4 - 6 October, 2023 / London, United Kingdom

TEASING THE SECRETS FROM THREAT ACTORS: MALWARE CONFIGURATION EXTRACTORS

Mark Lim & Zong-Yu Wu

Palo Alto Networks, Singapore & UK

malim@paloaltonetworks.com
zwu@paloaltonetworks.com

ABSTRACT

Malware, like most complex software systems, utilizes the concept of software configuration. Configurations provide directives for how the malware should behave and they are pervasive across the various malware families we analyse. This configuration data embedded in malware can be a goldmine of information about what the malware authors are up to. The main problem is that configuration data in malware is usually difficult to parse statically from the file by design. Malware authors know the intelligence value of these configurations because they give away the marching orders for each instance of the malware, so they're almost always statically armoured.

During the past few years, we have been building a malware configuration extractor system internally at *Palo Alto Networks*. We are now open-sourcing extractors for several malware families in order to share it with the research community. The extractors are written in Python and are designed to scan and extract configuration data from the memory dumps of specific malware samples during dynamic execution.

In this paper we will dive into selected configuration protection techniques that have been utilized by several different malware families. There will be case studies from analysing major families like Trickbot (TheTrick), IcedID (Bokbot) and Emotet (Geodo). There will also be a demonstration of an infostealer (RedLine) compiled in MSIL (.NET). There is common protection design among families as well as customized anti-analysis, both of which we had to tackle in our extractors. Additionally, malware compiled to MSIL (.NET) follows a different Instruction Set Architecture (ISA) and data storage. We will highlight the evolution of the analysis techniques utilized by two malware families: Guloader and Emotet.

Configuration parsing can be fun and useful but it has never been easy. In this paper, we will start from the ground up by introducing what malware configurations are, then taking a deep dive into parsing and extraction. The case studies are comprised of malware families with different purposes that leverage various anti-analysis techniques. Unfortunately, as this cat-and-mouse game goes, this research definitely will not be the end. We hope that by sharing information about these malware configurations we will help everyone prepare for the next uphill battle.

INTRODUCTION

Malware configurations play a critical role in the functionality and behaviour of malicious software. These configurations, also known as command-and-control (C2) configurations, contain important settings that govern how the malware operates, communicates, and carries out its malicious activities.

Malware configurations typically include essential information such as C2 server addresses, port numbers, encryption keys, communication protocols and other settings specific to the malware's intended purpose. These configurations serve as a blueprint for the malware's operations, allowing it to establish communication channels with external entities, receive commands from threat actors, and exfiltrate stolen data.

The C2 configurations are often encrypted or obfuscated to prevent easy detection and analysis by security researchers and anti-virus solutions. Malware authors employ various techniques to hide and protect these configurations, making it challenging for defenders to understand the inner workings of the malware and devise effective countermeasures.

In recent years, the study of C2 configurations has become increasingly important as cybercriminals and advanced threat actors continue to develop sophisticated techniques to evade detection and establish persistence of their malware. Cybersecurity professionals continuously strive to uncover the hidden secrets within C2 configurations to better understand the tactics, techniques and motives behind these malicious campaigns.

In this paper we will delve into the various methods to uncover the secrets hidden within these configurations, providing insights into the evolving nature of malware and the ongoing battle to protect against its threats.

Table 1 lists the malware families and the key protections for their configurations that we will discuss in this paper.

Family	Protection
Guloader	Ciphertext splitting and control flow obfuscation
IcedID	XOR cipher
Trickbot	Erasing configuration after use, mixing decoys with actual C2 IP addresses and XOR cipher
Emotet – v5 in mid-2020	Plaintext C2 list and XOR cipher
Emotet – v6 in late-2021	O-LLVM protected and emulation is required
RedLine	Base64 and XOR cipher
WarZone RAT	RC4 and a variant with customized RC4+

Table 1: Malware families and their key protections.

1. EVOLUTIONARY JOURNEY OF GULOADER'S CONFIGURATION TACTICS

In this section we delve into the progressive evolution of the anti-analysis mechanisms implemented for Guloader's C2 configuration and we undertake the challenge of decrypting the encrypted C2 configuration, overcoming the complex defensive measures implemented in the latest iterations of Guloader.

1.1 Guloader's initial approach to protecting C2 configuration

Guloader is a prominent instrument employed by cybercriminals to propagate and implant diverse forms of malware onto targeted systems. Guloader was first active in 2020, as reported by *CrowdStrike* [1]. All the strings that included this threat's C2 configuration[1] were originally decrypted together in memory. A simple search for the substring 'http' would reveal the C2 configuration.

Figure 1: Searching the C2 configuration using the substring 'http'.

1.2 Additional protections

Since 2022, however, the Guloader authors have gone to great lengths to obfuscate the C2 configuration. Conducting a simple search for 'http' in the memory dumps of Guloader samples no longer yields the C2 configuration because, after decryption, the configurations no longer start with 'http'. Instead, Guloader's authors have added code that dynamically replaces the first eight random characters of the decrypted C2 configuration with the substring 'http://' or 'https://' during runtime.

Due to the absence of direct memory dump search capabilities for the C2 configuration, decrypting the C2 configuration requires knowledge of the following:

1. Encryption routine
2. Encryption key
3. Ciphertext (encrypted C2 configuration)

The encryption routine is a simple XOR loop for decrypting the ciphertext. Presumably, this approach was chosen for its ease of implementation. Furthermore, the offset to the bytes of the encryption key and ciphertext remained constant. As a result, brute-force techniques could be employed to retrieve the C2 configuration, as demonstrated in our blog post [2].

1.2.1 Use of ciphertext splitting

In the latter part of 2022, we observed modifications to the storage of ciphertext, rendering the previously discussed brute-force method ineffective.

In Figure 2, the left side illustrates the previous method of storing the ciphertext. The ciphertext was stored as a continuous sequence of bytes. On the right side, the diagram shows the new method of storing the ciphertext. In this new approach, the ciphertext is computed from a function. In this function, the ciphertext is first divided into four-byte DWORDs. Each DWORD is individually encrypted using randomized mathematical operations.

Figure 2: Comparing old and new methods of storing ciphertext.

[1] Guloader SHA256: bfa5dba46db1253587058b0392c04c8403846fa55d7dcf1044e94e6a654d4715

For example, to retrieve the first DWORD of the ciphertext in Figure 2, we have to compute the following mathematical operations:

```
1  B = [0×CA49C0B7, 0×6EC23534, 0×6D0F0C6D, 0×C984F9D2]
2  b1 = (B[0] ^ B[1]) & 0×FFFFFFFF
3  c1 = (b1 ^ B[2]) & 0×FFFFFFFF
4  result = (c1 - B[3]) & 0×FFFFFFFF
```

Figure 3: An example of computing a DWORD of the ciphertext.

To acquire the complete ciphertext, we would need to perform a series of operations, similar to the one mentioned above, for each individual DWORD. Subsequently, we would proceed to concatenate these DWORDs together, resulting in the formation of the ciphertext.

1.2.2 Control flow obfuscation

Not only did the storage of the ciphertext change, a control flow obfuscation technique was also progressively applied to Guloader samples. In September 2022, we observed many `0xCC` bytes (`INT 3` instructions) littered throughout a Guloader sample. These `INT 3` instructions would trigger the `EXCEPTION_BREAKPOINT` exception, as shown in Figure 4.

```
02212855       pop     eax
02212856       xor     dword ptr [ebx], 0D62B9FCCh
0221285C       int     3                               ; EXCEPTION_BREAKPOINT triggered!
0221285C ; ---------------------------------------------------------------------------
0221285D a2    db 3,13h,19h,15h,'2',12h,0AFh,'IMp|',0E9h,4Dh,4Dh ; junk bytes
0221286B ; ---------------------------------------------------------------------------
0221286B       add     dword ptr [ebx], 3FA3DA06h
02212871       sub     dword ptr [ebx], 0E5EEDCD8h
02212877       int     3                               ; EXCEPTION_BREAKPOINT triggered!
02212877 ; ---------------------------------------------------------------------------
02212878       db 1Dh,'l',0Fh,0B6h,0A3h,8Fh,'o',0A1h,0D5h,34h,0BAh,0AFh,'l',0F6h,8Eh,89h,0B5h ; junk bytes
02212878       db 0A9h
0221288B ; ---------------------------------------------------------------------------
0221288B       add     [eax], eax
0221288D       add     [esi+6B042D54h], bh
02212893       push    ecx
```

Figure 4: A disassembly of Guloader with instructions triggering EXCEPTION_BREAKPOINT.

In Figure 4, the disassembly has been manually improved to highlight the instructions that have been added for anti-analysis. Subsequent to the presence of `0xCC` bytes, extraneous or meaningless instructions were introduced. These additional bytes deliberately disrupted the disassembly process employed by static analysis tools, leading to an incorrect disassembly listing. To address this particular challenge, we provided a solution in the form of an *IDA* processor module extension. In a detailed blog post [3], we outlined the steps and techniques necessary for writing and implementing this extension, enabling accurate disassembly results.

At the beginning of 2023, we discovered a Guloader sample[2] that had an advancement in the implementation of the control flow obfuscation technique. In addition to the previously observed `0xCC` bytes, the malware authors incorporated instructions that deliberately resulted in a null pointer. These instructions would trigger the `EXCEPTION_ACCESS_VIOLATION` exception during runtime. Figure 5 shows the specific instructions that were introduced as part of the anti-analysis measures. The disassembly shown in this image has been manually improved to highlight the instructions added for anti-analysis.

```
02212812       mov     esi, 0D37212A5h
02212817       add     esi, 4C655390h
0221281D       xor     esi, 1FD76635h              ; esi=0
02212823       mov     [esi], esi                  ; EXCEPTION_ACCESS_VIOLATION
02212823 ; ---------------------------------------------------------------------------
02212825       db 1Ch,0C5h,74h,0C3h,4Ch,0F6h,8Fh,0FBh,0FAh,'36',92h,0Eh,0 ; junk bytes
02212833 ; ---------------------------------------------------------------------------
02212833       pop     esi
02212834       mov     dword ptr [ebx], 70609D36h
0221283A       push    eax
0221283B       mov     eax, 0C2520E2Ah
02212840       add     eax, 7711832h
02212845       xor     eax, 6868751Ah
0221284A       sub     eax, 0A1AB5346h             ; eax=0
0221284F       mov     [eax], ecx                  ; EXCEPTION_ACCESS_VIOLATION
0221284F ; ---------------------------------------------------------------------------
02212851       db 0Ah,0B8h                         ; junk bytes
02212851       db 0B8h,'T',0
02212855 ; ---------------------------------------------------------------------------
02212855       pop     eax
```

Figure 5: A disassembly of Guloader with instructions triggering EXCEPTION_ACCESS_VIOLATION.

[2] Guloader SHA256: 32ea41ff050f09d0b92967588a131e0a170cb46baf7ee58d03277d09336f89d9

During our research for this paper, we encountered a Guloader sample[3] that exhibited zero *VirusTotal* (*VT*) detections. In addition to the instructions that caused the `EXCEPTION_BREAKPOINT` and `EXCEPTION_ACCESS_VIOLATION` exceptions, instructions that enabled the trap flag to trigger the `EXCEPTION_SINGLE_STEP` exception were also incorporated. Similar to the case of the `0xCC` bytes, redundant bytes were introduced here as well. Figure 6 illustrates how all these instructions were placed together for anti-analysis.

```
02B993E3 Func_C2_ciphertext_construct:
02B993E3         mov     eax, [esp+4]
02B993E7         int     3                              ; EXCEPTION_BREAKPOINT triggered!
02B993E7 ; ---------------------------------------------------------------------------
02B993E8         db 0B4h,0A1h,'l',13h,0D8h,0FFh,'`+',0A5h,0E3h,30h,26h,'t'
02B993F6 ; ---------------------------------------------------------------------------
02B993F6         push    eax
02B993F7         mov     eax, 433D4EDBh
02B993FC         xor     eax, 9436930Fh
02B99401         xor     eax, 0A78DF586h
02B99406         sub     eax, 392289D1h
02B9940B         sub     eax, 37639D81h                 ; eax=0x100
02B99410         push    ebx
02B99411         pushf
02B99412         mov     ebx, esp
02B99414         or      [ebx], eax                     ; enable Trap flag
02B99416         popf
02B99417         test    edi, ecx                       ; EXCEPTION_SINGLE_STEP triggered!
02B99417 ; ---------------------------------------------------------------------------
02B99419 aW      db 'w',6,0B7h,0B6h,0CFh,14h,'\',0ACh,0A6h,0B8h,'%',0
02B99425 ; ---------------------------------------------------------------------------
02B99425         cmp     ebx, ecx
02B99427         pop     ebx
02B99428         cmp     eax, edx
02B9942A         pop     eax
02B9942B         mov     dword ptr [eax], 3D6E3A33h
02B99431         push    edi
02B99432         mov     edi, 82A8E946h
02B99437         sub     edi, 9707FB40h
02B9943D         sub     edi, 0EBA0EE06h
02B99443         mov     [edi], edi
02B99445         mov     esp, offset unk_78C8DF
02B9944A         pop     edi
02B9944B         xor     dword ptr [eax], 8344C565h
02B99451         xor     dword ptr [eax], 74FD1098h
02B99457         sub     dword ptr [eax], 0CAD7EF8Ch
02B9945D         push    edi
02B9945E         mov     edi, 0AA2AD49Dh
02B99463         add     edi, 676C9CAAh
02B99469         add     edi, 0EE688EB9h                ; edi = 0x0
02B9946F         mov     [edi], ecx                     ; EXCEPTION_ACCESS_VIOLATION triggered!
02B9946F ; ---------------------------------------------------------------------------
```

Figure 6: A disassembly of Guloader with control flow obfuscation, with instructions triggering EXCEPTION_ACCESS_VIOLATION, EXCEPTION_BREAKPOINT and EXCEPTION_SINGLE_STEP.

