Tackling Anti-Analysis Techniques of GuLoader and RedLine Stealer

By Mark Lim, Zong-Yu Wu

Published: 2024-01-05 · Archived: 2026-04-05 13:51:36 UTC

Executive Summary

Malware, like many complex software systems, relies on the concept of software configuration. Configurations establish guidelines for malware behavior and they are a common feature among the various malware families we examine. The configuration data embedded within malware can offer invaluable insights into the intentions of cybercriminals. However, due to its significance, malware authors deliberately make configuration data challenging to parse statically from the file.

Over the past few years, we have developed a system to extract internal malware configurations. We will share code from our extractors for multiple malware families with the research community. These extractors, written in Python, are designed to scan and extract configuration data from memory dumps associated with specific malware samples.

We will also introduce selected configuration protection techniques employed by two malware families: GuLoader and RedLine Stealer. For those interested in more details, please look into the whitepaper, slides or video we presented at Virus Bulletin 2023 in London.

Palo Alto Networks customers are better protected from these threats through our Next-Generation Firewall with cloud-delivered security services including WildFire. If you think you might have been compromised or have an urgent matter, get in touch with the Unit 42 Incident Response team.

Technical Analysis of GuLoader

The GuLoader authors went to great lengths to obfuscate their C2 configuration. Figure 1 provides a timeline illustrating the evolution of GuLoader obfuscation techniques.

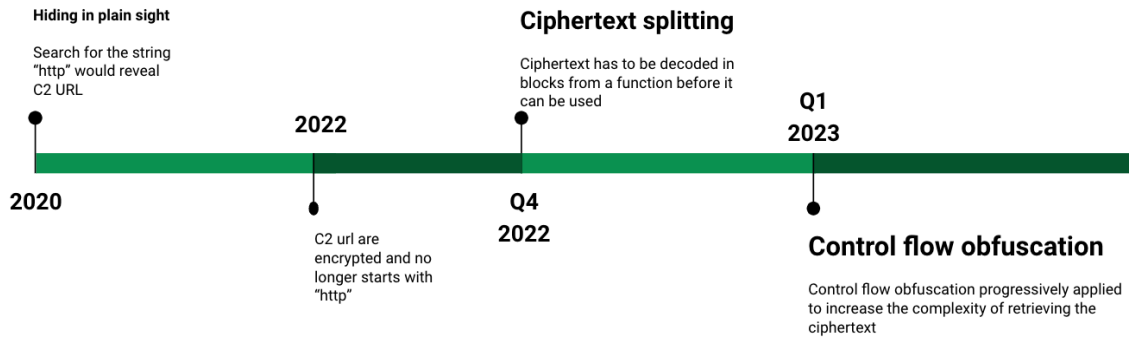

Figure 1. Timeline showing the evolution of obfuscation techniques used by Guloader.

This evolution has defeated our previous approach to extracting GuLoader malware configuration. The GuLoader authors' newer techniques include ciphertext splitting and control flow obfuscation.

Ciphertext Splitting

We have labeled GuLoader's previous method of storing encrypted configuration data (ciphertext) in the top section of Figure 2 as the "old method." In this old method, the ciphertext was stored as a continuous sequence of bytes.

