

Malware Transmutation! - Unveiling the Hidden Traces of BloodAlchemy

Published: 2024-05-23 · Archived: 2026-04-05 14:48:49 UTC

- [Introduction](#)
- [Malware group History](#)
- [Analysis of BloodAlchemy](#)
 - [Initial infection vector and infection flow](#)
 - [Analysis of malicious DLL](#)
 - [Analysis of shellcode](#)
 - [Analysis of payload \(BloodAlchemy\)](#)
 - [Structures](#)
 - [Functions](#)
 - [Creation of VFT associated with each communication protocol](#)
 - [Backdoor commands](#)
- [The code similarities with Deed RAT](#)
- [Summary](#)
- [Appendix](#)

This post is also available in: [日本語](#)

Introduction

This article examines the analysis of a malware called "BloodAlchemy" that we observed in an attack campaign. In October 2023, BloodAlchemy was named by Elastic Security Lab [1](#) as a new RAT (Remote Access Trojan). However, our investigation has revealed that BloodAlchemy is not an entirely new malware but an evolved version of Deed RAT, the successor to ShadowPad.

Malware group History

Let's look at ShadowPad first. ShadowPad is a particularly notorious malware family used in APT (Advanced Persistent Threat) campaigns. It was first reported in a software supply chain attack in July 2017. At that time, ShadowPad was embedded in one of the code libraries of a server management software for enterprise networks provided by NetSarang [2](#).

In the early stages of 2019, it was believed that only APT41 was using ShadowPad. However, since 2020, many security researchers reported that it may have been utilized by various APT groups [3](#).

Moving on to Deed RAT, it is believed to have been used as a RAT by the threat group called Space Pirates, active since at least 2017, based on its implementation. Additionally, Positive Technologies' security team suggests that Deed RAT shows a high degree of code similarity with ShadowPad [4](#).

Now, let's delve into BloodAlchemy, the malware in question. According to Elastic Security Lab's analysis, this malware exhibits several characteristics, such as using legitimate binaries to load malicious DLLs, multiple run modes, persistence mechanisms, and importing specific functions of various communication protocols when communicating with its command and control (C2) server. These traits indicate that BloodAlchemy is a new variant of Deed RAT that is still being actively developed by attackers.

The public information of ShadowPad, Deed RAT, and BloodAlchemy is as follows:

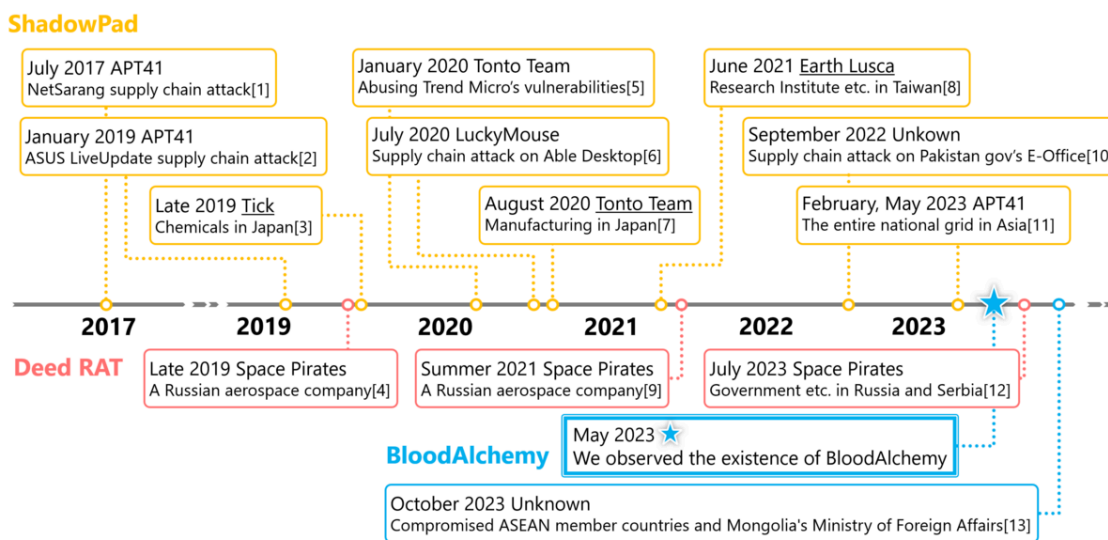


Figure 1. Public information on ShadowPad, Deed RAT, and BloodAlchemy

References of Figure 1

- [1] [ShadowPad in corporate networks](#)
- [2] [Operation ShadowHammer: a high-profile supply chain attack](#)
- [3] [Cyber Espionage Tradecraft in the Real World Adversaries targeting Japan in the second half of 2019](#)
- [4] [Space Pirates: analyzing the tools and connections of a new hacker group](#)
- [5] [ShadowPad: the Masterpiece of Privately Sold Malware in Chinese Espionage](#)
- [6] [Operation StealthyTrident: corporate software under attack](#)
- [7] [APT Threat Landscape in Japan 2020](#)
- [8] [RedHotel: A Prolific, Chinese State-Sponsored Group Operating at a Global Scale](#)
- [9] [Chinese State-Sponsored Activity Group TAG-22 Targets Nepal, the Philippines, and Taiwan Using Winnti and Other Tooling](#)
- [10] [Attacks on industrial control systems using ShadowPad](#)
- [11] [Redfly: Espionage Actors Continue to Target Critical Infrastructure](#)

[12] [Space Pirates: a look into the group's unconventional techniques, new attack vectors, and tools](#)

[13] [Introducing the ref5961 intrusion set](#)

Analysis of BloodAlchemy

Initial infection vector and infection flow

In this case, we analyzed that the attacker used a file set to infect targets with BloodAlchemy by taking over a vender-use-only maintenance account on a VPN device. Figure 2 shows the infection flow.

