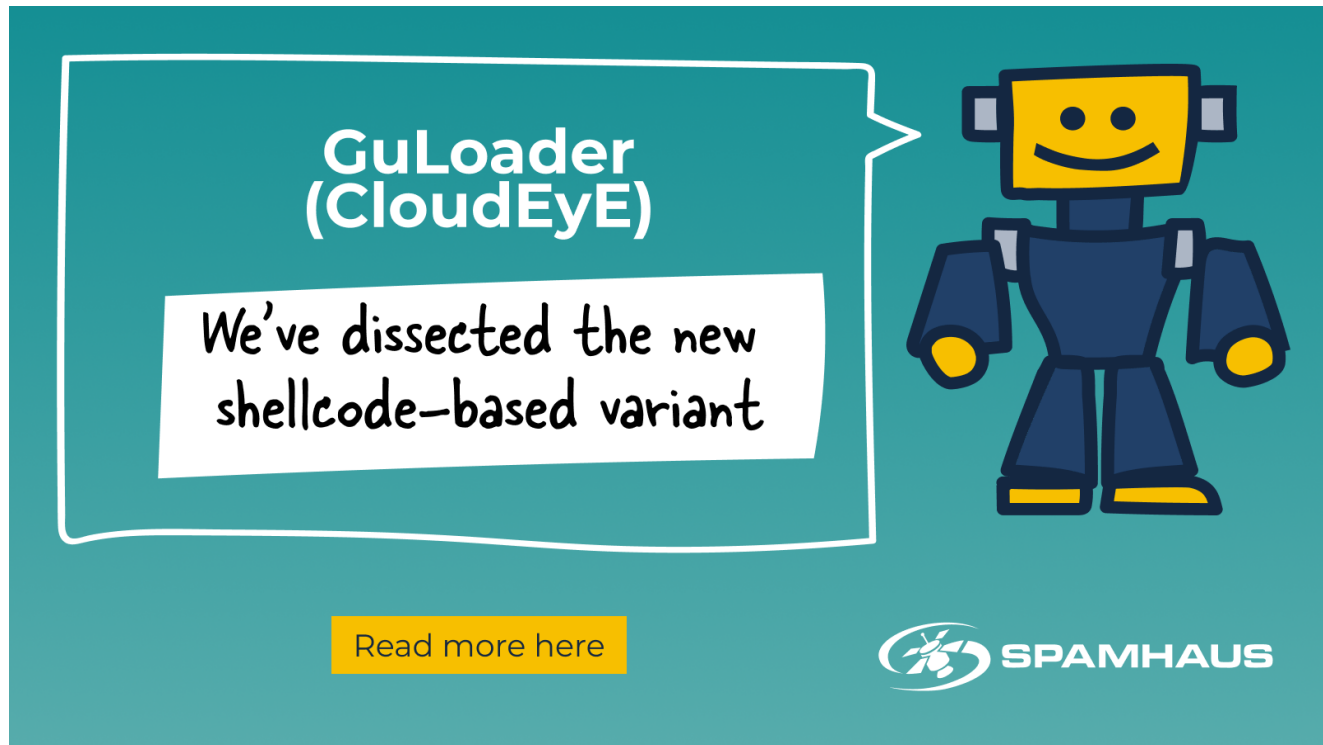


Dissecting the new shellcode-based variant of GuLoader (CloudEyE)

 spamhaus.com/resource-center/dissecting-the-new-shellcode-based-variant-of-guloder-cloudeye/

October 12, 2022



One of the Spamhaus Project's malware specialists has been dissecting GuLoader, attempting to analyze this tricky malware. They have taken time out from reverse engineering and sandbox detonations to share their findings...

What is GuLoader?

GuLoader, or as it is also known, CloudEye, is a small VB5/6 downloader malware. Typically, it downloads Remote Access Tools (RATs) and Stealers, such as Agent Tesla, Arkei/Vidar, Formbook, Lokibot, Netwire, and Remcos, often (but not always), from Google Drive.