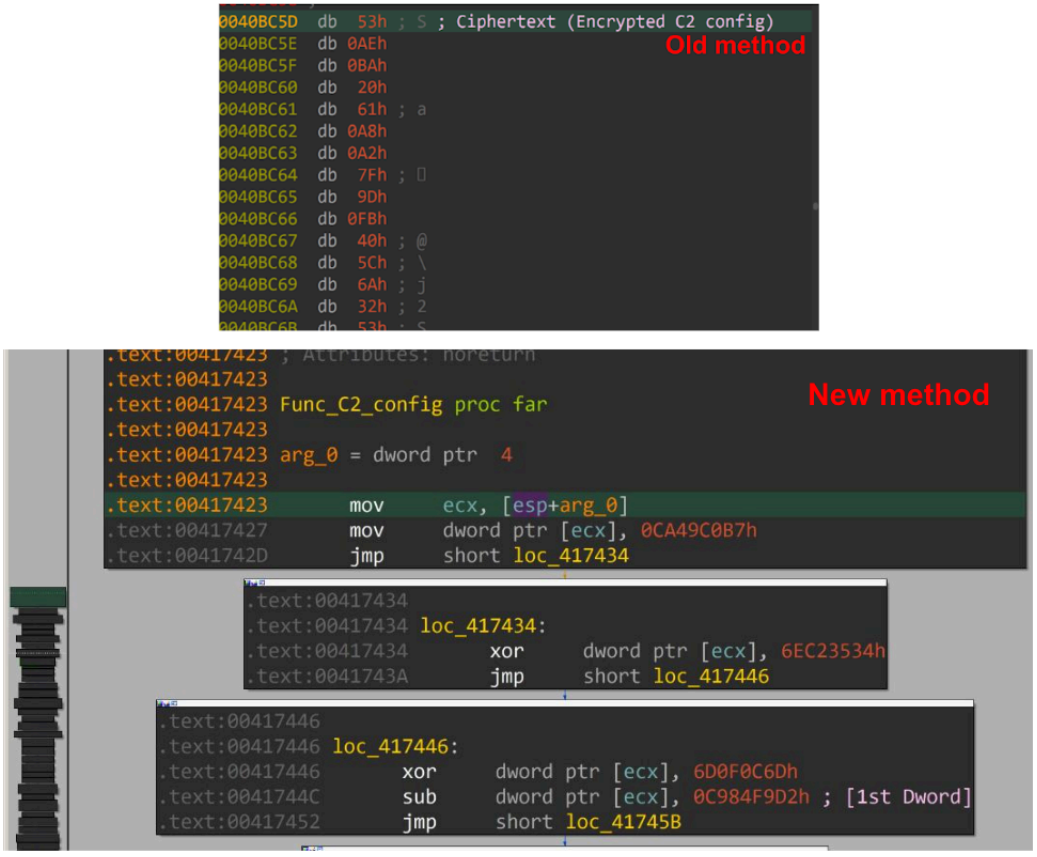

Figure 2. Comparing old and new methods of storing ciphertext.

In the lower section of Figure 2 above, we have labeled the new approach as GuLoader's "new method," where the ciphertext is computed from a function. In this function, the ciphertext is first divided into a 4-byte DWORD. Each DWORD is individually encrypted using randomized mathematical operations.

For example, to retrieve the first DWORD of the ciphertext from GuLoader's new method, we must perform the mathematical operations illustrated below in Figure 3.

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

Figure 3. An example of computing a DWORD of the ciphertext from Guloader's new method of storing ciphertext.

To acquire the complete ciphertext from this new method, we perform a series of operations similar to the method shown in Figure 3 above for each individual DWORD. Subsequently, we concatenate these DWORD values together, resulting in the complete ciphertext.

Control Flow Obfuscation

In early 2023, we encountered a GuLoader sample that originally had zero VirusTotal (VT) detections. Using Hex-Rays IDA Pro to disassemble and analyze this malware sample, we found instructions that attempted to prevent further analysis. These anti-analysis instructions were designed to cause EXCEPTION_BREAKPOINT, EXCEPTION_ACCESS_VIOLATION and EXCEPTION_SINGLE_STEP violations.

Figure 4 illustrates how GuLoader implemented all these instructions 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,'ŧ'
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 4. Dissembler analysis of the Guloader sample revealing the anti-analysis instructions.

The anti-analysis instructions noted in Figure 4 above rendered our previous solution of writing an IDA processor module extension ineffective. Due to the variable nature of the length of Intel x86 CPU instructions, we could not detect the huge combination of instructions that triggered EXCEPTION_ACCESS_VIOLATION and EXCEPTION_SINGLE_STEP exceptions.

Since our previous solution was no longer effective, we had to manually analyze the code to find these anti-analysis instructions and bypass them to extract the configuration. We explained in detail how we extracted the configuration in our whitepaper for Virus Bulletin.

Technical Analysis of RedLine Stealer

The SHA256 hash for the RedLine Stealer sample used in this analysis is a4cf69f849e9ea0ab4eba1cdc1ef2a973591bc7bb55901fdbceb412fb1147ef9. Using an MSIL decompiler called dnSpy, we quickly identified the configuration data as shown below in Figure 5.