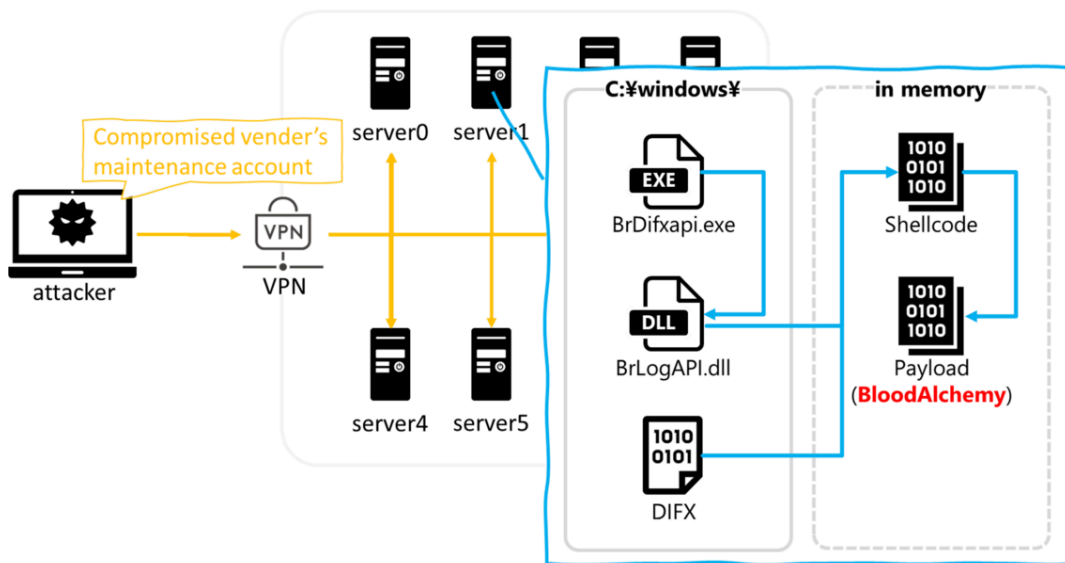


Figure 2. The infection flow of BloodAlchemy

The malicious file set consisted of three files: `BrDifxapi.exe` , `BrLogAPI.dll` , and `DIFX` . These files were stored under the directory `C:\windows\` . Additionally, a scheduled task (`C:\Windows\System32\Tasks\De11\BrDifxapi`) was created for persistence.

Created0x10	Create...	File Size	Parent Path	File Name
=	=	=	c:	c:
2023-05-06 08:42:00		3220	.\Windows\System32\Tasks\De11	BrDifxapi
2023-05-06 08:42:00		0	.\Windows\System32\Tasks	De11
2023-05-06 08:41:44		66112	.\Windows	DIFX
2023-05-06 08:41:41		129536	.\Windows	BrLogAPI.dll
2023-05-06 08:41:37		111472	.\Windows	BrDifxapi.exe

Figure 3. Discovered malicious file set.

Analysis of malicious DLL

When `BrDifxapi.exe` is executed on the infected host, it leverages the DLL side-loading technique to load a malicious DLL file called `BrLogAPI.dll` in the same directory. Subsequently, this malicious DLL loads the

DIFX file and decrypts shellcode from it, executing the shellcode in memory. The crypto algorithm is AES128 (CBC mode), and the key is the first 16 bytes of the DIFX file.

```

43     strcpy(String2, "DIFX");
44     lstrcatA(Filename, String2);
45     FileA = CreateFileA(Filename, 0x80000000, 1u, 0, 3u, 0, 0);
46     hFile = FileA;
47     v15 = FileA;
48     if ( FileA == -1
49         || (FileSize = GetFileSize(FileA, 0), nNumberOfBytesToRead = FileSize, FileSize == -1)
50         || (dec_shellcode = VirtualAlloc(0, FileSize, 0x3000u, 4u)) == 0 )
51     {
52         LastError = GetLastError();
53     }
54     else
55     {
56         NumberOfBytesRead = 0;
57         if ( ReadFile(hFile, dec_shellcode, nNumberOfBytesToRead, &NumberOfBytesRead, 0) )
58         {
59             enc_data = NumberOfBytesRead;
60             v17[72] = 0;
61             v19 = 0i64;
62             aes_init(v8, &v19);
63             enc_data -= 16;
64             aes_dec(dec_shellcode + 16, enc_data);
65             v10 = enc_data - dec_shellcode[enc_data - 1];
66             v17[0] = 0;
67             NumberOfBytesRead = v10;
68             ModuleHandleA = GetModuleHandleA("kernel32.dll");
69             VirtualProtect = GetProcAddress(ModuleHandleA, "VirtualProtect");
70             if ( VirtualProtect(dec_shellcode, 4096, 32, v17) )
71                 LastError = (dec_shellcode)(0);
72             else