What's so special about GuLoader?

GuLoader is notorious for its anti-virtual machine (anti-VM) tactics, i.e., thwarting any attempts for researchers to analyze it. In fact, it was so successful at evading analysis that, at one point, not even one of the most famous online sandboxes could detonate the malware successfully.

Utilizing a packer to swerve detection

GuLoader utilizes the Nullsoft Scriptable Install System (NSIS) packer to compress and encrypt its payload. The NSIS packer is a free, open-source tool commonly used to create Windows installers. However, it can also pack other types of files, such as executables. It is GuLoader's use of the NSIS packer that makes it harder for antivirus programs to detect and remove the malware.

When GuLoader packs its payload, it first compresses the file using the NSIS packer, then encrypts the compressed file with a custom encryption algorithm. The encrypted file is then embedded into the GuLoader executable. When GuLoader is run, it decrypts and unpacks the payload, then executes it.

GuLoader employs a multitude of “anti” strategies

We all know that virtualization is a common way to improve infrastructure efficiency and reduce costs across the IT industry. However, attackers can abuse it to evade detection and launch attacks. Here are some of the ways GuLoader evades detection:

1. **Checking for common VM tools:** GuLoader checks for the presence of common VM tools such as VMware, VirtualBox, and QEMU. If any of these tools are detected, GuLoader will not execute.
2. **Checking for debuggers:** GuLoader checks for the presence of debuggers such as OllyDbg and WinDbg. If a debugger is detected, GuLoader will not execute.
3. **Checking for sandboxes:** GuLoader checks for the presence of sandboxes such as Cuckoo Sandbox and Anubis. If a sandbox is detected, GuLoader will not execute.

We've covered the basic overview; now brace yourselves as we get down and dirty looking under the hood of GuLoader.

How does Guloader resolve an API?

GuLoader uses a hashed technique to resolve an API call; it's a modified version of djb2 – see the example below:

```
mov     ecx, [ebp+1Ch]
cmp     edx, ecx
mov     edx, 474A81E4h
call    ResolveAPICall
mov     [ebp+28h], eax
cmp     ecx, 385A195Ch
mov     ecx, [ebp+1Ch]
mov     edx, 4DD2FD7Dh
test    dx, bx
call    ResolveAPICall
mov     [ebp+0C8h], eax
jmp     short loc_11A0632
```

Binary code obfuscation techniques

GuLoader uses these to make code more difficult to understand and reverse engineer. They work by making the code appear random or meaningless, making it harder for humans to understand what it does.

One of the most common obfuscation techniques is “opaque predicates“, which are Boolean expressions that always return true or false values. However, the value is not known beforehand, making it difficult to understand what it does. Opaque predicates are often used with other obfuscation techniques, such as code permutation, to make the code even more difficult to understand.