As a consequence of the addition of instructions that trigger access violations and single-step exceptions, the previous solution of writing an *IDA* processor module extension to counter the anti-analysis technique became ineffective. This is because, whilst in our *IDA* Process Module extension we could detect the `INT 3` (`0xCC`) instruction, due to the variable length nature of *Intel x86* CPU instructions, we could not detect the huge combination of instructions that triggers access violation and single-step exceptions.

In the next section, we will comprehensively outline our approach which successfully circumvents the anti-analysis mechanisms, allowing for the automated decryption of the C2 configuration from the Guloader memory dump.

Before delving into the intricacies of our solution for decrypting Guloader's C2 configuration, let's provide a concise overview of the anti-analysis measures recently implemented by the Guloader authors to safeguard their C2 configuration:

1. **Ciphertext splitting**: The encrypted C2 configuration is divided into multiple DWORDs, which are individual four-byte data units. Each DWORD is then subjected to encryption using randomized mathematical operations. This fragmentation and encryption technique adds an extra layer of complexity to the analysis process.
2. **0xCC (INT 3) instructions**: Guloader incorporates `0xCC` instructions, also known as `INT 3` instructions, within its code. These instructions trigger an `EXCEPTION_BREAKPOINT` at runtime, causing the execution of the malware to halt. This deliberate interruption serves as an anti-analysis measure, preventing static analysis tools from accurately disassembling the code.
3. **Null pointer access instructions**: The malware authors introduced instructions that intentionally cause null pointer access violations during runtime. These instructions lead to an `EXCEPTION_ACCESS_VIOLATION`, where the

[3] Guloader SHA256: beda408709feea7d2023f328e9c97bf4d090bcfb3948fc4e4d9c5c580d8f5858a

program attempts to access a memory location that is invalid or uninitialized. This technique aims to disrupt the program flow and impede analysis by introducing runtime errors.

4. **CPU trap flag instructions**: Guloader also employs instructions that enable the CPU trap flag during runtime, resulting in an `EXCEPTION_SINGLE_STEP` exception. This flag triggers an interrupt after each executed instruction, making the analysis process more challenging as it complicates the normal execution flow and introduces additional breakpoints.

Understanding these anti-analysis measures is crucial in appreciating the complexity and evolving nature of Guloader's defence mechanisms. With this knowledge, we can proceed to explore our solution for decrypting the elusive C2 configuration and overcoming these formidable obstacles.

1.3 Our solution to automatically decrypt C2 configuration from Guloader memory dumps

We begin with our analysis of how the Guloader sample intricately handles three exceptions (`EXCEPTION_BREAKPOINT`, `EXCEPTION_ACCESS_VIOLATION` and `EXCEPTION_SINGLE_STEP`). Let us focus on the Vector Exception Handler (VEH) function within the Guloader sample. The diagram in Figure 7 presents a pseudocode representation of the VEH function, carefully annotated and modified for clarity.

```
int __stdcall Func_VEH_handler(_EXCEPTION_POINTERS ExceptionInfo)
{
  int ExceptionCode;
  CONTEXT *Context;
  _BYTE *Eip;
  int offset;
  _BYTE *i;

  ExceptionCode = *(_DWORD *)ExceptionInfo.ExceptionRecord->ExceptionCode;
  switch ( ExceptionCode )
  {

        case EXCEPTION_BREAKPOINT:
            Context = (CONTEXT *)Func_Anti_HW_Breakpoints();      // return Context if no Hardware BP is detected
            Eip = (_BYTE *)Context->Eip;
            if ( *Eip == 0xCC )
            {
                offset = Eip[1] ^ 0xBB;                              // decrypt offset
                for ( i = (_BYTE *)(offset + Context->Eip - 1); (_BYTE *)(Context->Eip + 2) != i; --i )
                {
                if ( *i == 0xCC )                                 // anti debugging loop
                    return 0;                                                       // exit from handler if debugger is used
                }
                Context->Eip += offset;
                return EXCEPTION_CONTINUE_EXECUTION;                // exception is handled by Func_VEH_handler()
            }
            break;
        case EXCEPTION_ACCESS_VIOLATION:
            if ( *(_DWORD *)(ExceptionInfo.ExceptionRecord->ExceptionCode + offsetof(CONTEXT, Dr7)) )
                return 0;                                                           // exit from handler if debugger is used
            goto EXCEPTION_SINGLE_STEP_ACCESS_VIOLATION;

        case EXCEPTION_SINGLE_STEP:

        EXCEPTION_SINGLE_STEP_ACCESS_VIOLATION:                   //same code for handling both SINGLE STEP and ACCESS VIOLATION exceptions
            Context = (CONTEXT *)Func_Anti_HW_Breakpoints();      // return Context if no Hardware BP is detected
            Context->Eip += *(_BYTE *)(Context->Eip + 2) ^ 0xBB;  // add EIP with decrypted offset
            return EXCEPTION_CONTINUE_EXECUTION;                  // exception is handled by Func_VEH_handler()
  }
  return 0;
}
```

Figure 7: Pseudocode of the VEH function in Guloader.

1.3.1 Analysing EXCEPTION_BREAKPOINT

When confronted with the `EXCEPTION_BREAKPOINT` exception, the VEH function starts by inspecting the utilization of hardware breakpoints. In the absence of hardware breakpoints, the function examines the address pointed to by the EIP for the presence of a `0xCC` byte. To decrypt the offset, the byte following the EIP is XOR'ed with a single-byte key, which varies from sample to sample (in this sample, it's `0xBB`). Subsequently, a loop scans for software breakpoints that could be placed by debuggers (`0xCC` byte) after the EIP-addressed location. If no software breakpoint is detected, the offset is added to the EIP, allowing code execution to resume from the updated EIP position. The updated EIP will be after the extraneous instructions.

1.3.2 Analysing EXCEPTION_ACCESS_VIOLATION and EXCEPTION_SINGLE_STEP

The `EXCEPTION_ACCESS_VIOLATION` and `EXCEPTION_SINGLE_STEP` exceptions share the same handling code. Following the examination of hardware breakpoints, the offset undergoes decryption by XOR'ing the second byte after the EIP with a single-byte key, which is the same one as used for `EXCEPTION_BREAKPOINT`. The resulting offset is then added to the EIP, allowing code execution to proceed from the updated EIP position. The updated EIP will be after the extraneous instructions.

1.3.3 Walking through our solution

For our solution, we utilized the *Unicorn* CPU emulator framework [4] to automatically decrypt the C2 configuration from the Guloader sample memory dump. By leveraging the capabilities of the *Unicorn* framework, we were able to replicate the necessary mathematical operations required to overcome the ciphertext splitting technique employed by the Guloader authors.

One key advantage of using the *Unicorn* framework is its ability to handle multiple exceptions generated by instructions during code execution. We incorporated hooks into our implementation to intercept and handle these exceptions effectively. The handler code within the hooks enables us to update the instruction pointer with the decrypted offset, allowing us to bypass extraneous instructions and continue code execution at the correct decrypted location.

Let's delve into the code snippet that specifically handles the `EXCEPTION_BREAKPOINT` exception. In this snippet, the provided code is registered as a hook for the `EXCEPTION_BREAKPOINT` exception. When this particular exception occurs, the hook function is triggered. The code first checks if the interrupt number is 3, which corresponds to the `EXCEPTION_BREAKPOINT` exception. If it is, the function proceeds to read the byte after the instruction that triggered the exception. By performing an XOR operation between this byte and the key value `0xBB`, we obtain the decrypted offset value. Finally, the function updates the EIP by adding the decrypted offset, effectively redirecting code execution to the correct decrypted location.

```
if intno == 3: #EXCEPTION_BREAKPOINT
    r_eip = uc.reg_read(UC_X86_REG_EIP)
    enc_offset = int.from_bytes(uc.mem_read(r_eip,1),'little')
    offset = enc_offset ^ key
    uc.reg_write(UC_X86_REG_EIP, offset + r_eip -1)
    return True
```

Figure 8: Python code snippet that handles the EXCEPTION_BREAKPOINT exception.

The next code snippet addresses the `EXCEPTION_SINGLE_STEP` exception. This exception is handled in a similar manner to the `EXCEPTION_BREAKPOINT` exception, with a few notable differences. Instead of reading the byte immediately after the triggering instruction, it reads the second byte after that instruction. This difference in byte offset is specific to handling `EXCEPTION_SINGLE_STEP`.

Additionally, in order to prevent the exception from being re-triggered by subsequent instructions, it is necessary to clear the trap flag before continuing with code execution. Clearing the trap flag ensures that the exception is not continuously triggered by subsequent instructions, allowing the program to proceed without interruptions.

```
if intno == 1: # EXCEPTION_SINGLE_STEP
    r_eip = uc.reg_read(UC_X86_REG_EIP)
    enc_offset = int.from_bytes(uc.mem_read(r_eip+2,1),'little')
    offset = enc_offset ^ key
    uc.reg_write(UC_X86_REG_EIP, offset + r_eip)
    # clear the trap flag to continue execution
    uc.reg_write(UC_X86_REG_EFLAGS, uc.reg_read(UC_X86_REG_EFLAGS) & ~0x100)
    return True
```

Figure 9: Python code snippet that handles the EXCEPTION_SINGLE_STEP exception.

In the final code snippet, we address the `EXCEPTION_ACCESS_VIOLATION` exception. Due to limitations in the *Unicorn* framework's memory access handling, we employed a work-around to handle this exception in a somewhat crude manner.

The `hook_code()` function contains code to replace the memory address of a null pointer with an inaccessible memory address. This modification causes the instructions that originally relied on the null pointer to trigger the `EXCEPTION_ACCESS_VIOLATION` exception when executed. This is a way to simulate the behaviour of accessing invalid memory.

```
# callback for tracing instructions
def hook_code(uc, address, size, user_data):
    try:
        uc.mem_unmap(vir_mem_addr, 0x1000)
    except UcError:
        pass

    #overwrite address of null pointer with valid memory address
    registers = [UC_X86_REG_EAX, UC_X86_REG_EBX, UC_X86_REG_ECX, UC_X86_REG_EDX, UC_X86_REG_ESI, UC_X86_REG_EDI]
    for reg in registers:
        val = uc.reg_read(reg)
        if val == 0:
            uc.reg_write(reg, vir_mem_addr)
    return True
```

Figure 10: Python code snippet in the hook_code() function.

When the `EXCEPTION_ACCESS_VIOLATION` exception is encountered, the code snippet presented next serves to update the EIP with the decrypted offset, much like the previous code snippets. However, in order for the execution to proceed smoothly, it becomes imperative to allocate memory at the previously inaccessible memory address. This allocation ensures that the execution can continue without encountering any further memory access problems or interruptions.