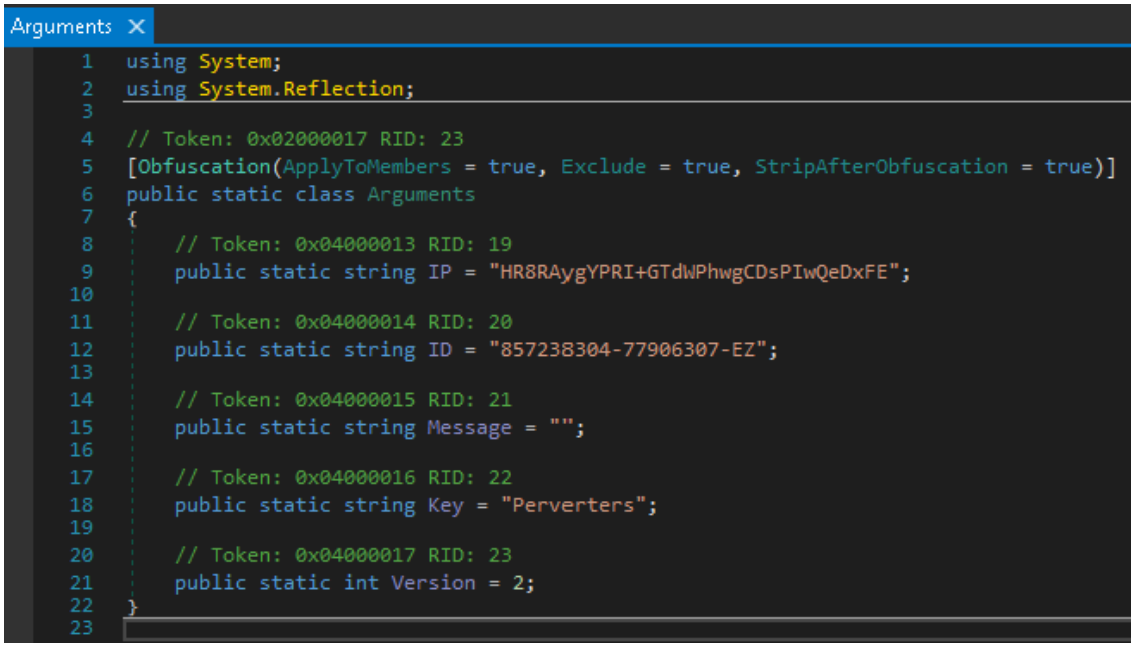

Figure 5. Screenshot taken from dnSpy that contains the RedLine Stealer sample's encrypted configuration block.