```

Figure 4. The decryption process of shellcode in BrLogAPI.dll

Before

0000000000: 46 5C 45 00 7A 66 C4 DC	DD C9 27 A8 26 8B C6 26
0000000010: 61 36 8A FA FA D4 56 91	48 88 35 DD 07 82 43 6D
0000000020: 76 EC 43 12 6A 13 28 F8	4A 4F 63 F4 20 20 FA 4C
0000000030: 56 BA 85 63 27 95 85 23	8A F9 D5 61 DE F5 99 48
0000000040: B5 2B 4A CD 95 58 05 69	88 86 AC E2 C4 BB 7A D2
0000000050: 4F DB A2 30 FC EF F4 B3	62 83 5D E0 71 79 67 12
0000000060: D5 F6 B5 C1 94 29 B3 65	BA F3 B3 E7 76 29 E9 44

16bytes
key



AES128 CBC

After

0000000000: E8 00 00 00 00 58 8D 40	FB 8B 54 24 04 68 3A 04
0000000010: 00 00 68 74 F8 00 00 68	75 05 00 00 50 52 E8 03
0000000020: 00 00 00 C2 04 00 55 8B	EC 83 EC 34 33 C0 53 8B
0000000030: D8 89 45 E4 89 45 E0 89	45 DC 89 45 D8 89 45 E8
0000000040: 89 45 D4 64 A1 30 00 00	00 56 57 89 5D FC 8B 40
0000000050: 0C 8B 78 14 E9 F3 01 00	00 8B 47 28 33 DB 89 45
0000000060: F4 8B CB 8B F0 8A 00 C1	C9 07 0F B6 D0 3C 61 72

Figure 5. The DIFX data (before) and the decrypted shellcode (after)

Analysis of shellcode

The decrypted shellcode contains an encrypted and compressed form of BloodAlchemy. This custom decryption process based on the FNV-1a hash algorithm and the lznt1 compression.

```

seg000:02620275      mov     [ebp+buf], ecx ; 00AE0000
seg000:02620278      movzx  ebx, word ptr [edx] ; [0x2620575] = 0x5511
seg000:0262027B      add     edx, eax ; 0x2620577
seg000:0262027D      lea    eax, [esi-2] ; 0xF872
seg000:02620280      mov     [ebp+key?], ebx ; 0x5511
seg000:02620283      mov     [ebp+dst_imagebase], eax ; 0xF872
seg000:02620286      test   eax, eax
seg000:02620288      jz     short loc_26202E8
seg000:0262028A      sub     edx, ecx ; 01B40577
seg000:0262028C      mov     [ebp+size], eax ; size = 0xF872
seg000:0262028F      mov     esi, ecx ; 00AE0000
seg000:02620291      mov     [ebp+var_34], edx ; 0x1B40577
seg000:02620294
seg000:02620294      FNV1a?: ; CODE XREF: main_loader+288!j
seg000:02620294      push   2
seg000:02620296      mov     edi, 2166136261 ; offset_basis
seg000:02620298      xor     ecx, ecx
seg000:0262029D      pop    edx
seg000:0262029E      loc_262029E: ; CODE XREF: main_loader+288!j
seg000:0262029E      movzx  eax, byte ptr [ebp+ecx+key?] ; 0x5511
seg000:026202A3      xor     eax, edi
seg000:026202A5      imul  edi, eax, 16777619 ; FNV_prime
seg000:026202AB      inc    ecx
seg000:026202AC      cmp    ecx, edx
seg000:026202AE      jb     short loc_262029E
seg000:026202B0      mov     edx, [ebp+var_34] ; 0x1B40577
seg000:026202B3      imul  ecx, edi, 2001h
seg000:026202B9      mov     eax, ecx
seg000:026202BB      shr     eax, 7
seg000:026202BE      xor     eax, ecx
seg000:026202C0      imul  eax, 9
seg000:026202C3      mov     ecx, eax
seg000:026202C5      shr     ecx, 11h
seg000:026202C8      xor     ecx, eax
seg000:026202CA      imul  eax, ecx, 21h ; '!'
seg000:026202CD      xor     ebx, eax
seg000:026202CF      mov     al, [edx+esi] ; r0: [0x2620577] = 0x86
seg000:026202CF      ; r1: [0x2620578] = 0xC0
seg000:026202D2      xor     al, bl
seg000:026202D4      mov     [ebp+key?], ebx ; 0x5511
seg000:026202D7      mov     [esi], al ; r0: [0xAE0000] = 0xD4
seg000:026202D7      ; r1: [0xAE0001] = 0xBA
seg000:026202D9      inc    esi
seg000:026202DA      sub     [ebp+size], 1 ; size -= 1
seg000:026202DE      jnz   short FNV1a? ; ret. 0xEA0000 =
seg000:026202DE      ; 00AE0000 D4 BA 00 45 AB 45 AB 10 00 00 00 24 8C 69 00 40 0s;E#E#...$.i.@
seg000:026202DE      ; 00AE0010 00 40 01 40 00 00 00 70 01 00 AB 6F 01 00 BC B0 .@.@...p...mo..W
seg000:026202DE      ; 00AE0020 63 01 00 00 01 70 00 54 50 00 8C 44 AD 0F 01 3C C....p.TP...D...<
seg000:026202DE      ; 00AE0030 01 00 FD 00 1C E0 02 39 01 50 01 00 DD 49 01 00 ..y..b.g.P..Vt..
seg000:026202E0      mov     eax, [ebp+dst_imagebase] ; 0xF872
    
```

Figure 6. The custom crypto method using FNV-1a hash.

