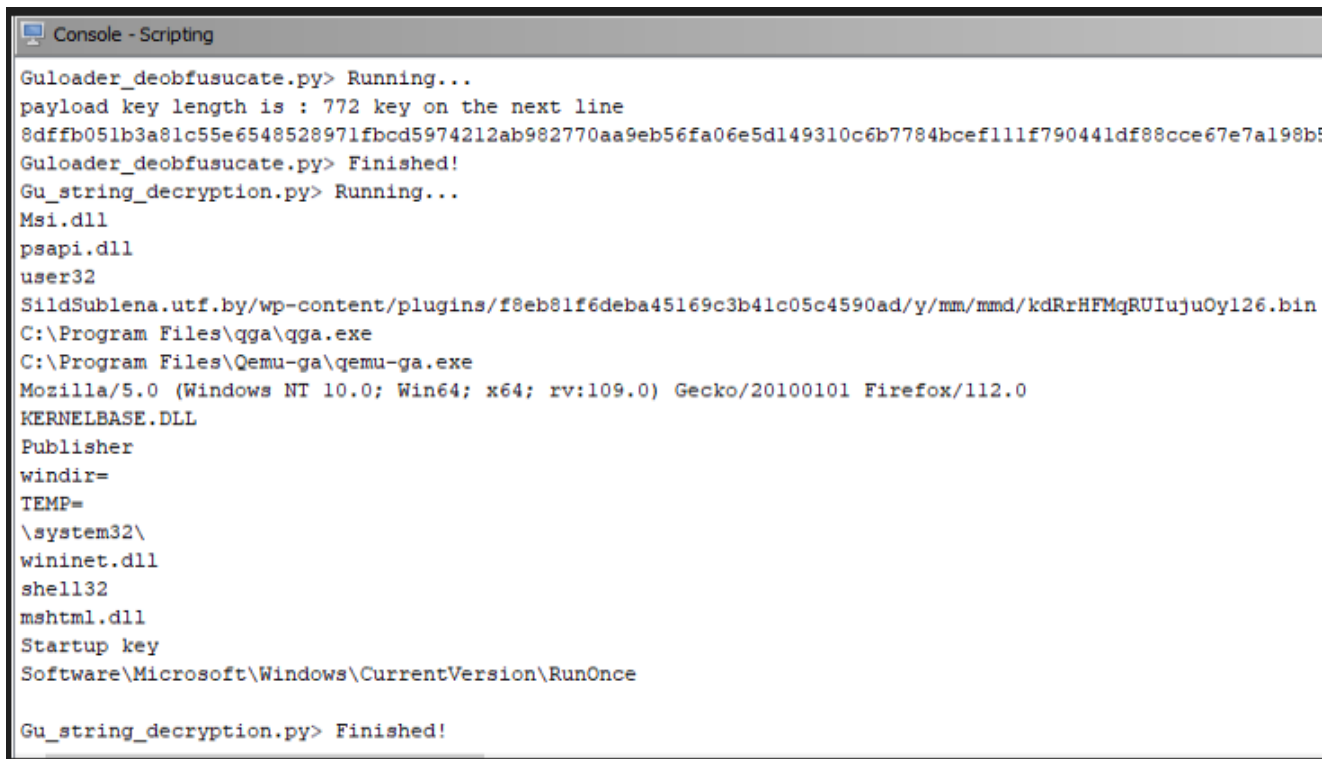
Analyzing the shellcode dynamically will be very tiring because it has multiple checks for Anti-Analysis . Static Analysis will also take much time due to Control Flow Obfuscation. So I wrote 2 Scripts to help in this Analysis . one is to Deobfuscate the Control Flow and the other is to Decrypt the Strings

Please Follow the Below Steps to Reduce your time on Guloader

1. Import the Shellcode to Ghidra
2. Disassemble (Key Binding D) the Start of the Shell code
3. Run the [Guloader\\_deobfuscate.py](#) script Providing the Decryption Key and key for EIP modification
4. Run the [Gu\\_string\\_decryption.py](#) script Providing the Decryption Key

**NOTE:** For Both Cases Provide the keys as Hex String with out Ox Example if the key is 0x6A Provide it as 6A. My Scripts are not the best way to Achieve this . Note to Self need to improve to write better code

After these Steps the Control Flow will be Deobfuscated and Decrypted Strings and Payload key will be printed in the Console . If you have a closer look at the below image you will see the C2 URI will not be starting with http:// this is to prevent it from XOR Bruteforcing



```
Console - Scripting
Guloder_deobfuscate.py> Running...
payload key length is : 772 key on the next line
8dffb051b3a81c55e6548528971fbcd5974212ab982770aa9eb56fa06e5d149310c6b7784bcefl11f790441df88cce67e7a198b:
Guloder_deobfuscate.py> Finished!
Gu_string_decryption.py> Running...
Msi.dll
psapi.dll
user32
SildSublena.utf.by/wp-content/plugins/f8eb81f6deba45169c3b41c05c4590ad/y/mm/mmd/kdRrHFMqRUIujuOy126.bin
C:\Program Files\qga\qga.exe
C:\Program Files\Qemu-ga\qemu-ga.exe
Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:109.0) Gecko/20100101 Firefox/112.0
KERNELBASE.DLL
Publisher
windir=
TEMP=
\system32\
wininet.dll
shell32
mshtml.dll
Startup key
Software\Microsoft\Windows\CurrentVersion\RunOnce
Gu_string_decryption.py> Finished!
```

I Have only Checked one sample with the scripts .Feel free to use it with Other Guloder Shell Code and let me know if you are able to see the Decrypted strings

File SHA1 : 992d98aa6f31ae6f8f42fac9866a19c2a2f879be

## References

1. [SonicWall](#)
2. [CheckPoint](#)
3. [AnyRun](#)

[Back](#) | [Home](#)