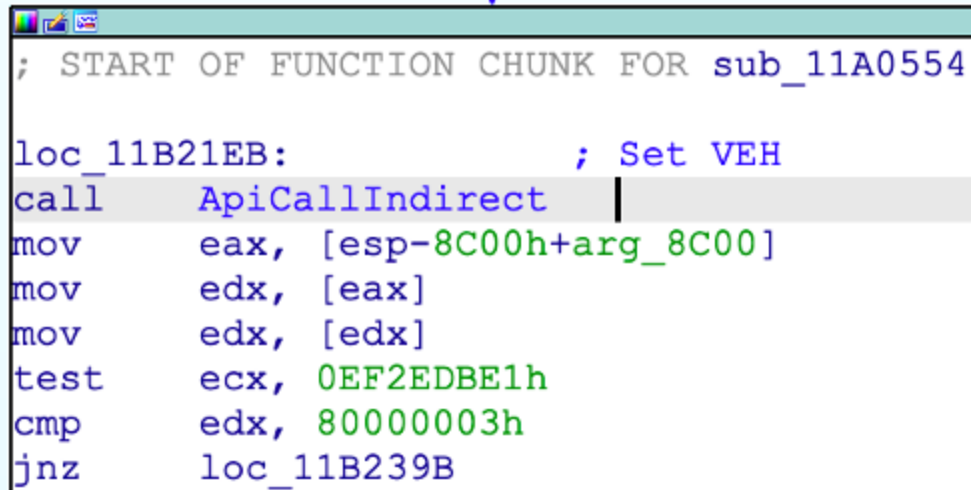
```
loc_11B1361:
cmp     dx, cx
mov     [ebp+223h], edx
mov     edx, 0E3C40EDFh
test    bh, 0FCh
test    edx, eax
sub     edx, 114C188Ah
test    eax, ecx
xor     edx, 8BC46D40h
cmp     dh, ah
sub     edx, 59B39A5Dh
cmp     [ebx+4], dl
mov     edx, [ebp+223h]
jnz    loc_11B123A
```

```
mov     [ebp+176h], eax
test    bl, cl
mov     eax, 859C8587h
cmp     dx, 3268h
xor     eax, 0F15297A3h
test    dl, cl
cmp     cx, bx
xor     eax, 0C04F6496h
add     eax, 4B7E8A10h
cmp     cx, 0EBC0h
cmp     [ebx+10h], al
mov     eax, [ebp+176h]
jnz    loc_11B123A
```

GuLoader uses vectored exceptional handler to change code flow

One interesting feature of GuLoader is how it manages to change the code flow during runtime. This is done using vectored exceptional handler (VEH), a software exception handling mechanism. A VEH can be used to intercept and handle exceptions generated by the operating system or running programs.

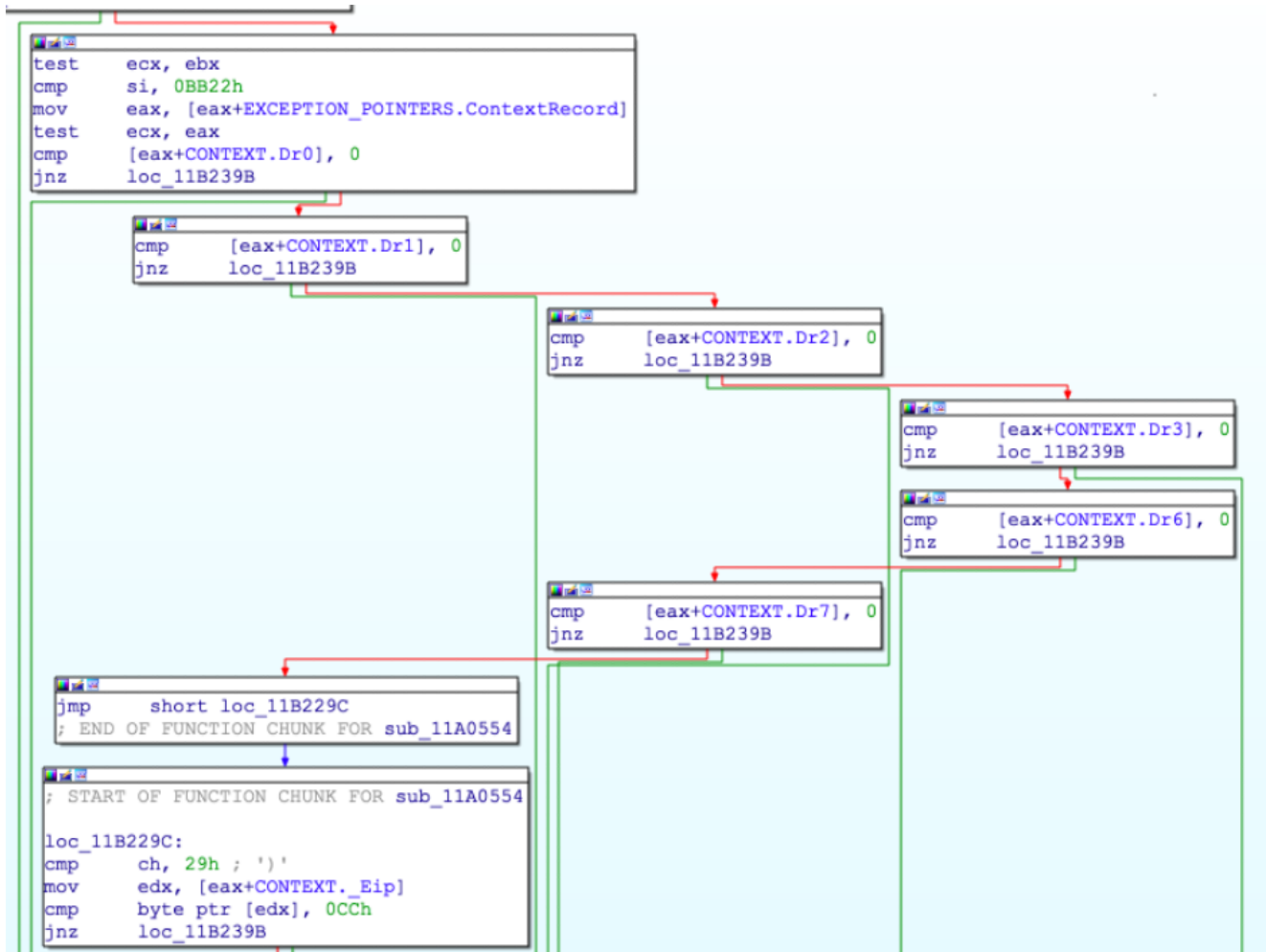
The operating system or program will generate an exception code when an exception occurs. The VEH then looks up the address of the exception handler associated with that exception code and calls it. GuLoader uses VEH to modify the extended instruction pointer (EIP) at an exception to point to the next legitimate instruction. Here's the VEH setup:

A screenshot of a debugger window showing assembly code. The window title is partially visible as "START OF FUNCTION CHUNK FOR sub_11A0554". The code is as follows:

```
; START OF FUNCTION CHUNK FOR sub_11A0554  
loc_11B21EB:           ; Set VEH  
call    ApiCallIndirect  
mov     eax, [esp-8C00h+arg_8C00]  
mov     edx, [eax]  
mov     edx, [edx]  
test    ecx, 0EF2EDBE1h  
cmp     edx, 80000003h  
jnz     loc_11B239B
```

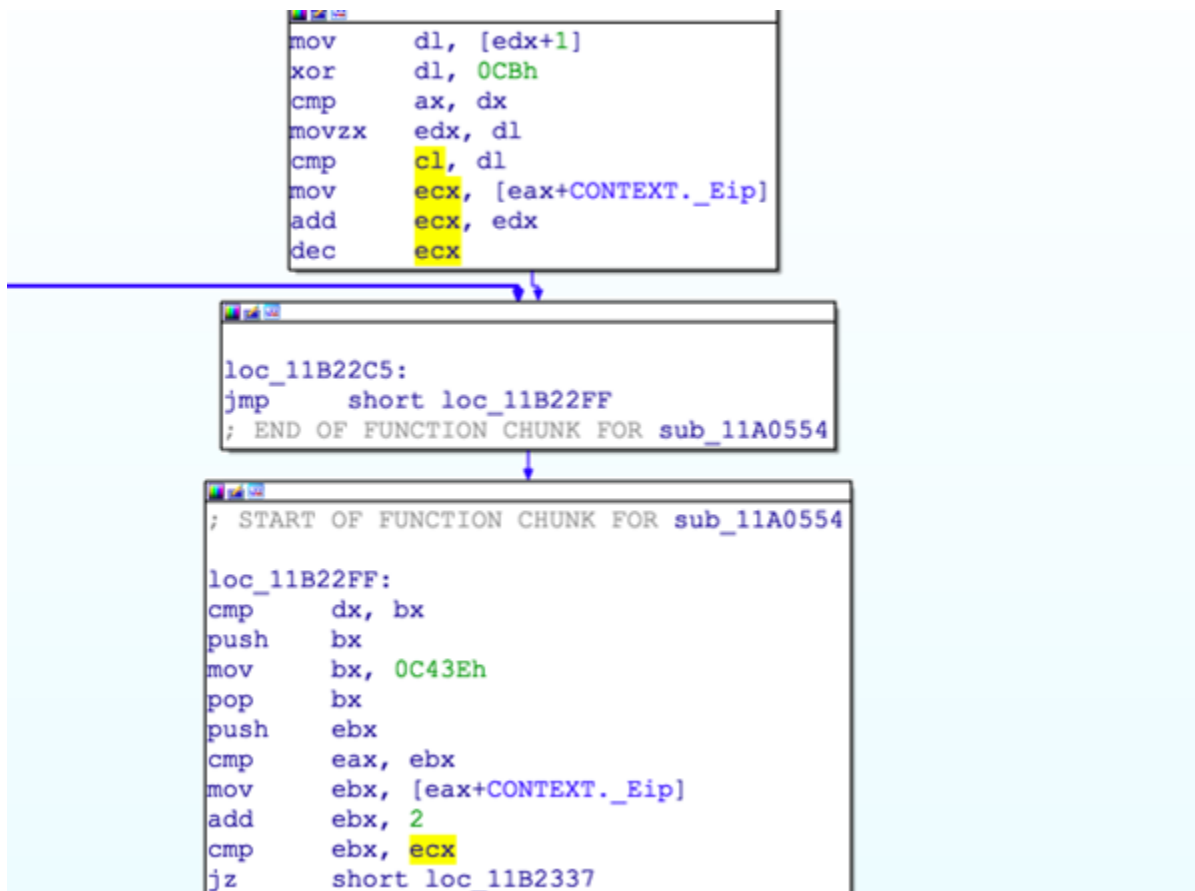
The instruction `call ApiCallIndirect` is highlighted in grey. A blue arrow points down to the top of the window.

Initially, as the exception happens, **_CONTEXTRECORD** structure is checked for the presence of hardware registers, i.e., the x86 debug registers:



The EIP, where the exception happened, is compared against byte 0xcc, i.e., the software break point. This is a necessary condition for the exception to proceed and to generate the next EIP.