It's important to note that this approach is a work-around and may not be the most elegant solution. It addresses the specific limitations of the *Unicorn* framework when it comes to handling memory access with null pointers. A more robust and accurate memory handling mechanism would be preferable.

```
if access == UC_MEM_WRITE_UNMAPPED:
    r_eip = uc.reg_read(UC_X86_REG_EIP)
    enc_offset = int.from_bytes(uc.mem_read(r_eip+2,1),'little')
    offset = enc_offset ^ key
    uc.reg_write(UC_X86_REG_EIP, offset + r_eip)
    registers = [UC_X86_REG_EAX, UC_X86_REG_EBX, UC_X86_REG_ECX, UC_X86_REG_EDX, UC_X86_REG_ESI, UC_X86_REG_EDI]
    for reg in registers:
        val = uc.reg_read(reg)
        if val == vir_mem_addr:
            uc.mem_map(vir_mem_addr, 0x1000)
            return True
```

Figure 11: Python code snippet that handles the EXCEPTION_ACCESS_VIOLATION exception.

1.4 Summary

In this section, we have explored the continuous evolution of anti-analysis techniques employed by the creators of Guloader to safeguard its C2 configuration. We have discussed the challenges faced in decrypting the encrypted C2 configuration and how we have successfully overcome the complex defensive measures implemented in the latest versions of Guloader.

2. LOCATING ICEDID'S CONFIGURATIONS USING YARA

In this section, we document a case study using YARA rules to locate IcedID's encrypted configurations in memory. IcedID, a financially motivated threat, has garnered our attention due to its dynamic nature and continuous evolution. This sophisticated malware operates as a botnet and has established connections with other notorious malware families such as Trickbot and Emotet. Given its malicious intent and interconnectedness, it is crucial for IcedID to effectively hide its C2 configurations.

2.1 IcedID's attack chain

We focused on a common attack chain [5] employed by the authors of IcedID, shown in Figure 12.

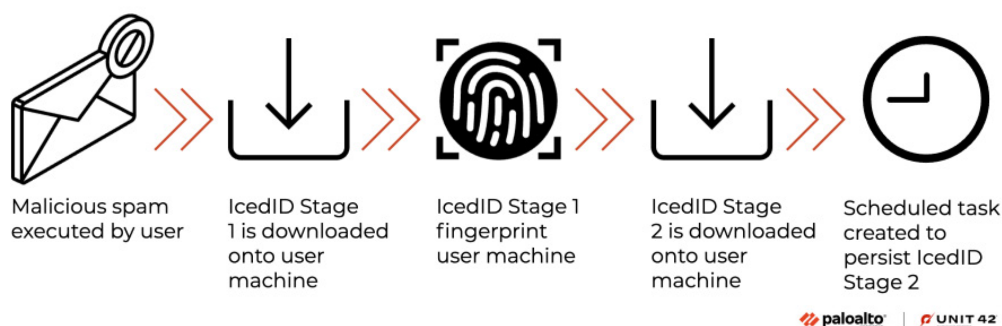

Figure 12: Attack chain of IcedID.

The two-stage attack chain employed by the authors of IcedID served as a clever strategy to conceal their configurations. The IcedID Stage 1 only had a single C2 URL in its configuration, while Stage 2 had more information, such as multiple C2 URLs and campaign IDs. IcedID Stage 2 would only be downloaded if the victim's machine met specific criteria set by the threat actors, adding an additional layer of secrecy. This also allowed the threat actors to have multiple sets of C2 infrastructures, which added complexity to the analysis process. This complexity makes it more difficult for defenders to understand the operation of the threat actors.

While the encryption routines used by both stages of IcedID remained consistent, the authors implemented measures to obscure the locations of the ciphertext. In the following sections, we will focus on using YARA rules to extract the ciphertext. The details of the encryption routines have already been documented in [6].

2.2 IcedID Stage 1

The address of the ciphertext of IcedID Stage 1 is located in the decryption loop for its configuration. We wrote a YARA rule to locate the pointer to the ciphertext, together with instructions of the decryption loop.

The following diagrams show examples of the various implementations of the decryption loop. Our YARA rule had to be flexible enough to locate the pointer to the ciphertext and have low false positive rates.

```
_17c8.dll:000000001DA624D 4C 8D 05 AC 2D 00 00        lea     r8, ciphertext
_17c8.dll:000000001DA6254 0F 1F 40 00                 nop     dword ptr [rax+00h]
_17c8.dll:000000001DA6258 0F 1F 84 00 00 00 00 00     nop     dword ptr [rax+rax+00000000h]

_17c8.dll:000000001DA6260
_17c8.dll:000000001DA6260                            loc_1DA6260:
_17c8.dll:000000001DA6260 42 0F B6 4C 02 40                  movzx   ecx, byte ptr [rdx+r8+40h]
_17c8.dll:000000001DA6266 42 0F B6 04 02                     movzx   eax, byte ptr [rdx+r8]
_17c8.dll:000000001DA626B 32 C8                              xor     cl, al
_17c8.dll:000000001DA626D 88 8C 15 A0 01 00 00               mov     byte ptr [rbp+rdx+430h+var_290], cl
_17c8.dll:000000001DA6274 48 FF C2                           inc     rdx
_17c8.dll:000000001DA6277 48 83 FA 20                        cmp     rdx, 20h ; ' '
_17c8.dll:000000001DA627B 72 E3                              jb      short loc_1DA6260
```

Figure 13: First implementation of IcedID Stage 1 decryption loop.

```
.text:00000001800039CD
.text:00000001800039CD                    loc_1800039CD:
.text:00000001800039CD 48 8D 15 2C 36 00+        lea     rdx, ciphertext
.text:00000001800039CD 00
.text:00000001800039D4 8A 44 11 40               mov     al, [rcx+rdx+40h]
.text:00000001800039D8 32 04 11                  xor     al, [rcx+rdx]
.text:00000001800039DB 88 84 0C E0 00 00+        mov     [rsp+rcx+238h+var_158], al
.text:00000001800039DB 00
.text:00000001800039E2 48 FF C1                  inc     rcx
.text:00000001800039E5 48 83 F9 20               cmp     rcx, 20h ; ' '
.text:00000001800039E9 72 E2                     jb      short loc_1800039CD
```

Figure 14: Second implementation of IcedID Stage 1 decryption loop.

We wrote the following YARA rule to locate the ciphertext in IcedID Stage 1:

```
1  rule IcedID_Config
2  {
3      strings:
4          $IcedID_Config = { (4c|48) 8d ?? ?? ?? 00 00 [0-12] (42|8a) [1-4] 40 [0-5] 32}
5
6      condition:
7          $IcedID_Config
8  }
```

Figure 15: YARA rule to locate the ciphertext in IcedID Stage 1.

This rule also located the ciphertext in a recent IcedID Stage 1 sample[4] discovered by a fellow researcher [7] in May 2023.

[4] IcedID SHA256: 6df2ece892c9192c90d4d9fdec768beb17aecfb17d44adc69a11cb50721fa68e

```
debug046:0000000000241DEB 4C 8D 0D 0E 52 00+        lea     r9, large cs:247000h ; ciphertext
debug046:0000000000241DEB 00
debug046:0000000000241DF2 49 2B C9                  sub     rcx, r9

debug046:0000000000241DF5
debug046:0000000000241DF5                 decryption_loop:
debug046:0000000000241DF5 4B 8D 14 08         lea     rdx, [r8+r9]
debug046:0000000000241DF9 49 FF C0            inc     r8
debug046:0000000000241DFC 8A 42 40            mov     al, [rdx+40h]
debug046:0000000000241DFF 32 02               xor     al, [rdx]
debug046:0000000000241E01 88 44 11 40         mov     [rcx+rdx+40h], al
debug046:0000000000241E05 49 83 F8 20         cmp     r8, 20h ; ' '
debug046:0000000000241E09 72 EA               jb      short decryption_loop
```

Figure 16: IcedID Stage 1 decryption loop of a recent IcedID Stage 1 sample.

2.3 IcedID Stage 2

The configurations within IcedID Stage 2 encompass crucial elements such as C2 URLs and campaign IDs. The campaign IDs serve as essential links, connecting IcedID samples to specific threat actors, enabling further investigation and attribution.

The following diagram shows examples of the various implementations of the pointer to ciphertext. The instructions to load both the pointer to ciphertext and its length as arguments to the description function are deliberately mixed among other function calls.

```
000000018000122B 48 8D 05 CE 1D 00 00                     lea     rax, ciphertext
0000000180001232 41 B8 04 01 00 00                        mov     r8d, 104h
0000000180001238
0000000180001238                                  IcedID_180001238:
0000000180001238 48 89 85 5E 02 00 00                     mov     [rbp+280h+var_22], rax
000000018000123F 48 8D 54 24 52                           lea     rdx, [rsp+380h+Filename+2]
0000000180001244 66 C7 44 24 50 42 01                     mov     word ptr [rsp+380h+Filename], 142h
000000018000124B 48 C7 85 66 02 00 00 5C 02 00 00         mov     [rbp+280h+var_1A], 25Ch ; ciphertext size
0000000180001256 FF 15 A4 0D 00 00                        call    cs:GetModuleFileNameA
```

Figure 17: First implementation of IcedID Stage 2 pointer to ciphertext.

```
0000000180001200 48 8D 05 F9 1D 00+     lea     rax, enc_Config
0000000180001200 00
0000000180001207 66 C7 44 24 50 42+     mov     word ptr [rsp+3C0h+String1], 142h
0000000180001207 01
000000018000120E 48 8D 15 5B 20 00+     lea     rdx, Filename ; .
000000018000120E 00
0000000180001215 48 89 85 9E 02 00+     mov     [rbp+2C0h+var_22], rax
0000000180001215 00
000000018000121C 48 8D 4C 24 52         lea     rcx, [rsp+3C0h+String1+2]
0000000180001221 48 C7 85 A6 02 00+     mov     [rbp+2C0h+var_1A], 25Ch ; enc_config_len
0000000180001221 00 5C 02 00 00
000000018000122C FF 15 CE 0D 00 00      call    cs:lstrcpyA
```

Figure 18: Second implementation of IcedID Stage 2 pointer to ciphertext.

We wrote the following YARA rule to locate the ciphertext in IcedID Stage 2:

```
1  rule IcedID_Config_dat
2  {
3      strings:
4          $IcedID_Config_dat = { 48 8D ?? ?? ?? ?? ?? [0-18] 66 ?? ?? ?? ?? ?? ?? [0-19] 48 ?? ?? ?? ?? ?? ?? 5c 02}
5      condition:
6          $IcedID_Config_dat
7
```

Figure 19: YARA rule to locate the ciphertext in IcedID Stage 2.

The above rule also located the ciphertext in a recent IcedID Stage 2 sample[5] discovered by a fellow researcher in May 2023 [7].

[5] IcedID SHA256: f85d883717d113fcf20afab161470ef2911c729d4d6b04382da0de746b53f0f2

```
0000000180001200 48 8D 05 F9 1D 00+        lea     rax, enc_Config
0000000180001200 00
0000000180001207 66 C7 44 24 50 42+        mov     word ptr [rsp+3C0h+String1], 142h
0000000180001207 01
000000018000120E 48 8D 15 5B 20 00+        lea     rdx, Filename ; .
000000018000120E 00
0000000180001215 48 89 85 9E 02 00+        mov     [rbp+2C0h+var_22], rax
0000000180001215 00
000000018000121C 48 8D 4C 24 52            lea     rcx, [rsp+3C0h+String1+2]
0000000180001221 48 C7 85 A6 02 00+        mov     [rbp+2C0h+var_1A], 25Ch ; enc_config_len
0000000180001221 00 5C 02 00 00
000000018000122C FF 15 CE 0D 00 00         call    cs:lstrcpyA
```

Figure 20: Recent implementation of IcedID Stage 2 pointer to ciphertext.

2.4 Summary

By locating the ciphertext from the memory dumps of IcedID Stages 1 and 2 we could extract their configurations. Some of these C2 URLs in their configurations may not be readily unveiled during the execution of the IcedID binaries. These C2 URLs and the campaign IDs allowed defenders to piece together a network of knowledge on the threat actors using the IcedID malware.

3. REVEALING THE TRICKERY OF TRICKBOT

In this section, we investigate the measures taken by Trickbot to protect its configurations. Trickbot is a highly sophisticated banking trojan and modular malware framework that has been active since 2016. It has evolved over time, incorporating new features and techniques to evade detection and persist on infected systems.