What is FNV-1a hash algorithm?

Fowler-Noll-Vo (FNV) is a hash algorithm based on an idea originally submitted as reviewer comments to the IEEE POSIX P1003.2 committee by Glenn Fowler and Phong Vo in 1991. It was later improved by Noll. > FNV is an abbreviation that combines the names of its creators. FNV is widely used for various purposes, including DNS servers, X (formerly Twitter) services, database index hashing, major web search/index engines, Message-ID search functionality in netnews history files, and > spam filtering, among others.

[The FNV Non-Cryptographic Hash Algorithm](#)

```

seg000:026202E8      mov     _VirtualAlloc, [ebp+VirtualAlloc]
seg000:026202EB      mov     [ebp+EP_payload], eax
seg000:026202EE      imul   eax, 5
seg000:026202F1      push   4
seg000:026202F3      push   edi
seg000:026202F4      add    eax, 1000h
seg000:026202F9      push   eax
seg000:026202FA      push   0
seg000:026202FC      mov     [ebp+var_C], eax
seg000:026202FF      call   _VirtualAlloc
seg000:02620301      mov     ebx, eax
seg000:02620303      lea    eax, [ebp+var_C]
seg000:02620306      push   eax
seg000:02620307      push   [ebp+EP_payload] ; 0xF872
seg000:0262030A      mov     [ebp+key?], ebx ; 0x5511
seg000:0262030D      push   [ebp+buf] ; 00AE0000 D4 BA 00 45 AB 45 AB 10 00 00 00 24 8C 69 00 40
seg000:02620310      ; 00AE0010 00 40 01 40 00 00 00 70 01 00 AB 6F 01 00 BC 80
seg000:02620313      ; 00AE0020 63 01 00 00 01 70 00 54 50 00 0C 44 AD 0F 01 3C
seg000:02620316      ; 00AE0030 01 00 FD 00 1C E0 02 39 01 50 01 00 DD 49 01 00
seg000:02620319      push   [ebp+var_C] ; 0x4EA3A
seg000:0262031B      push   ebx ; src_imagebase = 0x2780000
seg000:0262031D      push   2
seg000:0262031F      pop    eax
seg000:02620321      push   eax
seg000:02620323      call   [ebp+RtlDecompressBuffer] ; ret.
seg000:02620326      ; 02780000 45 AB 45 AB 10 00 00 00 8C 69 00 00 00 00 40 00
seg000:02620329      ; 02780010 00 00 00 00 00 70 01 00 AB 6F 01 00 BC 63 01 00
seg000:02620332      ; 02780020 00 10 00 00 00 00 00 00 50 00 00 00 AD 0F 01 00
seg000:02620335      ; 02780030 00 10 01 00 FD 0F 01 00 E0 39 00 00 00 50 01 00
seg000:02620338      push   4
seg000:0262033A      push   edi ; 0x3000
seg000:0262033C      push   dword ptr [ebx+14h] ; 0x17000
    
```

Figure 7. Decompression process using the lznt1.

What is LZNT1 compression algorithm

The compression algorithm that can be easily used by calling the Windows API named RtlDecompressBuffer.

[LZNT1 Algorithm Details | Microsoft Learn](#)

It has been discovered that the restored BloodAlchemy payload has a unique data format that closely resembles the PE format but is different. Below are the data structures of the custom format.

offset	Descriptions	Data
0x00	magic number	45 AB 45 AB
0x04	plugin id	0x10
0x08	entry point	0x698c
0x0c	original base	0x400000
0x10	absolute offset	0
0x14	size of virtualalloc	0x17000
0x18	size of raw data	0x16fab
0x1c	size of unknown	0x163bc
0x20	base of code?	0x1000

offset	Descriptions	Data
0x24	section1: virtual address	0x0
0x28	section1: raw data address	0x50
0x2c	section1: size of raw data	0x10fa0
0x30	section2: virtual address	0x11000
0x34	etc..	