The EIP is calculated relative to the place where the exception occurred:



The byte after the exception EIP is XORed with 0xcb, and the result is added to the current EIP to get the next location for execution. The instruction in between the exception EIP and the calculated EIP is filled with junk instructions to confuse the disassembler, as illustrated below:

```

add     [esp+arg_1], 0C43Eh
int     3                ; Trap to Debugger
retn   8BF8h
ApiCallIndirect endp

; -----
;               retf
; -----

jno    short loc_11A06B0
or     al, 0Ah

; ===== S U B R O U T I N E =====

; Attributes: thunk

sub_11A0690    proc near
              jmp     loc_11AD5AF
sub 11A0690    endp

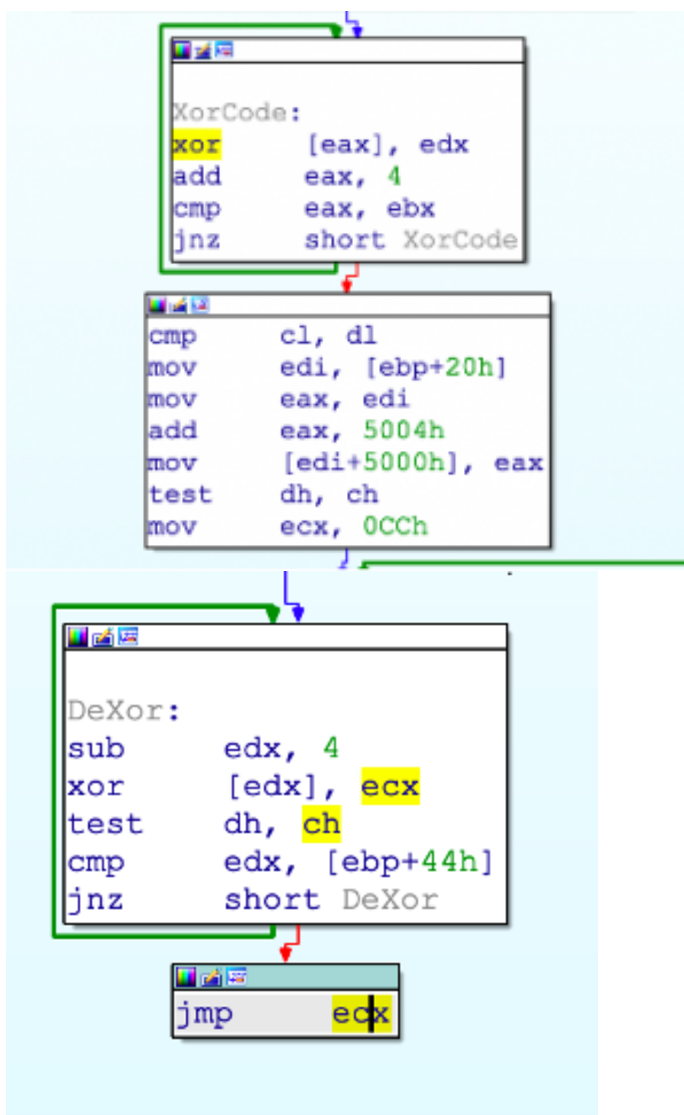
```