3.1 Flashback to 2018

Back in 2018, the authors of Trickbot left their configuration in memory after decrypting it, allowing it to be easily located via a YARA rule. Figure 21 shows how easy it is to locate Trickbot's configuration from the memory dump of a Trickbot sample[6] first seen in 2018.

```
3C 00 6D 00 63 00 63 00 6F 00+     text "UTF-16LE", '<mcconf><ver>1000158</ver><gtag>ser0328</gtag><serv'
6E 00 66 00 3E 00 3C 00 76 00+     text "UTF-16LE", 's><srv>109.95.113.130:449</srv><srv>87.101.70.109:4'
65 00 72 00 3E 00 31 00 30 00+     text "UTF-16LE", '49</srv><srv>31.134.60.181:449</srv><srv>85.28.129.'
30 00 30 00 31 00 35 00 38 00+     text "UTF-16LE", '209:449</srv><srv>82.214.141.134:449</srv><srv>81.2'
3C 00 2F 00 76 00 65 00 72 00+     text "UTF-16LE", '27.0.215:449</srv><srv>31.172.177.90:449</srv><srv>'
3E 00 3C 00 67 00 74 00 61 00+     text "UTF-16LE", '185.55.64.47:449</srv><srv>78.155.199.225:443</srv>'
67 00 3E 00 73 00 65 00 72 00+     text "UTF-16LE", '<srv>92.63.103.193:443</srv><srv>85.143.175.248:443'
30 00 33 00 32 00 38 00 3C 00+     text "UTF-16LE", '</srv><srv>185.159.129.31:443</srv><srv>194.87.237.'
2F 00 67 00 74 00 61 00 67 00+     text "UTF-16LE", '178:443</srv><srv>195.123.216.12:443</srv><srv>54.3'
3E 00 3C 00 73 00 65 00 72 00+     text "UTF-16LE", '8.56.154:443</srv><srv>82.146.60.85:443</srv><srv>1'
76 00 73 00 3E 00 3C 00 73 00+     text "UTF-16LE", '85.228.232.139:443</srv></servs><autorun><module na'
72 00 76 00 3E 00 31 00 30 00+     text "UTF-16LE", 'me="systeminfo" ctl="GetSystemInfo"/><module name="'
39 00 2E 00 39 00 35 00 2E 00+     text "UTF-16LE", 'injectDll"/></autorun></mcconf>',0
```

Figure 21: Trickbot configuration left in the memory.

3.2 Bags of tricks

In 2022, a Trickbot sample[7] added many more measures to protect its configurations. A function was added to erase the decrypted configurations once utilized. Figure 22 shows the configuration being overwritten with null bytes.

The threat actors then mixed genuine and decoy C2 IP addresses in the configuration. The decoy IP addresses were tagged with '`<srva>`', while the genuine ones were tagged with '`<srv>`'. Figure 23 shows the decoy C2 IP addresses (in the green box) among the genuine IP addresses.

The purpose of the decoy IP addresses was to conceal the genuine IP addresses. To reveal the genuine IP addresses, we utilized a script that decodes the decoy addresses. This script starts by separating the port number from the IP address and converting each octet of the IP address into an integer value. The script then performs a series of XOR operations, similar to the malware code, to decode the values in each octet of the IP address. After decoding the octets, the script computes the port number. Finally, the script constructs the genuine IP address using the decoded octets and port number.

[6] Trickbot SHA256: 2153be5c6f73f4816d90809febf4122a7b065cbfddaa4e2bf5935277341af34c
[7] Trickbot SHA256: 0374bb627e51aa5fa5df0640a5468939cf190a1a1bc0c8a0f3df4bc9b3e92171

```
32BA2960 aMcconfVer10001 db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
32BA2960 db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
32BA2960 db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
32BA2960 db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
32BA2960 db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
32BA2960 db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
32BA2960 db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
32BA2960 db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
32BA2960 db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
32BA2960 db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,':443</srv><srva>128.91.73.5:497'
32BA2960 db '8</srva><srva>181.223.1.108:57103</srva><srva>79.96.133.102:58481'
32BA2960 db '</srva><srva>176.102.19.64:29720</srva><srva>97.238.76.73:41878</'
32BA2960 db 'srva><srva>252.138.69.114:52856</srva><srva>236.20.139.187:40614<'
32BA2960 db '/srva><srva>111.45.199.23:60265</srva><srva>86.213.127.150:44008<'
32BA2960 db '/srva><srva>236.250.70.243:48491</srva><srva>154.106.43.36:16496<'
32BA2960 db '/srva><srva>193.205.3.82:53113</srva><srva>180.10.211.155:55516</'
32BA2960 db 'srva><srva>142.15.255.81:61898</srva><srva>138.141.51.26:48898</s'
32BA2960 db 'rva><srva>55.204.130.62:20238</srva><srva>115.192.20.134:54748</s'
32BA2960 db 'rva><srva>111.15.213.109:56065</srva><srva>142.53.192.107:62709</'
32BA2960 db 'srva></servs><autorun><module name="pwgrab"/></autorun></mcconf>',0
```

Figure 22: Trickbot authors nulling configurations after use.

```
dq offset a67483618449          ; "67.48.36.18:449"
dq 8289E8C400040006h
dq offset a46254128174449       ; "46.254.128.174:449"
dq 8289E8C400040007h
dq offset a41216166142449       ; "41.216.166.142:449"
dq 8289E8C400040008h
dq offset a18114325115444       ; "181.143.251.154:449"
dq 8289E8C400040009h
dq offset a77232163203449       ; "77.232.163.203:449"
dq 8289E8C40004000Ah
dq offset a879717892449         ; "87.97.178.92:449"
dq 8289E8C40004000Bh
dq offset a1859417215449        ; "185.94.172.15:449"
dq 8289E8C40004000Ch
dq offset a185230543443         ; "185.230.5.43:443"
dq 8289E8C40004000Dh
dq offset a912431255443         ; "91.243.125.5:443"
dq 8289E8C40004000Eh
dq offset a18524216811844       ; "185.242.168.118:443"
dq 9691B8A70004000Fh
dq offset a128917354978         ; "128.91.73.5:4978"
dq 9691B8A700040010h
dq offset a18122311085710       ; "181.223.1.108:57103"
dq 9691B8A700040011h
dq offset a79961331025848       ; "79.96.133.102:58481"
dq 9691B8A700040012h
dq offset a17610219642972       ; "176.102.19.64:29720"
dq 9691B8A700040013h
dq offset a97238767341878       ; "97.238.76.73:41878"
dq 9691B8A700040014h
dq offset a25213869114528       ; "252.138.69.114:52856"
dq 9691B8A700040015h
```

Figure 23: Mixing decoy and real C2 IP addresses in the configuration.

```
Please enter script body

 1 fake_C2 = '128.91.73.5:4978'
 2
 3 ip,port = fake_C2.split(':')
 4 octets = [int(i) for i in (ip.split('.'))]
 5 oct0, oct1, oct2, oct3 = octets
 6
 7 tmp0 = oct0 ^ oct2
 8 tmp2 = oct2 ^ oct3
 9 tmp1 = oct1 ^ tmp2
10 tmp3 = oct1 ^ oct2
11
12 v12 = (tmp0 & 0xFF)
13 port = int(port) ^ (v12 ^ (tmp3 << 8 & 0xFF00))
14
15 actual_C2 = (f'{tmp0}.{tmp1}.{tmp2}.{tmp3}:{int(port)}')
16 print(actual_C2)

Line:16   Column:17
Scripting language                Python  Tab size

Output
201.23.76.18:443
```

Figure 24: Python script to decode actual C2 IP addresses.

Fast forward to late 2022, the authors of Trickbot made major changes and adopted the AnchorMail framework [8]. For these samples[8], we decrypted the configuration using the script shown in Figure 25.

```
Please enter script body

 1 enc_cfg = 0x07FEE97CBEB2
 2 enc_cfg_key = 0x000007FEE97CC2B0
 3 enc_cfg_key_b = get_bytes(enc_cfg_key,0x10)
 4 enc_cfg_b = get_bytes(enc_cfg,0x400)
 5
 6 def extract_config(input):
 7     pattern = rb"\{([^}]+)\}"
 8     matches = re.findall(pattern, input)
 9     return matches
10
11 def decode_config(enc_blob, key):
12     key_length = len(key)
13     bytes_clr_config = []
14
15     for i in range(len(enc_blob)):
16         byte_clr = enc_blob[i] ^ key[i % key_length]
17         bytes_clr_config.append(byte_clr)
18
19     bytes_clr_config = bytes(bytes_clr_config)
20     return bytes_clr_config
21
22
23 result = decode_config(enc_cfg_b,enc_cfg_key_b)
24 clr_config = extract_config(result)
25 print(clr_config)

Line:1    Column:1
Scripting language    Python    Tab size    4    Run    Export    Import

Output
[b'batman@naxzs.com|Asdqwe123!!!|box.naxzs.com:993*|box.naxzs.com:465*', b'aquaman@naxzs.com|Asdqwe123!!!|box.naxzs.com:993*|box.naxzs.com:465*', b'joker@naxzs.com']
```

Figure 25: Script to decrypt recent Trickbot sample C2 configuration.

3.3 Summary

We have shown the various techniques used by the authors of Trickbot to protect their configurations from analysis and detection by security researchers. Overall, the authors of Trickbot have demonstrated a significant level of sophistication in their efforts to protect their malware from analysis and detection.

4. DISSECTING THE EVERYDAY HUSTLER: EMOTET (AKA GEODO)

Emotet is an example of a piece of malware for which a configuration extractor can be powerful, but also painful to implement. Back in 2019 when Emotet became a prominent threat, we were able to identify the botnet infrastructure design [9] by clustering the RSA key and the C2 endpoints extracted from the configurations. A configuration extracted from Emotet contained a set of C2 IP addresses plus an RSA key (in the older version) or ECDH key (in the newer version). The C2 IPs were compromised hosts on which a proxy that redirected incoming traffic to the next layer of botnet infrastructure was installed. Once a host was infected by an Emotet sample, a UPnP module was dropped [10] and it became part of the first tier of C2 infrastructure. The C2 IPs were reused for days to weeks, which meant that if the configuration extractor had caught them at time zero, there was a good chance that we would be able to provide protection in the future. This is a success story of configuration extraction that benefited both detection and threat intelligence.

During the time the botnet was active it changed network protocol, configuration protection and obfuscation on a weekly basis. Additionally, the botnet disappeared from time to time, coming back each time with a brand new protocol or a brand new protection.

4.1 Version 5 samples in mid-2020

In the version 5 samples seen in mid-2020, both the configuration and the encrypted resources used by the binary were stored in the `.data` section. We needed to locate the pointer that referenced the encrypted C2 payload and RSA key. The encrypted C2 payload was stored in sequential byte strings that ended in four bytes of `\x00`. We were able to leverage the code pattern shown in Figure 26 to locate the pointer (`0x40A328` in this case).

[8] Trickbot SHA256 52901478b6fb8c1ae7803997708648eaff9e32a93d017fd6945464fb41f3f9a1

```
.text:00401723
.text:00401723                          loc_401723:
.text:00401723 41                       inc     ecx
.text:00401724 89 48 18                 mov     [eax+18h], ecx
.text:00401727 83 3C CD 28 A3 40 00 00 cmp     c2_raw[ecx*8], 0
.text:0040172F 75 F2                    jnz     short loc_401723
```

Figure 26: The code block that loaded the C2 from the binary in Emotet's main binary in mid-2020.

Once we located the pointer and read out the payload, we noticed that the C&C lists were not encrypted at all. Next, we moved on to the RSA key. The RSA public key embedded in the file was used to encrypt the communication protocol, which is only decryptable with the possession of the private key. The version of Emotet in this period used CryptImportKey [11] to load the decrypted RSA key and used the handle to encrypt data later. Our goal was to quickly locate the pointer to the RSA by searching the APIs used. Unfortunately, the *Windows* APIs used by the Emotet main module were obfuscated. The APIs were dynamically loaded using API name hash matching. The following decompiled code snippet shows how HeapFree was dynamically invoked.

```
v5 = lib_get_dll_va_by_hash(0xD85F614E);      // kernel32.dll
v6 = lib_get_api_by_hash(v5, 0xB34B73B0);     // LoadLibraryW
*(g_buffer_49AC94 + 4 * a2) = v6(v13);
v7 = lib_get_dll_va_by_hash(0xD85F614E);      // kernel32.dll
v8 = lib_get_api_by_hash(v7, 0x28552017);     // GetProcessHeap
v12 = v8();
v9 = lib_get_dll_va_by_hash(0xD85F614E);      // kernel32.dll
HeapFree = lib_get_api_by_hash(v9, 0x7E95B6A);// HeapFree
return HeapFree(v12, 0, v4);
```

Figure 27: The decompiled code snippet that dynamically loaded the API. The comments are labelled automatically by our IDA Pro script.