Once the BloodAlchemy payload is restored, the previous mentioned shellcode interprets this custom format for deploying the final payload into memory and executes it as the fireless malware (Figure 8).

```

call    [ebp+RtlDecompressBuffer] ; ret.
        ; 02780000 45 AB 45 AB 10 00 00 00 8C 69 00 00 00 00 40 00
        ; 02780010 00 00 00 00 00 70 01 00 AB 6F 01 00 BC 63 01 00
        ; 02780020 00 10 00 00 00 00 00 50 00 00 00 AD 0F 01 00
        ; 02780030 00 10 01 00 FD 0F 01 00 E0 39 00 00 00 50 01 00
push    4
push    edi                ; 0x3000
push    dword ptr [ebx+14h] ; 0x17000
push    0
call    _VirtualAlloc
mov     esi, [ebx+28h]     ; raw_address1 = 0x50
mov     edi, [ebx+24h]     ; virtual_address1 = 0
add     esi, ebx          ; src_imagebaase + raw_address1 = 0x02780050
mov     ecx, [ebx+2Ch]    ; virtual_size1 = 0x10FAD
add     edi, eax          ; dst_imagebase + virtual_address1 = 0x2640000
rep movsb
mov     esi, [ebx+34h]    ; raw_address2 = 0x10FFD
mov     edi, [ebx+30h]    ; virtual_address2 = 0x11000
add     esi, ebx          ; src_imagebaase + raw_address2 = 0x2790FFD
mov     ecx, [ebx+38h]    ; virtual_size2 = 0x39E0
add     edi, eax          ; dst_imagebase + virtual_address2 = 0x2651000
rep movsb
mov     esi, [ebx+40h]    ; raw_address3 = 0x149DD
mov     edi, [ebx+3Ch]    ; virtual_address3 = 0x15000
add     esi, ebx          ; src_imagebaase + raw_address3 = 0x27949DD
mov     ecx, [ebx+44h]    ; virtual_size3 = 0x180A
add     edi, eax          ; dst_imagebase + virtual_address3 = 0x2655000
rep movsb
mov     esi, [ebx+1Ch]    ; 0x163BC
mov     [ebp+dst_imagebase], eax ; dst_imagebase = 0x2640000
    
```

Figure 8. The code that interprets the custom format to deploy the BloodAlchemy.

Analysis of payload (BloodAlchemy)

Structures

BloodAlchemy has several features that are not commonly found in other malware. One of these features is the 'run mode' value. When transferring the processing from the shellcode mentioned earlier to the entry point of the payload, it is called with six specified arguments.

The first argument set the value of run mode, and the BloodAlchemy's behavior varies significantly based on this value. The following table summarizes the values for each run mode and their corresponding behaviors:

run mode	Behavior corresponding to each run mode
0	Communication with C2 and backdoor functionality, creation of specified process for code injection, code injection into specified processes, anti-debugging, anti-sandbox techniques, Persistence
1	Communication with C2 and backdoor functionality
2	Creation of thread for Communication with C2 and backdoor functionality
3	Communication with C2 and backdoor functionality, code injection into specified processes, anti-debugging, anti-sandbox techniques, Persistence
4	Creation of specified process for code injection
5	Creation of named pipes
6	Installation of malware

It has been confirmed that BloodAlchemy exhibits the ability to load a malware configuration. This configuration is embedded in an encrypted state within the previous shellcode and, it is decrypted and utilized during BloodAlchemy's execution (Figure 9).

Furthermore, if a file with a 15-character filename consisting of [a-zA-Z] exists within the directory `C:\ProgramData\Store`, it will be loaded as the malware configuration. The same decryption algorithm used in the previously mentioned payload was utilized for this decryption process.

```

11 p_ntdll_RtlInitializeCriticalSection(&dword_2652644);
12 dec = memcpy_c_dec_FNV1a(v2); // From the previous loader shellcode ->
13 //
14 // key =
15 // 0262FDE9 86 58
16 //
17 // enc_data =
18 // 0262FDEB 94 3E 17 18 3C 31 34 1C 6F DA ED 31 84 E2 16 12
19 // 0262FDFB CF D3 7D A1 BC 10 E9 79 EE AA 31 C6 EC 38 C7 32
20 // 0262FE08 09 00 AA D6 33 77 F6 9A 16 A9 50 89 D6 24 3F 52
21 // 0262FE1B 23 DE 90 91 EE 6E 34 68 D4 A5 12 02 68 5B F6 4F
22 //
23 // ret. dec =
24 // 006FE008 37 B4 00 34 05 00 00 09 AD 67 BC 00 9A 36 B0 05
25 // 006FE018 01 00 00 00 4E 58 01 20 00 00 04 30 00 76 00 20
26 // 006FE028 90 55 00 0C A9 00 0C C3 00 0C D1 04 60 D7 15 00
27 // 006FE038 1C DD 00 0C F5 00 06 0D 02 00 D4 00 13 00 06 42
28 v0 = c_RtlDecompressBuffer(1500, v2, dec, &dec); // ret. decompressed =
29 // 02652068 34 05 00 00 09 AD 67 BC 9A 36 B0 05 01 00 00 00
30 // 02652078 58 01 00 00 00 00 00 01 00 00 00 00 00 00
31 // 02652088 76 01 00 00 90 01 00 00 A9 01 00 00 C3 01 00 00
32 // 02652098 D1 01 00 00 00 00 00 00 D7 01 00 00 DD 01 00 00
33 if ( v0 < 0 )
34 c_RtlNtStatusToDosError(v0);

```

Figure 9. The decryption and loading code of the malware configuration.

The malware configuration contains important data related to malicious code processing. This data includes values to manipulate the behavior set in the run mode, the URL of the C2 server, process names specified for code

injection, and more. Some important data such as a MUTEX value, C2 server, target process name etc., are primarily encrypted. Additionally, it also includes offset values indicating the positions of these encrypted data like ShadowPad.