We implemented a decryption routine in Python as shown below in Figure 6. We invite readers to manually grab example ciphertexts and keys to test whether the script from Figure 6 decrypts correctly.

```
@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 6. The decryption routine written in Python.

Next, we located the configuration (shown in Figures 7 and 8) and prepared the decrypt function in Python. However, before decrypting the data, we had to manually grab the ciphertext and key from the decompiled result

generated by dnSpy.

When writing C code, we directly access system memory, so we sometimes call the executables compiled from C code as native executables. However, in .NET MSIL, everything is managed. A pointer leads to the character array stored somewhere in the binary in native C code, but all we see in the compiled MSIL are the tokens.

When accessing these tokens, the runtime library (CLR) parses where the ciphertext is actually stored, which is one less thing for an analyst to worry about. For example, in Figure 7 comments generated by dnSpy show that the string IP is a token number 0x04000013.

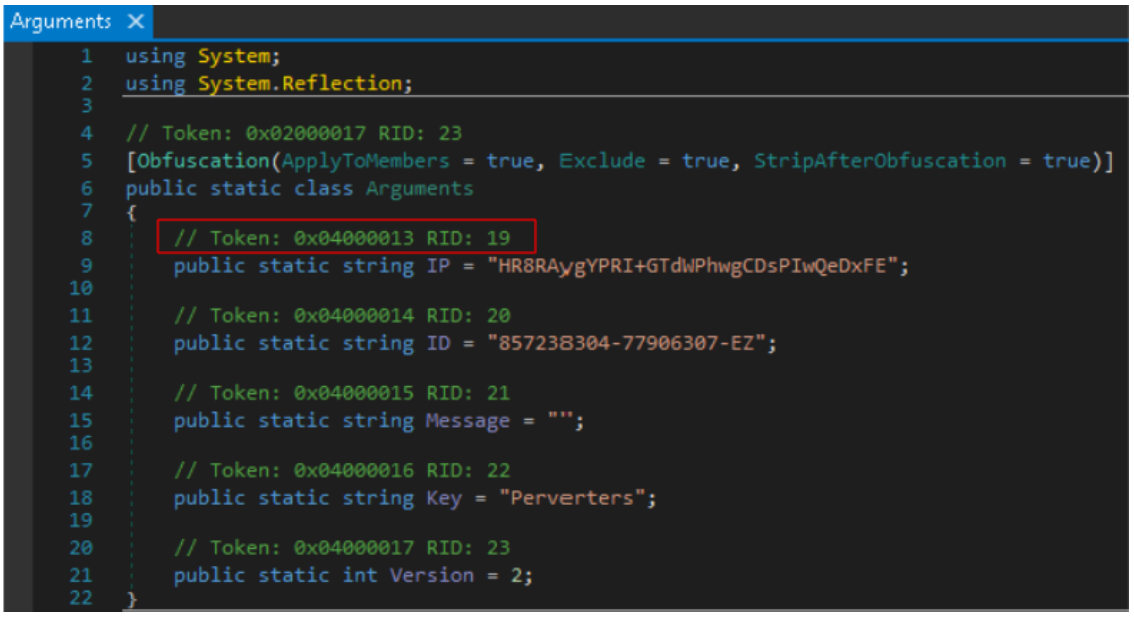

Figure 7. DnSpy labeling the token number of the decompiled strings.

Next, we open the RedLine Stealer sample in IDA Pro and navigate to the same function. Figure 8 shows that the ldstr commands push object reference for the metadata strings located at seg000:29F1, seg000:29FB, seg000:2A05 and seg000:2A0F. The object references are enclosed in black boxes in Figure 8.

These metadata strings are set by instructions located at seg000:29F6, seg000:2A00, seg000:2A0A and seg000:2A14 respectively. The stsfld instructions replace the value of a static field with a value from the evaluation stack. The values on the evaluation stack for each field are enclosed in red boxes.

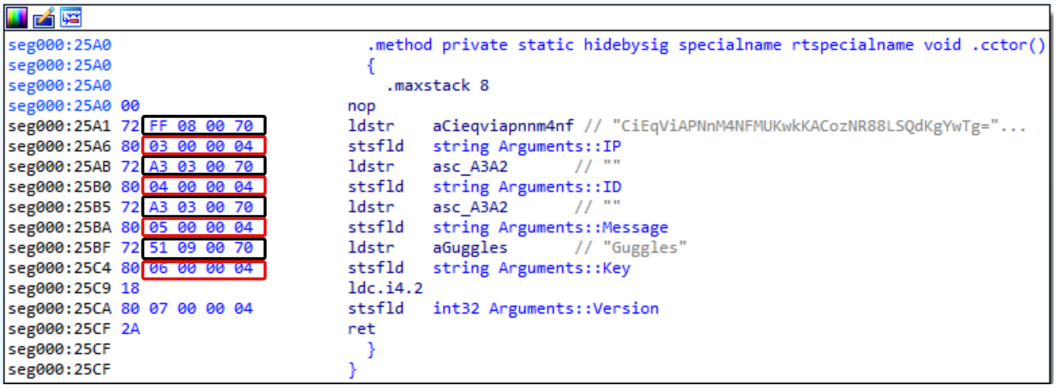

Figure 8. IDA Pro disassembler view of the configuration setup function.

The IP field from Figure 7 is not enough to statically extract the configuration. The source of the string that was pushed onto the stack for the IP field has not yet been identified. The operand type of the instruction ldstr shown in Figure 8 is, according to Microsoft, a string token, and string tokens are stored in the #US (User-Stream) table.

To find the string token, we used an open-source library called dnfile, which is like a .NET version of PEfile. Dnfile allows us to easily access the #US tokens by just giving the .NET runtime identifier (RID). Dnfile also provides the interface to access the user streams and a lot more.

The Python implementation shown in Figure 9 is an example of how we accessed user streams by offset. We passed the user string into the decryption routine shown in Figure 9 once we got the user stream by the token. This should return the decrypted configuration.

```
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 9. An implementation that uses dnfile to get the resource by a given .NET MSIL token.

Conclusion

By delving into the methods used for GuLoader and RedLine Stealer, we shed light on the process of locating and extracting C2 configurations from various malware families.

Leveraging our insights gained from analyzing these malware configurations, we can enhance our ability to detect, analyze and develop effective countermeasures against malicious software. Through continuous collaboration and knowledge sharing, we can collectively stay ahead of cybercriminals to help safeguard our digital systems and networks.

Palo Alto Networks customers are better protected from the threats discussed in this article through the following products:

- Next-Generation Firewall with cloud-delivered security services including WildFire detect the files mentioned within this report as malicious.

If you think you might have been compromised or have an urgent matter, get in touch with the Unit 42 Incident Response team or call:

- North America Toll-Free: 866.486.4842 (866.4.UNIT42)

- EMEA: +31.20.299.3130
- APAC: +65.6983.8730
- Japan: +81.50.1790.0200

Palo Alto Networks has shared these findings with our fellow Cyber Threat Alliance (CTA) members. CTA members use this intelligence to rapidly deploy protections to their customers and to systematically disrupt malicious cyber actors. Learn more about the Cyber Threat Alliance.

Indicators of Compromise

SHA256 Hash of the GuLoader Sample Analyzed in This Article

- 32ea41ff050f09d0b92967588a131e0a170cb46baf7ee58d03277d09336f89d9

SHA256 Hash of the RedLine Stealer Sample Analyzed in This Article

- a4cf69f849e9ea0ab4eba1cdc1ef2a973591bc7bb55901fdbceb412fb1147ef9