Now, let's talk about how to automatically identify these encrypted APIs. Navigating to the function we labelled as `lib_get_api_by_hash`, we found that the hash algorithm was fairly simple. Note that there is a simple XOR against a random DWORD in both `lib_get_dll_va_by_hash` and `lib_get_api_by_hash`.

```
def name_hash(name, magic):
    v = 0
    for c in name:
        v = (v << 16) + (v << 6) + ord(c) - v
    return (v^magic)&0xFFFFFFFF
```

Figure 28: The equivalent hash function implemented in Python.

Given that every dynamically loaded API had to call these two functions, our first goal was to get the argument passed to the function. First, we manually labelled the `lib_get_dll_va_by_hash` and `lib_get_api_by_hash` functions within *IDA Pro*. We then iterated through the xrefs [12] of `lib_get_dll_va_by_hash` and `lib_get_api_by_hash` to get all the arguments that were passed to the functions. These arguments were the hashed API or DLL name the Emotet sample was looking to invoke. Since hashing is a one-way function, it is designed to digest the input and be irreversible. The best way we could find out what the inputs were was through brute forcing. Given that the *Windows* APIs Emotet needed were those provided by default in a *Windows* system, our approach was to list all the DLLs and their export names in the *Windows* system, compute the hash and check if the output matched any arguments that were passed to the functions `lib_get_dll_va_by_hash` and `lib_get_api_by_hash`. The Python code shown in Figure 29 does the trick to list and compute the export name.

```
def _get_exports(self, pe_f):
    if not os.path.exists(pe_f): return False
    pe = pefile.PE(pe_f)
    expname = []
    if ((not hasattr(pe, 'DIRECTORY_ENTRY_EXPORT')) or
        (pe.DIRECTORY_ENTRY_EXPORT is None)):
            print "[*] No exports for %s" % pe_f
    else:
        for exp in pe.DIRECTORY_ENTRY_EXPORT.symbols:
            if exp.name: expname.append(exp.name)
    return expname
```

Figure 29: An example of a Python script that gets the exports from a given DLL.

For example, we got an argument for `lib_get_api_by_hash` with value `0x12345678` and the `VirtualAllocEx` function also hashed to the same value. This means that the code snippet was dispatching `Kernel32.dll.VirtualAllocEx`. If the API is clear for us, it becomes easy to locate where the `CryptImportKey` is used. Interestingly, the encrypted RSA key is located just above the first C2.

```
v5 = lib_get_dll_va_by_hash(0xD85F614E);      // kernel32.dll
v6 = lib_get_api_by_hash(v5, 0xB34B73B0);     // LoadLibraryW
*(g_buffer_49AC94 + 4 * a2) = v6(v13);
v7 = lib_get_dll_va_by_hash(0xD85F614E);      // kernel32.dll
v8 = lib_get_api_by_hash(v7, 0x28552017);     // GetProcessHeap
v12 = v8();
v9 = lib_get_dll_va_by_hash(0xD85F614E);      // kernel32.dll
HeapFree = lib_get_api_by_hash(v9, 0x7E95B6A);// HeapFree
return HeapFree(v12, 0, v4);
```

Figure 30: The encrypted RSA public key binary payload in IDA Pro screenshot.

```
1   struct resource
2   {
3       DWORD key;
4       DWORD size; // it's encrypted acutal size = size XOR key
5       BYTE data[size];
6   };
7
```

Figure 31: Structure of the encrypted RSA public key in C code format.

4.2 Version 6 samples in late-2021 and 2022

To respond to rapid updates in the detection capabilities of security products against Emotet, the developer behind Emotet adopted agile development and released a new version of the trojan on a weekly basis. In addition, an Obfuscator-LLVM compiler was used to obfuscate the main binary. The frequent changes to Emotet thanks to the source-code level obfuscation raised the cost of analysing new versions and made it impossible to extract configuration statically from the binary. Given that Emotet has become so difficult to analyse, we will share some tips on extracting configuration.

Take a look at the following decompilation of the code snippet where the C2s are set. Each C2 endpoint is computed dynamically from a separate function.

```
__int64 __fastcall cfg_prepare_c2(__int64 a1, __int64 a2, __int64 a3, __int64 a4
{
  _QWORD *v4; // rcx
  unsigned int v5; // ebx
  int i; // eax

  v4 = g_c2;
  v5 = 0;
  for ( i = 1017002; i == 1017002; i = 1044504 )
  {
    g_c2 = mem_alloc(10062682i64, 145835i64, a3, a4, 0x238u);
    if ( !g_c2 )
      return v5;
    v4 = g_c2;
    *(g_c2 + 48) = 0;
  }
  v4[44] = sub_180010D88;
  v4[56] = sub_18002B054;
  v4[26] = sub_180001528;
  v4[38] = sub_18002A4CC;
  v4[41] = sub_18002C2B8;
  v4[11] = sub_18002BF80;
  v4[35] = sub_18002AA74;
  v4[27] = sub_18000DAB0;
  v4[68] = sub_180023584;
  v4[61] = sub_180014644;
  v4[40] = sub_18000FD58;
  v4[43] = sub_180015690;
```

Allocate Memory

C&Cs are computed from these functions

Figure 32: An example of Emotet version 6 setup endpoint configuration in the Decompile view in IDA Pro.

sub_10D88

```
sub_180010D88 proc near

var_18= dword ptr -18h
var_10= dword ptr -10h
var_C= dword ptr -0Ch
arg_0= dword ptr  8
arg_8= dword ptr  10h
arg_10= dword ptr  18h
arg_18= dword ptr  20h

sub     rsp, 18h
mov     [rsp+18h+var_10], 1069Ch
xor     eax, eax
mov     r8, rcx
mov     [rsp+18h+var_C], eax
mov     [rsp+18h+arg_0], 85C265h
mov     r9, rdx
add     [rsp+18h+arg_0], 0FFFF8EB0h
shl     [rsp+18h+arg_0], 0Bh
xor     [rsp+18h+arg_0], 2A8E5524h
mov     eax, [rsp+18h+arg_0]
mov     [rsp+18h+arg_0], eax
mov     [rsp+18h+arg_10], 223256CAh
mov     [rsp+18h+var_18], 79D11385h
mov     [rsp+18h+arg_8], 0ECC4AC1Bh
mov     [rsp+18h+arg_18], 786A1384h
mov     [rsp+18h+arg_0], 2306ECh
imul    eax, [rsp+18h+arg_0], 31h
mov     [rsp+18h+arg_0], eax
imul    eax, [rsp+18h+arg_0], 70h
```

sub_2BF80

```
; __int64 __fastcall sub_18002BF80(_DWORD *, _DWORD *)
sub_18002BF80 proc near

var_28= dword ptr -28h
var_20= dword ptr -20h
var_1C= dword ptr -1Ch
var_18= dword ptr -18h
arg_0= dword ptr  8
arg_8= dword ptr  10h
arg_10= dword ptr  18h
arg_18= dword ptr  20h

sub     rsp, 28h
mov     [rsp+28h+var_20], 83265h
mov     [rsp+28h+var_1C], 424Ah
xor     eax, eax
mov     [rsp+28h+var_18], eax
mov     [rsp+28h+arg_0], 0B78E6Eh
mov     r8, rcx
add     [rsp+28h+arg_0], 0FFFF57A7h
mov     eax, [rsp+28h+arg_0]
add     eax, eax
mov     [rsp+28h+arg_0], eax
xor     [rsp+28h+arg_0], 163FA37h
mov     eax, [rsp+28h+arg_0]
mov     [rsp+28h+arg_0], eax
mov     [rsp+28h+arg_10], 613C4389h
mov     [rsp+28h+var_28], 6E6FC338h
mov     [rsp+28h+arg_8], 0EFDDF5E2h
mov     [rsp+28h+arg_18], 71FFC339h
mov     [rsp+28h+arg_0], 0E4832Bh
add     [rsp+28h+arg_0], 0FFFFBE62h
```

Figure 33: There are pieces of junk code within each function and none of the functions are identical. This causes a big problem for decrypting the payload statically.

This is where the emulation framework came into play. An emulation engine emulates CPU instructions without actually executing them. This is what we used to compute the returned value of each function. There are choices for the tools we could use, such as *Qiling* [13] and *Unicorn* [4]. We used *Unicorn*, but they would both work.

First, we had to locate the function to be emulated. This was not too difficult if we flipped to the Assembly view in *IDA Pro* as we saw repeated and consecutive instructions that stored the function pointer. These function pointers were later invoked to decrypt the actual C&C server.

```
.text:0000000180019146 48 8D 05 3B 7C FF FF    lea     rax, sub_180010D88
.text:000000018001914D 48 89 81 60 01 00 00    mov     [rcx+160h], rax
.text:0000000180019154 48 8D 05 F9 1E 01 00    lea     rax, sub_18002B054
.text:000000018001915B 48 89 81 C0 01 00 00    mov     [rcx+1C0h], rax
.text:0000000180019162 48 8D 05 BF 83 FE FF    lea     rax, sub_180001528
.text:0000000180019169 48 89 81 D0 00 00 00    mov     [rcx+0D0h], rax
.text:0000000180019170 48 8D 05 55 13 01 00    lea     rax, sub_18002A4CC
.text:0000000180019177 48 89 81 30 01 00 00    mov     [rcx+130h], rax
.text:000000018001917E 48 8D 05 33 31 01 00    lea     rax, sub_18002C2B8
.text:0000000180019185 48 89 81 48 01 00 00    mov     [rcx+148h], rax
.text:000000018001918C 48 8D 05 ED 2D 01 00    lea     rax, sub_18002BF80
```

Figure 34: The code block that set the function pointers that decrypted C2s in the configuration setup function in an Emotet v6 binary.

```
29  def emotet_get_c2_pair(code: bytes) -> Dict:
30      emulator = Uc(UC_ARCH_X86, UC_MODE_64)
31
32      # stack
33      stack_start = 0x40000
34      stack_pivot = 0x41000
35      stack_size = 0x10000
36      stack_end = stack_start + stack_size
37      commit_mem(emulator, stack_start, stack_size)
38      emulator.reg_write(UC_X86_REG_RSP, stack_pivot)
39
40      # code
41      code_start = 0x1000000
42      code_size = len(code)
43      code_end = code_start + code_size
44      #the code mem block should be something bigger like 3rd arg
45      commit_mem(emulator, code_start, 0x1000, code)
```

Figure 35: Preparing the stack and the code to be executed by the emulator.

```
47      # ecx, IP
48      ecx_start = 0x8000000
49      ecx_size = 4
50      commit_mem(emulator, ecx_start, 0x1000)
51      emulator.reg_write(UC_X86_REG_RCX, ecx_start)
52
53      # edx, Port
54      edx_start = 0x8100000
55      edx_size = 4
56      commit_mem(emulator, edx_start, 0x1000)
57      emulator.reg_write(UC_X86_REG_RDX, edx_start)
```

Figure 36: The deobfuscated C2 IP address was saved in register ECX, and the port was saved in EDX respectively. We have to allocate the memory for the emulator to write the output.

```
59      emulator.hook_add(UC_HOOK_CODE, code_tracer, None, code_start, code_end)
60      emulator.emu_start(code_start, code_end)
61      rcx = emulator.mem_read(edx_start, edx_size)
62      port = int.from_bytes(rcx[-2:], 'little')
63      rdx = emulator.mem_read(ecx_start, ecx_size)
64      ip = socket.inet_ntoa(rdx)
65      result = {"ip": ip, "port": port}
66      return result
```

Figure 37: Once every argument is ready, launch the emulator. This will get you a list of IPs and ports from the emulation results.

4.3 Summary

Emotet has always offered a good exercise for tracking botnets, from a technical perspective. We have shown that it stored configuration in plaintext in version 5 and then suddenly changed to a configuration format that is impossible to extract without a CPU emulator. There were further challenges in identifying the customized protocol compression and the tricks threat actors used to filter out emulated communication.

5. STUMBLE UPON A NEW SPECIES: WARZONE RAT STEALER

It would take a lot of effort to distinguish WarZone RAT from its close association with AveMaria, since the builder has been leaked several times. We will focus instead on how to extract the configuration from two WarZone RAT variants.