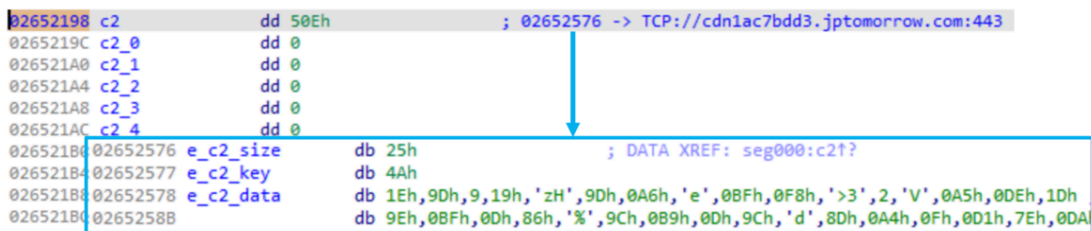


Figure 10. The encrypted data and the offset values indicating their positions in the configuration.

Each of these encrypted data is stored in the following order: the size of the encrypted data, a byte key, and the encrypted data itself.

offset	descriptions	data
0x00	size of data	0x25
0x01	a byte key	0x41
0x02	encrypted data	1E 9D 09 19 7A D0 9D 9D ...

The decryption is performed using another custom algorithm with the stored key. We created a simple Python script to decrypt the encrypted data.

```

import struct

def dec_cmt(offset):
    s = struct.unpack("B", ida_bytes.get_bytes(offset, 1))[0]
    data = ida_bytes.get_bytes(offset, s + 2)
    iv = data[1]
    enc = data[2:]
    dec = ""
    for i in range(s):
        dec += chr(iv ^ enc[i] & 0xFF)
        ku0 = iv << (i % 5 + 1) & 0xFF
        ku1 = iv >> (7 - i % 5) & 0xFF
        iv = (iv + (ku0 | ku1)) & 0xFF
    return dec[:-1]
    
```

As an example, the resolved offsets and decrypted data for each value in malware configuration using the Python script is as follows:

```

:02652068 offset_config dd 534h ; DATA XREF: load_dec_config_and_check_file+3
:02652068 ; get_value_from_config_by_arg0+14to ...
:02652068 ; conf_size
:0265206C unkown dd 08C67AD09h
:02652070 unkown_0 dd 5B0369Ah
:02652074 createmutex_flag dd 1 ; 0: off
:02652074 ; 1: on
:02652078 mutex_value dd 158h ; 0x26521c0 -> DFYNBEDKJHGAFSTIJECYUKFDEUJH
:0265207C selefdelete_flag dd 0
:02652080 antidbg_flag dd 1
:02652084 checksandbox_flag dd 0
:02652088 install_reg dd 176h ; 0x26521de -> SOFTWARE\Microsoft\Store
:0265208C install_dir dd 190h ; 0x26521f8 -> %ALLUSERSPROFILE%\Store
:02652090 leg_exe dd 1A9h ; 0x2652211 -> %AUTOPATH%\Test\test.exe
:02652094 mal_dll dd 1C3h ; 0x265222b -> BrLogAPI.dll
:02652098 blob dd 1D1h ; 0x2652239 -> DIFX
:0265209C persistence_flag dd 0 ; 0: off
:0265209C ; 1: service + startup + taskschd
:0265209C ; 2: service
:0265209C ; 3: startup
:0265209C ; 4: taskschd
:026520A0 servicename dd 1D7h ; 0x265223f -> Test
:026520A4 str_service dd 1DDh ; 0x2652245 -> Digital Imaging System
:026520A8 str_service_0 dd 1F5h ; 0x265225d -> Digital Imaging System
:026520AC reg_name dd 20Dh ; 0x2652275 -> Test
:026520B0 reg_key dd 213h ; 0x265227b -> L"SOFTWARE\Microsoft\Windows\Cu
:026520B4 str_task dd 242h ; 0x26522aa -> ONLOGON
:026520B8 taskname dd 248h ; 0x26522b3 -> Test
:026520BC procinjection_flag dd 0
:026520C0 p_injectionproc dd 251h ; 0x26522b9 -> %windir%\system32\SearchIndexer
:026520C4 p_injectionproc_0 dd 276h ; 0x26522de -> %windir%\system32\wininit.exe
:026520C8 p_injectionproc_1 dd 295h ; 0x26522fd -> %windir%\system32\taskhost.exe
:026520CC p_injectionproc_2 dd 2B5h ; 0x265231d -> %windir%\system32\svchost.exe
:026520D0 prochollowing_flag dd 0 ; 0: off
:026520D0 ; 1: on
:026520D4 p_processhollowing dd 2D4h ; 0x265233c -> %windir%\system32\wininit.exe
:026520D8 p_processhollowing_0 dd 2F3h ; 0x265235b -> %windir%\system32\taskeng.exe
:026520DC p_processhollowing_1 dd 312h ; 0x265237a -> %windir%\system32\taskhost.exe
:026520E0 p_processhollowing_2 dd 332h ; 0x265239a -> %windir%\system32\svchost.exe

```

Figure 11. Example of resolving offsets and decrypted data.