How to automate the extraction of indicators of compromise (IOCs) from GuLoader

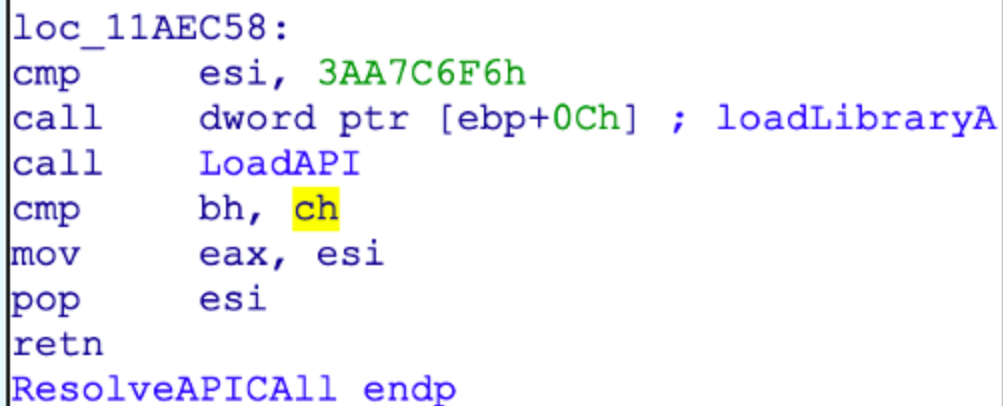
The manual dissection of GuLoader payloads becomes a cumbersome and tedious process due to the presence of all the various hardcore anti-VM, anti-analysis, and anti-debug mechanisms. Therefore, it is imperative to automate this extraction. To make the process successful, we must first automate the dumping and then write a script to extract the parameters necessary to get the URL out of GuLoader.

But GuLoader presents us with a big hurdle, i.e., the anti-dumping protection. This is a technique used to prevent reverse engineering and analysis of the code. It works by encrypting or obfuscating the code, making it exceptionally difficult to read and understand.

GuLoader encrypts the main binary code at any point in the calling of any system API, which invariably makes the dumped code useless. The following two images depict “XORing the code before the call” and “deXORing after the API call”, respectively.