The first variant has existed in the wild for years. The second variant was found when we were developing the configuration extractor for the first one. This is another example of where a configuration extractor can add intelligence value, especially for security product companies that have billions of samples being analysed every day.

5.1 The first species – sample used RC4

WarZone RAT stored a lot of unprotected strings in the payload itself. Once we had unpacked the real payload, we were able to identify the malware easily.

```
20  rule WarZoneRAT160_generic
21  {
22      strings:
23          $s0 = "warzone160" ascii fullword
24          $s1 = ".bss" ascii fullword
25          $s2 = "127.0.0.2" ascii fullword
26          $s3 = "An assertion condition failed" ascii fullword
27      condition:
28          all of them
29  }
```

Figure 38: Example of a generic YARA rule to identify the first variant of the family.

WarZone RAT was developed in C++, which made reverse engineering a bit harder. A small trick here is to press SHIFT+F12 to list all the strings from the sample and find the code referencing '.bss'. This function reads in the encrypted data at the beginning of the .bss section, decrypts it, and sets up the configuration. More of the in-depth analysis can be found in [14].

```
1  int __thiscall cfg_decrypt_and_init(void *this)
2  {
3    // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]
4
5    v31 = this;
6    Sleep(0x1F4u);
7    sub_10014470(v25);
8    v2 = sub_10015E6A();
9    sub_10014367(v2);
10   v3 = sub_10004404(&lpAddress, ".bss");
11   sub_100142DA(v25, v4, v26, v3);
12   sub_10006FD1(lpAddress);
13   sub_10004149(v29, v27);
14   sub_10003F7F(this + 20, v29);
15   mem_free(v29);
16   crypt_cfg_decrypt(this, v28);
```

Figure 39: A decompiled view of a function that reads in binary from the .bss section in WarZone RAT.

The format of the encrypted configuration of WarZone RAT is demonstrated in C code in Figure 40. It starts with an unsigned `INT` that gives out the size of the key. With the size of the key, we were able to read the full content of the ciphertext. The encrypted payload was stored right after the key with a size of `0x40` bytes.

```
1  struct configuration
2  {
3      UInt32 key_size;
4      BYTE key[key_size];
5      BYTE ciphertext[0x40];
6  };
```

Figure 40: The structure of encrypted resources in WarZone RAT in C code.

Having located the configuration, now we have to figure out what protection has been used by the malware. With the help of *IDA Pro* decompiled code and additional knowledge of common cryptography algorithms, we were able to identify the encryption routine as a standard RC4 [15] implementation. There was a symbolic for-loop to initialize the session key array and swap elements in the array based on the key. This procedure is called the key-scheduling algorithm (KSA). After the session key is initialized, a pseudo-random number generator (PRNG) is used to encrypt the plaintext.

```
1  _DWORD *__cdecl crypt_rc4(int a1, int a2, int a3, int a4)
2  {
3    _DWORD *v4; // ecx
4    _DWORD *v5; // esi
5    int i; // eax
6    char v8[256]; // [esp+4h] [ebp-108h]
7    void *v9[2]; // [esp+104h] [ebp-8h] BYREF
8
9    v5 = v4;
10   sub_40304C(v9, &a3);
11   for ( i = 0; i < 256; ++i )                // KSA_INIT
12     v8[i] = i;
13   crypt_rc4_ksa(a2);
14   crypt_rc4_prng(a4);
15   sub_40304C(v5, &a3);
```

Figure 41: The common decompiled structure seen in RC4 implementation from IDA Pro when the source code was not optimized.

Once the cryptographic algorithm was identified, we had to locate the ciphertext. *PEfile* provided us with a handy interface to get the payload from the .bss section. One important thing we checked was that the input file should be in disk format.

```
1  for section in pefile_obj.sections:
2      if section.Name.rstrip(b'\x00') == b'.bss':
3          data = section.get_data()
```

Figure 42: The code snippet that reads in data in the .bss section by PEfile.

5.2 The second species – RC4+

While we had a WarZone RAT configuration extractor in the RC4 decryption scheme implementation, we noticed there were still many configurations that were not correctly extracted. Diving deep into the sample, we found that there was a variant encrypted in a customized RC4+ encryption scheme. Malware authors usually try to implement either customized or not-so-well-known encryption algorithms. There are different approaches to identify the algorithm. A fun approach was to try our luck on every encoding and encryption algorithm in *CyberChef* [16]. In this case, we learned it might be a member of the RC4 family because it looked similar to RC4 from the decompiled code. We identified that the algorithm was a variant of RC4+ by looking through every variant of RC4 on the RC4 *Wikipedia* page.

```
sk = a1->session_key;                    // sk[j+vsk[j]] ^
*(v13 + a2) ^= *((a1->j + _sk_j) + sk) ^ (*((_sk_j + _sk_i) + sk)// _sk_j is now _sk_i since they swap
                                        + *(((*(((32 * a1->j) ^ (a1->i >> 3)) + sk)
                                           + *(((32 * a1->i) ^ (a1->j >> 3)) + sk)) ^ 0xAA)
                                         + sk));
```

Figure 43: Screenshot of the variant implemented decryption based on a customized RC4+. There is one additional XOR 0xAA added to the standard RC4+.

```
while h < len(input_data):
    i += 1

    _k_i = WarzoneRATPlusCrypto._sign_ext(k[i], 8)
    j = c_int32(j + _k_i).value
    k[c_uint8(i).value], k[c_uint8(j).value] = k[c_uint8(j).value], k[c_uint8(i).value]

    _sk_i = k[c_uint8(i).value]
    _aa = k[c_uint8(j + _sk_i).value]

    _sk_i = k[c_uint8(i).value]
    _sk_j = k[c_uint8(j).value]
    _a = k[c_uint8(_sk_i + _sk_j).value]

    _x_i = c_uint8(i << 5 ^ j >> 3).value
    _x = c_int32(WarzoneRATPlusCrypto._sign_ext(k[_x_i], 8)).value
    _y_i = c_uint8(j << 5 ^ i >> 3).value
    _y = c_int32(WarzoneRATPlusCrypto._sign_ext(k[_y_i], 8)).value
    _b = k[c_uint8(c_int32(_x + _y).value ^ 0xFFFFFFAA).value]

    _bb = c_uint8(_a + _b).value
    _K = c_uint8(_aa ^ _bb).value
    output_data.append(input_data[h] ^ _K)
    i += 1
    h += 1
return bytes(output_data)
```

Figure 44: Screenshot of a Python implementation for the PRNG routine of the WarZone RAT Plus customized RC4+.

5.3 Summary

Here we have shared how we discovered the second variant of the WarZone RAT using a configuration extractor. Implementing a configuration extractor is useful for tracking malware updates. It usually breaks the extractor when there is a new update found in the wild.

6. MSIL – ~~MALWARE’S~~ MICROSOFT INTERMEDIATE LANGUAGE: REDLINE STEALER

So far throughout this article we have been discussing configuration extractors for native PE files. Now is a good time to introduce the first configuration extractor for MSIL executables. The case study we will look at is RedLine stealer[9] – it is straightforward, but a different approach is needed. The RedLine payload was packed in an executable. We can either unpack it manually or dump the process memory when executing the malware. The RedLine stealer payload itself was an MSIL executable[10], which can be found on *VirusTotal*.

Just like writing configuration extractors for every other family, the first step is to identify the sample. To do this, we chose to match the code that RedLine stealer uses to check if the victim is located in a post-Soviet state.

```
51      // Token: 0x0400003A RID: 58
52      private static readonly string[] RegionsCountry = new string[]
53      {
54          "Armenia",
55          "Azerbaijan",
56          "Belarus",
57          "Kazakhstan",
58          "Kyrgyzstan",
59          "Moldova",
60          "Tajikistan",
61          "Uzbekistan",
62          "Ukraine",
63          "Russia"
64      };
65  }
```

Figure 45: Decompiled code snippet of blocklisted countries in which RedLine will not execute.

The YARA rule shown in Figure 46 is shared as an example that can be used to identify the malware family. Note that the configuration extractor is running against in-memory payloads. We would avoid checking PE headers (like `uint16be(0) == 0x4d5a` or `pe.is_pe`) as PE headers might be wiped for detection evasion.

```
1   rule RedLine_example
2   {
3       strings:
4           $ = "mscoree.dll" ascii fullword
5           $ = "Armenia" wide fullword
6           $ = "Azerbaijan" wide fullword
7           $ = "Belarus" wide fullword
8           $ = "Kazakhstan" wide fullword
9           $ = "Kyrgyzstan" wide fullword
10          $ = "Moldova" wide fullword
11          $ = "Tajikistan" wide fullword
12          $ = "Uzbekistan" wide fullword
13          $ = "Ukraine" wide fullword
14          $ = "Russia" wide fullword
15          $ = "https://api.ip.sb/ip" wide fullword
16          $ = "0.0.0.0" wide fullword
17      condition:
18          all of them
19  }
```

Figure 46: Example YARA rule to identify RedLine stealer.

Once the family is identified, we have to analyse the sample manually and identify where the configuration is. The sample we had was not name-stripped or obfuscated, and we were able to quickly navigate through each class and discover that the configuration was located inside the class called ‘Arguments’.

Figure 47 is a screenshot from dnSpy showing the encrypted RedLine configuration block.

The configuration seems to be encrypted, so now we have to locate the decryption function. Our preferred approach is always to decrypt the ciphertext using an analyser written in a language we are familiar with. To do this in the dnSpy tool, right-click on the IP field and click analyse. By going through the result, we were able to find the code snippet that is trying to decrypt and parse the configuration.

[9] The sample we are using for this demo is SHA256 a4cf69f849e9ea0ab4eba1cdc1ef2a973591bc7bb55901fdbceb412fb1147ef9
[10] RedLine stealer payload: SHA256 8b2ee4656bc26913c5a85415e8638a9eb8e3f63d352911eae73faeaea009b49f

```
Arguments ✕
1   using System;
2   using System.Reflection;
3
4   // Token: 0x02000017 RID: 23
5   [Obfuscation(ApplyToMembers = true, Exclude = true, StripAfterObfuscation = true)]
6   public static class Arguments
7   {
8       // Token: 0x04000013 RID: 19
9       public static string IP = "HR8RAygYPRI+GTdWPhwgCDsPIwQeDxFE";
10
11      // Token: 0x04000014 RID: 20
12      public static string ID = "857238304-77906307-EZ";
13
14      // Token: 0x04000015 RID: 21
15      public static string Message = "";
16
17      // Token: 0x04000016 RID: 22
18      public static string Key = "Perverters";
19
20      // Token: 0x04000017 RID: 23
21      public static int Version = 2;
22  }
23
```

Figure 47: Screenshot from dnSpy showing the encrypted RedLine configuration block.

```
while (!flag3)
{
    foreach (string address in StringDecrypt.Read(Arguments.IP, Arguments.Key).Split(new char[]
    {
        '|'
    }))
    {
        bool flag4 = connectionProvider.Id1(address);
        if (flag4)
        {
            flag3 = true;
            break;
        }
    }
    Thread.Sleep(5000);
}
```

Figure 48: Arguments.IP is decrypted by the function StringDecrypt.Read with key as an argument.

Digging into `StringDecrypt.Read()`, we found a function that reads in two arguments. The first argument comes with a Base64-encoded string as the ciphertext, and it is followed by a string that is used as the key. The function checks whether the ciphertext b64 is empty (null or a white space, ‘ ’) and if it is not, the ciphertext is decoded from Base64 and decrypted.

```
31      // Token: 0x0600005A RID: 90 RVA: 0x000059A4 File Offset: 0x00003BA4
32      public static string Read(string b64, string stringKey)
33      {
34          string result;
35          try
36          {
37              bool flag = string.IsNullOrWhiteSpace(b64);
38              if (flag)
39              {
40                  result = string.Empty;
41              }
42              else
43              {
44                  string input = StringDecrypt.FromBase64(b64);
45                  result = StringDecrypt.FromBase64(StringDecrypt.Xor(input, stringKey));
46              }
47          }
48          catch
49          {
50              result = b64;
51          }
52          return result;
53      }
54  }
```

Figure 49: The decryption function used in RedLine stealer.

```
// Token: 0x06000057 RID: 87 RVA: 0x000058F8 File Offset: 0x00003AF8
public static string Xor(string input, string stringKey)
{
    StringBuilder stringBuilder = new StringBuilder();
    for (int i = 0; i < input.Length; i++)
    {
        int utf = (int)(input[i] ^ stringKey[i % stringKey.Length]);
        stringBuilder.AppendFormat("{0}", char.ConvertFromUtf32(utf));
    }
    return stringBuilder.ToString();
}
```

Figure 50: The decryption function is straightforward. It is an XOR cipher.

```
@staticmethod
def decrypt(ctx: bytes, key: bytes) -> Optional[bytes]:
    try:
        return b64decode(bytes([a ^ b for (a, b) in zip(b64decode(ctx), cycle(key))]))
    except (ValueError, binascii.Error):
        return None
```

Figure 51: The equivalent code in Python can be implemented as this. Feel free to manually grab example ciphertexts and keys to test if it decrypts correctly.