Not only malware configuration, but the same encryption is also used for other embedded data, such as important data related to some specific file paths. This Python script can also decrypt these data as well.

```

02651000 e__ProgramFiles__db 60h, 45h, 70h, 0D2h, 0CAh, 98h, 8Ch, 9Ah, 87h, 7, 3Ch
02651000 ; DATA XREF: check_arch_for_path_test_exe+1
0265100B db 93h, 98h, 8Ah, 0EDh, 0 ; %ProgramFiles%
02651010 e__LocalAppData__Programs db 94h, 0B1h, 0F1h, 0DCh, 33h, 34h, 93h, 0BFh, 89h, 0B8h
02651010 ; DATA XREF: check_arch_for_path_test_exe+F
0265101A db 10h, 0BFh, 0EFh, 68h, 74h, 3Ah, 62h, 0E4h, 9Fh, 10h ; %LocalAppD
02651024 db 9Ch, 0AAh, 0Fh, 98h
02651028 e__c_program_files__x86__db 29h ; ) ; DATA XREF: check_arch_for_path_test_exe+1
02651028 ; L"C:\Program Files (x86)"
02651029 db 0Ch
0265102A dw 1A2Bh
0265102C dd 526302C4h, 0C684B892h, 2CDD8D9Ah, 7420ADE6h
0265103C e__AppData__ db 9Bh, 0BEh, 93h, 6Dh, 75h, 11h, 9Eh, 8Ah, 98h, 0EDh
0265103C ; DATA XREF: check_arch_for_path_test_exe+1
02651046 db 2 dup(0) ; %AppData%
02651048 e__AUTOPATH__ db 7Fh ; ; DATA XREF: check_arch_for_path_test_exe+3
02651049 db 5Ah, 3Ch, 27h
0265104C dd 0BFAF1A51h, 7180ADh

```

Figure 12. Example of decrypting data other than the malware configuration.

As mentioned above, BloodAlchemy behaves differently depending on the run mode and the values in the malware configuration. From this characteristic, we believe the BloodAlchemy is a rather unique sample. The main function of BloodAlchemy is communication with a C2 server and controlling the infected host through the implemented backdoor commands.

The individual functionalities implemented in BloodAlchemy are introduced here.

Persistence

The payload incorporates a persistence capability. If the run_mode is 0 or 3 and the execution file path is not for persistence, and if the persistence_flag (a value of 0x34 in the malware configuration) is not 0, the persistence method will be chosen based on the value of the persistence_flag from 1 to 4.

- 1: service + startup + taskschd (COM obj)
- 2: service
- 3: startup
- 4: taskschd (COM obj)

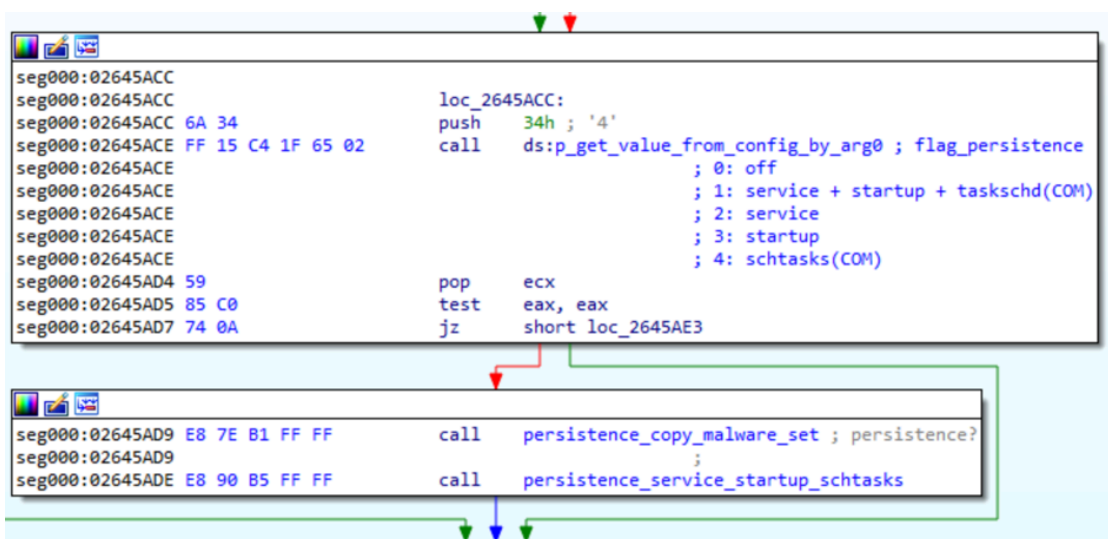


Figure 13. The calling a function of persistence depending on the persistence_flag.

The persistence mechanism is designed for the malware set consisting of test.exe, BrLogAPI.dll, and DIFX to be created within one of the corresponding directories based on the infected environment.

- %AUTOPATH%\Test\
- %LocalAppData%\Programs\Test\
- %ProgramFiles%\Test\
- %ProgramFiles(x86)%\Test\

Anti Sandbox

The payload also has anti-sandbox capabilities to evade analysis in sandbox environments. This feature only functions when the run_mode is 0, the executable file path is not for persistence, and the value of 0x1c in the

configuration is 1. It checks the process_name, files, and DNS results. It is speculated that the purpose of this feature is to avoid detection from Trellix sandbox functionality, based on the checked process names.



Figure 14. The anti-sandbox capabilities are enabled by the value of configuration.

Process Injection

The process injection feature was implemented with following conditions which were the run_mode is 0 or 3 and the value of 0x54 in the configuration is 1, it attempts to inject the previous shellcode into the following processes specified in the configuration from 0x58 to 0x64.