A weakness you can exploit is the initial API call made by GuLoader, which is not wrapped in the subroutine that does all the aforementioned “anti “checks.



```

loc_11AEC58:
cmp     esi, 3AA7C6F6h
call   dword ptr [ebp+0Ch] ; loadLibraryA
call   LoadAPI
cmp     bh, ch
mov     eax, esi
pop     esi
retn
ResolveAPICall endp

```

To exploit this weakness, you can set up a DLL hook after OEP is reached, as shown in the code below:

```

void SetHooks()
{
    DWORD x = 0;
    buffer = getAddr("LoadLibraryA", "kernel32.dll");
    VirtualProtect(buffer, 5, PAGE_EXECUTE_READWRITE, &x);

    memcpy(buffer, CALL_OPCODE(Hook2, buffer), 5);
}

unsigned char EPInst[6];

void __declspec(naked) EPHook()
{
    static int *esp_ = 0;

    __asm mov esp_, ESP

    memcpy(esp_[0] - 5, EPInst, 5);

    SetHooks();
    __asm
    {
        POP EAX
        SUB eax, 5
        PUSH EAX
        ret
    }
}

VirtualProtect(EP + OptHdr.AddressOfEntryPoint, 5, PAGE_EXECUTE_READWRITE, &x);

memcpy(EPInst, EP + OptHdr.AddressOfEntryPoint, 5);
memcpy(EP + OptHdr.AddressOfEntryPoint, CALL_OPCODE(EPHook, EP + OptHdr.AddressOfEntryPoint), 5);

```


Once the dump is successful, we have to locate the key. This process is reasonably straightforward, as GuLoader encrypts the botnet command and controller (C&C) with the same subroutine as the strings. Here's the string decoding subroutine:

```

mov     ebx, [esp+arg_8]
mov     eax, [esp+arg_C]
add     eax, ebx
mov     esi, eax
neg     ebx
mov     edi, ebx
cmp     dl, 0C0h
cmp     ax, 0E4B1h

loc_11AF226:
mov     al, [edx+ecx]
add     ebx, esi
xor     al, [ebx]
sub     ebx, esi
inc     ebx
jnz     short loc_11AF234

mov     ebx, edi

loc_11AF234:
mov     [edx+ecx], al
inc     ecx
jnz     short loc_11AF226

```

The interesting pattern to observe is how the parameters are supplied to the subroutine; being a subroutine with **__stdcall** calling conventions, the stack is cleaned by the called function, but only three parameters are pushed onto the stack.

Tracking back, we can observe that the key parameter is pushed using a direct call opcode. That's precisely the location of the XOR key to be used.

```

; seg000:011AFBD4↓j
call    StrDec
-----
db  51h ; Q
db  0Eh
db  0B8h
db  10h
db  0FDh
db  17h
db  86h
db  36h ; 6
db  0B8h
db  0B3h
db  8Ch
db  0Fh
db  0C4h
db  26h ; &

```

Once you have extracted the key, you can easily brute force for the presence of URLs in the memory dump using the following code:

```
def DecodeC2GuLoader(key , data ):
    FinData = array.array("B")
    for i in range(0, len(data)):
        FinData.append( ( data[i] ^ key[i % len(key)] ) )
        if len(FinData.tobytes()) == 8:
            x = FinData.tobytes()
            x = x[0 : indexNonPrintable(x)]

            if b"http://" not in x:
                return b""

    FinData = FinData.tobytes()

    print ("c2 = %s " % FinData [0 : indexNonPrintable(FinData)])
    return FinData [0 : indexNonPrintable(FinData)]
```

The output will be:

```
python Gu2Extract.py MemDump
GuLoader c2 = b'http://192.3.245.147/2022.bin'
```

Easy as that 😊!

As is evident from what we've discussed, GuLoader is challenging to detect and remove and can pose a severe threat to both individual users and organizations. There are several ways to protect against GuLoader, most of them are IT basics – but sadly, the basics often get missed:

- Keep your software up to date
- Using a reliable antivirus solution
- Train users to be careful when opening email attachments.

Further malware information

To see what malware families our researchers are currently observing in our botnet command and controller (C&C) report, visit our [Botnet Quarterly Update](#).