Next, we locate the configuration and prepare the decrypt function in Python.

For now, we still have to grab the ciphertext and key manually from the decompiled result of *dnSpy*. The next step is to obtain them automatically. When we are writing C code, we access the memory directly, so we sometimes refer to native executables. However, in .NET MSIL, everything is managed. We have a pointer that points to the char array that is stored somewhere in the binary in native C code, but all we see in compiled MSIL are tokens. When these tokens are accessed, the runtime library (CLR) parses out where the token is actually stored and the user does not have to worry about it.

```
Arguments X
1   using System;
2   using System.Reflection;
3
4   // Token: 0x02000017 RID: 23
5   [Obfuscation(ApplyToMembers = true, Exclude = true, StripAfterObfuscation = true)]
6   public static class Arguments
7   {
8       // Token: 0x04000013 RID: 19
9       public static string IP = "HR8RAygYPRI+GTdWPhwgCDsPIwQeDxFE";
10
11      // Token: 0x04000014 RID: 20
12      public static string ID = "857238304-77906307-EZ";
13
14      // Token: 0x04000015 RID: 21
15      public static string Message = "";
16
17      // Token: 0x04000016 RID: 22
18      public static string Key = "Perverters";
19
20      // Token: 0x04000017 RID: 23
21      public static int Version = 2;
22  }
```

Figure 52: The comments generated by dnSpy showed that the string IP is a token number, 0x04000013.

Figure 53 shows the result of opening the sample in *IDA Pro* and navigating to the same function. We see the `ldstr` [17] command push several metadata strings onto the stack (tokens are enclosed by red rectangles) and the tokens (enclosed by purple rectangles) are assigned to corresponding fields by `stsfld` [18]. Both of the strings and fields are referenced as tokens.

```
seg000:29F0                        .method private static hidebysig specialname rtspecialname void .cctor()
seg000:29F0                        {
seg000:29F0                          .maxstack 8
seg000:29F0 00                     nop
seg000:29F1 72 11 09 00 70         ldstr    aHr8raygypriGtd // "HR8RAygYPRI+GTdWPhwgCDsPIwQeDxFE"
seg000:29F6 80 13 00 00 04         stsfld   string Arguments::IP
seg000:29FB 72 53 09 00 70         ldstr    a85723830477906 // "857238304-77906307-EZ"
seg000:2A00 80 14 00 00 04         stsfld   string Arguments::ID
seg000:2A05 72 B5 03 00 70         ldstr    asc_A3B4        // ""
seg000:2A0A 80 15 00 00 04         stsfld   string Arguments::Message
seg000:2A0F 72 7F 09 00 70         ldstr    aPerverters     // "Perverters"
seg000:2A14 80 16 00 00 04         stsfld   string Arguments::Key
seg000:2A19 18                     ldc.i4.2
seg000:2A1A 80 17 00 00 04         stsfld   int32 Arguments::Version
seg000:2A1F 2A                     ret
seg000:2A1F                        }
seg000:2A1F                      }
```

Figure 53: Result of sample being opened in IDA Pro and navigating to the same function.

The .NET framework added an additional mechanism for storing and referencing its token. One additional data directory can be found from the PE header. It points to the CLR (also known as Cor20) header. You will get several stream tables from referencing the metadata header, where you will find the mdToken stored.

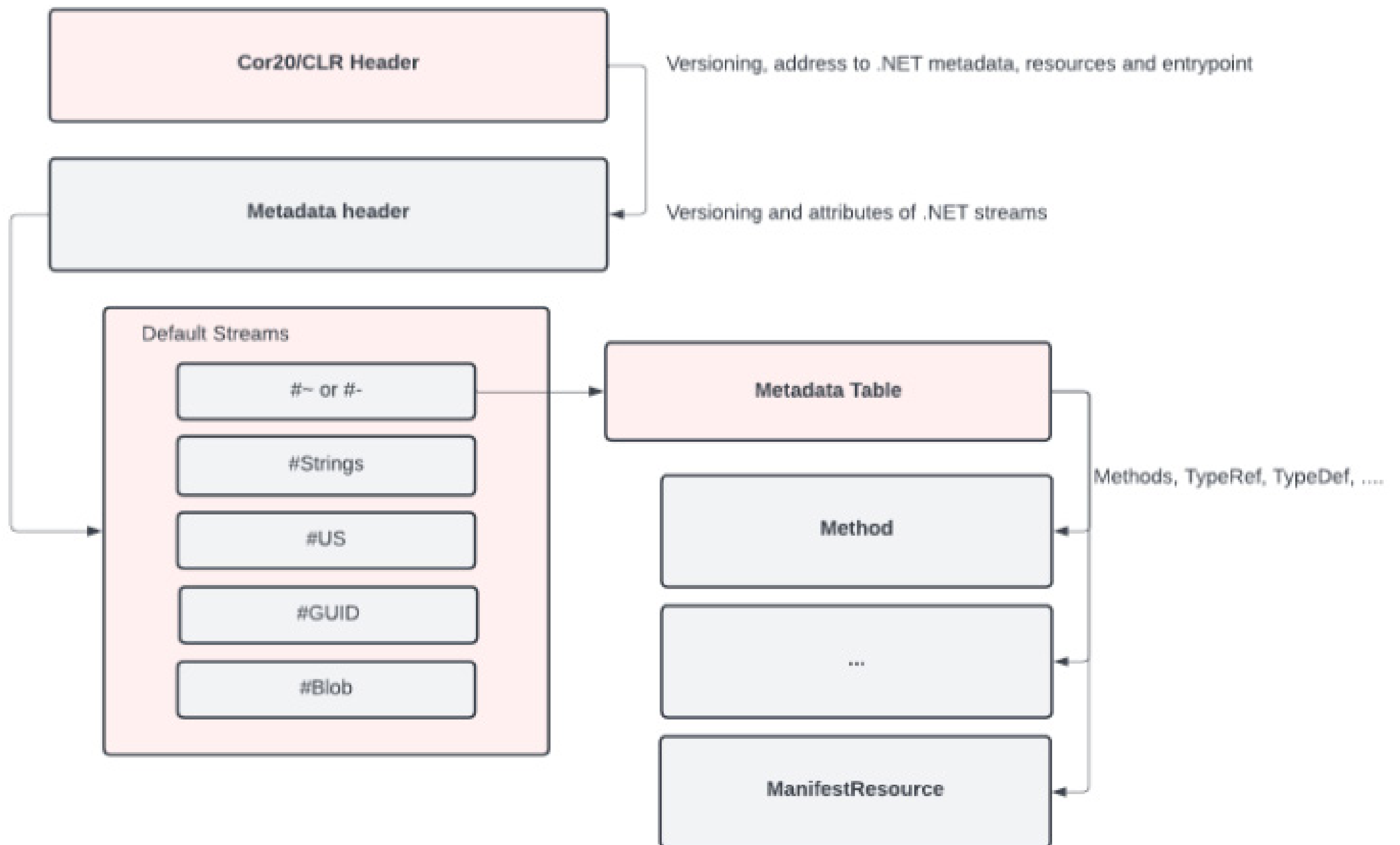

Figure 54: .NET-specific structures and the data they hold.

As shown in Figure 55, the IP field, as well as other fields belonging to the Arguments class, are stored in the metadata stream (#~). These fields are set by `stsfld` commands. Once you click on the token of the IP field, *dnSpy* shows the structure of how the token can be indexed.

Though we have located the IP field, it is still not enough to extract the configuration statically. The source of the string that was pushed onto the stack has not yet been figured out. The operand type of the instruction `ldstr` is a string token according to [19]. The string tokens are stored in the #US (User-Stream) table.

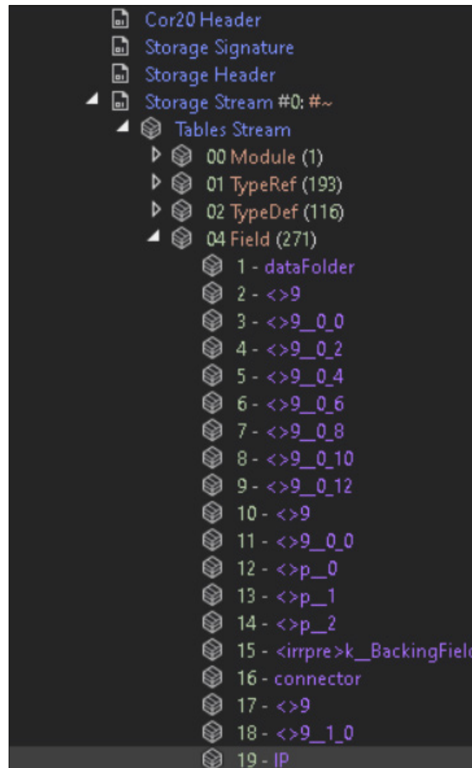

Figure 55: The IP field, as well as other fields belonging to the Arguments class, are stored in the metadata stream (#~).

Format	Assembly Format	Description
72 < T >	ldstr mdToken	Pushes a string object for the metadata string token mdToken.

Figure 56: The documentation of ldstr command usage.

There is a useful open-source library called *dnfile* [20], which is like the .NET version of *PEfile* [21]. The tool allows us to easily access the #US tokens [22] just by giving the RID. It also provides an interface to access the user streams and a lot more [23].

```
 7  def get_us_stream_by_offset(dn: dnfile.dnPE, offset: int) -> Optional[bytes]:
 8      us: dnfile.stream.UserStringHeap = dn.net.metadata.streams.get(b"#US", None)
 9      if not us:
10          return None
11
12      if us.sizeof() == 0:
13          return None
14
15      ret: Optional[Tuple[bytes, int]] = us.get_with_size(offset)
16      if ret is None:
17          return None
18      buf, _ = ret
19      try:
20          s = dnfile.stream.UserString(buf[:-1])
21          return s.value.encode()
22      except UnicodeDecodeError:
23          return None
```

Figure 57: An example implementation using dnfile to get the resource of a given .NET MSIL token.

We used a YARA rule to locate the code snippet where configuration is set up in Figure 53. Then, you can choose your favourite way to parse the 32-bit unsigned integer from the matched code and apply the mask (`0xFFFFFF`) on the token RID. For example, the first five bytes from the matched YARA rule are `72 11 09 00 70`.

opcode	0x72
Token RID	11 09 00 70 = 0x70009011

Table 2: The opcode and token RID for the command ldstr.

Token RID & RID_MASK = 0x9011. This is the value you can pass to the function implemented above. It returns a Base64 string that is encrypted by the key.

```
rule config
{
    strings:
        $ = {
                72 ?? ?? ?? 70
                80 ?? ?? ?? 04
                72 ?? ?? ?? 70
                80 ?? ?? ?? 04
                72 ?? ?? ?? 70
                80 ?? ?? ?? 04
                72 ?? ?? ?? 70
                80 ?? ?? ?? 04
                18
                80 ?? ?? ?? 04
            }
    condition:
        all of them
}
```

Figure 58: The YARA rule to locate the code snippet that sets up the configuration.

This is it! We have a YARA rule to identify the family, then we manually analyse the sample to locate the configuration and determine how the main program reads it in. Later, we figure out there is a simple Base64 encoding plus XOR cipher for the purpose of protection. Once we have everything drawn up, we use another YARA rule (or you can use a regex) to locate the code snippet that contains the token referencing the key and ciphertext. Parse all of the data with the help of *dnfile* and feed them to the decryptor. Boom, you got the configuration.

CONCLUSION

This paper has explored the crucial role of malware configuration extractors in uncovering the hidden secrets within the C2 configurations of various malware families. By delving into the methods used, we have shed light on the process of locating and extracting C2 configurations from six different malware families.

Moreover, in a collaborative effort to contribute to the research community, we have taken the initiative to open-source the Python code and YARA rules that were utilized in our analysis process. We hope this collaborative approach will help the research community and cybersecurity practitioners in their preparation for the inevitable uphill battle against malware.

By leveraging the insights gained from analysing malware configurations, we can enhance our ability to detect, analyse and develop effective countermeasures against malicious software. Through continuous collaboration and knowledge sharing, we can collectively stay ahead of cybercriminals and safeguard our digital systems and networks.

REFERENCES

[1] Wanve, U. GuLoader: Peering Into a Shellcode-based Downloader. Crowd Strike Blog. 25 June 2020. https://www.crowdstrike.com/blog/guloader-malware-analysis/.

[2] Rao, A.; Idrizovic, E.; Rokka Chhetri, S.; Jung, B.; Lim, M. Machine Learning Versus Memory Resident Evil. Unit 42 Palo Alto Networks. 31 January 2023. https://unit42.paloaltonetworks.com/malware-detection-accuracy/.

[3] Lim, M. Defeating Guloader Anti-Analysis Technique. Unit 42 Palo Alto Networks. 28 October 2022. https://unit42.paloaltonetworks.com/guloader-variant-anti-analysis/.

[4] Unicorn. https://www.unicorn-engine.org/.

[5] https://twitter.com/Unit42_Intel/status/1588524735368937484?s=20&t=YXkHyDy_wX1vbbynVm9R6A.

[6] Lim, M.; Raygoza, D.; Jung, B. Teasing the Secrets From Threat Actors: Malware Configuration Parsing at Scale. Unit 42 Palo Alto Networks. 3 May 2023. https://unit42.paloaltonetworks.com/teasing-secrets-malware-configuration-parsing.

[7] https://twitter.com/Unit42_Intel/status/1657014096200343554.

[8] Hammond, C.; Villadsen, O. Trickbot Group's AnchorDNS Backdoor Upgrades to AnchorMail. Security Intelligence. 25 February 2022. https://securityintelligence.com/posts/new-malware-trickbot-anchordns-backdoor-upgrades-anchormail/.

[9] Trend Micro. Exploring Emotet's Activities. https://documents.trendmicro.com/assets/white_papers/ExploringEmotetsActivities_Final.pdf.

[10] Global Research & Analysis Team, Kaspersky Lab. Emotet modules and recent attacks. Secure List. 13 April 2022. https://securelist.com/emotet-modules-and-recent-attacks/106290/.

[11] Microsoft. CryptImportKey function (wincrypt.h). https://learn.microsoft.com/en-us/windows/win32/api/wincrypt/nf-wincrypt-cryptimportkey.

[12] Hey-rays. IDA Help: Xrefs. https://hex-rays.com/products/ida/support/idadoc/313.shtml.

[13] Qiling Framework. https://qiling.io/.

[14] Harakhavik, Y. Warzone: Behind the Enemy Lines. Check Point Research. 3 February 2020. https://research.checkpoint.com/2020/warzone-behind-the-enemy-lines/.

[15] Wikipedia. RC4. https://en.wikipedia.org/wiki/RC4.

[16] CyberChef. https://github.com/gchq/CyberChef.

[17] Microsoft. OpCodes.Ldstr Field. https://learn.microsoft.com/en-us/dotnet/api/system.reflection.emit.opcodes.ldstr?view=net-7.0

[18] Microsoft. OpCodes.Stsfld Field. https://learn.microsoft.com/en-us/dotnet/api/system.reflection.emit.opcodes.stsfld?view=net-8.0.

[19] Dnfile. https://github.com/malwarefrank/dnfile.

[20] Pefile. https://github.com/erocarrera/pefile.

[21] Dnstrings.py. https://github.com/malwarefrank/dnfile/blob/master/examples/dnstrings.py#L10.

[22] Stream.py. https://github.com/malwarefrank/dnfile/blob/master/src/dnfile/stream.py#L59.

INDICATORS OF COMPROMISE (IOCS)

Samples

SHA-256	Malware Family
BFA5DBA46DB1253587058B0392C04C8403846FA55D7DCF1044E94E6A654D4715	Guloader
32ea41ff050f09d0b92967588a131e0a170cb46baf7ee58d03277d09336f89d9	Guloader
beda408709feea7d2023f328e9c97bf4d090bcfb3948fc4e4d9c5c580d8f5858å	Guloader
6df2ece892c9192c90d4d9fdec768beb17aecfb17d44adc69a11cb50721fa68e	IcedID
f85d883717d113fcf20afab161470ef2911c729d4d6b04382da0de746b53f0f2	IcedID
2153be5c6f73f4816d90809febf4122a7b065cbfddaa4e2bf5935277341af34c	Trickbot
0374BB627E51AA5FA5DF0640A5468939CF190A1A1BC0C8A0F3DF4BC9B3E92171	Trickbot
52901478b6fb8c1ae7803997708648eaff9e32a93d017fd6945464fb41f3f9a1	Trickbot
aea7a35212e49f49012cdfffd1439eb1ad9e6e761345b17ebcfbc5a8dd9dd7a5	WarZone RAT
67e04fe16e647e86b2226bae73b17349dfc8c4e8c9521e6caf08557714c2326e	WarZone RAT
a4cf69f849e9ea0ab4eba1cdc1ef2a973591bc7bb55901fdbceb412fb1147ef9	RedLine Stealer
8b2ee4656bc26913c5a85415e8638a9eb8e3f63d352911eae73faeaea009b49f	RedLine Stealer
1a18fcac97501f8482f6d8cfa22c124524b49b11f4b133ce2de51b9196798665	Emotet Version 5
808e8247efd685bdbae3ea0e55de1e8ed8aecd1359a213b0c6291b73f007fdaf	Emotet Version 6

Network Indicators

Guloader

https://drive.google[.]com/uc?export=download&id=1THD-itP7iOm05w_6SQSb-C3tgd3cLMzO

https://gemelw[.]tk/bbb/n

https://blog.nacex[.]es/wp-content/plugins/DRCOOOL/JtPpWnyvrb11.bin

IcedID

Campaign ID = 10663848

C2 url = nedgogolinh.com

Bot ID = 4049493703 URI = /news/

C2 URLs =

treylercompandium[.]com

gabrikxuira[.]com

Trickbot

<mcconf><ver>1000158</ver><gtag>ser0328</gtag><servs><srv>109.95.113.130:449</srv><srv>87.101.70.109:449</
srv><srv>31.134.60.181:449</srv><srv>85.28.129.209:449</srv><srv>82.214.141.134:449</
srv><srv>81.227.0.215:449</srv><srv>31.172.177.90:449</srv><srv>185.55.64.47:449</srv><srv>78.155.199.225:443</
srv><srv>92.63.103.193:443</srv><srv>85.143.175.248:443</srv><srv>185.159.129.31:443</
srv><srv>194.87.237.178:443</srv><srv>195.123.216.12:443</srv><srv>54.38.56.154:443</
srv><srv>82.146.60.85:443</srv><srv>185.228.232.139:443</srv></servs><autorun><module name=”systeminfo”
ctl=”GetSystemInfo”/><module name=”injectDll”/></autorun></mcconf>

{‘bot_ver’: ‘100015’, ‘gtag’: ‘mon169’, ‘autorun’: ‘pwgrab’, ‘c2_list’: [

‘67.48.36.18:449’, ‘46.254.128.174:449’,’41.216.166.142:449’, ‘181.143.251.154:449’, ‘77.232.163.203:449’, ‘87.97.178.
92:449’,’185.94.172.15:449’, ‘185.230.5.43:443’, ‘91.243.125.5:443’, ‘185.242.168.118:443’,

‘201.23.76.18:443’, ‘180.178.109.222:443’,

‘202.131.227.229:443’, ‘163.53.83.117:443’, ‘45.235.5.162:443’, ‘185.189.55.207:449’,

‘103.36.48.159:449’,

‘168.253.208.234:449’, ‘41.60.233.170:449’, ‘170.79.181.188:449’, ‘177.101.15.65:449’,

‘194.156.81.206:443’,

‘103.66.72.217:443’, ‘113.161.174.240:443’, ‘185.164.41.190:443’, ‘181.112.188.78:443’,

‘103.82.146.212:443’, ‘186.183.184.218:443’, ‘78.158.171.245:443’]}

[‘batman@naxzs.com|Asdqwe123!!!|box.naxzs.com:993*|box.naxzs.com:465*’, ‘aquaman@naxzs.com|Asdqwe123!!!|box.
naxzs.com:993*|box.naxzs.com:465*’, b’joker@naxzs.com’]

WarZone RAT

rajsavindia.hopto[.]org:5067

192.3.111[.]154:5200

RedLine Stealer

37.220.87[.]13:40676

Emotet Version 5

173.73.87.96:80
71.222.233.135:443
60.250.78.22:443
80.86.91.91:8080
104.236.28.47:8080
162.241.92.219:8080
74.208.45.104:8080
178.20.74.212:80
85.105.205.77:8080
190.220.19.82:443
78.24.219.147:8080
47.26.155.17:80
110.44.113.2:80
113.52.123.226:7080
120.151.135.224:80

108.191.2.72:80
70.127.155.33:80
98.156.206.153:80
47.6.15.79:443
104.131.44.150:8080
60.231.217.199:8080
70.184.9.39:8080
223.197.185.60:80
121.88.5.176:443
211.192.153.224:80
5.196.74.210:8080
70.187.114.147:80
190.143.39.231:80
24.164.79.147:8080
110.36.217.66:8080
24.94.237.248:80
47.156.70.145:80
125.207.127.86:80
108.6.140.26:80
173.24.68.195:80
105.27.155.182:80
189.212.199.126:443
47.153.183.211:80
75.114.235.105:80
160.16.215.66:8080
190.55.181.54:443
95.128.43.213:8080
190.146.205.227:8080
101.187.197.33:443
174.53.195.88:80
190.114.244.182:443
31.172.240.91:8080
100.6.23.40:80
76.104.80.47:443
103.86.49.11:8080
200.21.90.5:443
104.236.246.93:8080
139.130.242.43:80
181.126.70.117:80
188.0.135.237:80
217.160.182.191:8080
181.13.24.82:80
87.106.139.101:8080
205.185.117.108:8080
210.6.85.121:80
78.189.180.107:80
202.175.121.202:8090
88.249.120.205:80
209.141.54.221:8080
182.176.132.213:8090
136.243.205.112:7080
173.21.26.90:80
62.138.26.28:8080
209.97.168.52:8080

207.177.72.129:8080
41.60.200.34:80
211.63.71.72:8080
178.153.176.124:80
90.69.145.210:8080
152.168.248.128:443
85.152.174.56:80
47.155.214.239:443
31.31.77.83:443
139.130.241.252:443
115.65.111.148:443
37.187.72.193:8080
190.12.119.180:443
190.117.126.169:80
46.105.131.87:80
101.100.137.135:80
87.106.136.232:8080
201.184.105.242:443
201.173.217.124:443
177.239.160.121:80
74.108.124.180:80
218.255.173.106:80
24.105.202.216:443
76.104.80.47:80
173.16.62.227:80
195.244.215.206:80
45.55.65.123:8080
47.6.15.79:80
65.184.222.119:80
37.139.21.175:8080
46.105.131.69:443
101.187.134.207:8080
200.116.145.225:443
149.202.153.252:8080
108.190.109.107:80
78.142.114.69:80
179.13.185.19:80
222.144.13.169:80
66.34.201.20:7080
176.9.43.37:8080
45.33.49.124:443
186.6.245.26:443
93.147.141.5:443
78.186.5.109:443
47.155.214.239:80
62.75.187.192:8080
101.187.237.217:80
68.114.229.171:80
209.146.22.34:443
62.75.141.82:80
78.101.70.199:443
95.213.236.64:8080
91.205.215.66:443
76.86.17.1:80

181.143.126.170:80
5.32.55.214:80
60.142.249.243:80
70.180.35.211:80

Emotet Version 6

51.178.61.60:443
168.197.250.14:80
45.79.33.48:8080
196.44.98.190:8080
177.72.80.14:7080
51.210.242.234:8080
185.148.169.10:8080
142.4.219.173:8080
78.47.204.80:443
78.46.73.125:443
37.44.244.177:8080
37.59.209.141:8080
191.252.103.16:80
54.38.242.185:443
85.214.67.203:8080
54.37.228.122:443
207.148.81.119:8080
195.77.239.39:8080
66.42.57.149:443
195.154.146.35:443