- %windir%\system32\SearchIndexer.exe
- %windir%\system32\wininit.exe
- %windir%\system32\taskhost.exe
- %windir%\system32\svchost.exe

In order to set the injected shellcode as a queue for asynchronous procedure calls (APC), the QueueUserAPC() function is used. This technique is known as Early Bird Injection.

What is Asynchronous Procedure Call (APC)

A function that is executed asynchronously in the context of a specific thread. Each thread has its own APC queue, and an application can register an APC in the queue by calling the QueueUserAPC() function. This > will result in the execution of the APC function and the occurrence of a software interrupt during the next scheduled thread."

[Asynchronous Procedure Calls | Microsoft Learn](#)

```

9 | v8 = size_of_prv_shellcode + size_of_prv_shellcode_key_enc_payload + 1084;
10 | buf = p_kernel32_VirtualAllocEx(a1, 0, v8, 12288, 4);
11 | if ( !buf )
12 |     return p_kernel32_GetLastError();
13 | if ( !p_kernel32_WriteProcessMemory(a1, buf, 39976960, v8, &v8) )
14 |     goto LABEL_9;
15 | v5 = size_of_prv_shellcode;
16 | if ( size_of_prv_shellcode <= 0x1000 )
17 |     v5 = 4096;
18 | if ( !p_kernel32_VirtualProtectEx(a1, buf, v5, 32, v7)
19 |     || !p_kernel32_QueueUserAPC(buf, a2, a3)
20 |     || p_kernel32_ResumeThread(a2) == -1 )
21 | {

```

Figure 15. The process injection using QueueUserAPC() function.

As related feature of the payload, if the run_mode is 0 or 4 and the value of 0x68 in the configuration is 1, it creates the following processes specified from 0x6c to 0x74 and attempts to inject the shellcode into those processes using QueueUserAPC() too.

- %windir%\system32\wininit.exe
- %windir%\system32\taskeng.exe
- %windir%\system32\taskhost.exe
- %windir%\system32\svchost.exe

Creation of VFT associated with each communication protocol

The BloodAlchemy was designed for up to 10 C2 destinations. However, interestingly, in the samples we observed, only one C2 information was in there. Based on the C2 information, the communication protocol is extracted, and the Protocol ID to be used within the malware is set. Based on this Protocol ID, the functions necessary for communication are imported, and a Virtual Function Tables (VFT) is created..

What is Virtual Function Tables (VFT)

A table that stores pointers to virtual functions within a class. If a class has one or more virtual functions, the compiler creates a virtual function table for that class. Each instance of the class holds pointers to this > table.

Virtual Function Tables | Microsoft Learn

```

17  switch ( protocol_id )
18  {
19      case 1:
20          if ( p_get_value_from_config_by_arg0(0x98) )// 1
21          {
22              v4 = v10;
23              return TCP_MUX_protocol(v4);
24          }
25  LABEL_7:
26      v4 = 0;
27      return TCP_MUX_protocol(v4);
28  case 8:
29      goto LABEL_7;
30  case 2:
31      v3 = 0;
32      return HTTP_HTTPS_protocol(v3);
33  case 3:
34      return HTTP_HTTPS_protocol(v3);
35  case 4:
36      if ( p_get_value_from_config_by
37          v6 = v10;
38      else
39          v6 = 0;
40      return UDP_protocol(v6);
41  case 5:
42      if ( p_get_value_from_config_by_arg0(0x98) )// 1
43          v9 = v10;
44      else
45          v9 = 0;
46      p_get_value_from_config_by_arg0(0xB0); // 0x16
47      p_get_value_from_config_by_arg0(0xAC); // ONLOGON
48      return DNS_protocol(v9);
49  case 7:
50      v7 = 0x20000;
51      return PIPE_SMB_protocol(v7);
52  case 6:
53      v7 = 0;
54      return PIPE_SMB_protocol(v7);
41  initd->field_0 = 1;
42  result = initd;
43  initd->create_socket_connect = tcp_create_socket_connect;
44  initd->connect_send_recv = tcp_connect_send_recv;
45  initd->recv_data = tcp_recv_data;
46  initd->shutdown = tcp_shutdown;
47  initd->shutdown_closesocket = tcp_shutdown_closesocket;
48  initd->wsarecv = tcp_wsarecv;
49  initd->wsagetoverlappedresult = tcp_wsagetoverlappedresult;
50  initd->wsasend = tcp_wsasend;
51  initd->wsagetoverlappedresult_ = tcp_wsagetoverlappedresult;
52  initd->getsockname = tcp_getsockname;
53  initd->getsockname_ = tcp_getsockname;
54  return result;
    
```

Figure 16. A VFT is created in the malware to handle the corresponding protocol based on the Protocol ID.

Backdoor commands 

15 backdoor commands were implemented to control victim machine. The operations performed by each command ID are as follows:

command id	descriptions
0x1101	update config
0x1102	get current config
0x1201	update test.exe
0x1202	update BrLogAPI.dll
0x1203	update DIFX
0x1204	uninstall and terminated
0x1205	launch persistence_dir\test.exe
0x1301	unknown

command id	descriptions
0x1302	load received payload and store it into registry value
0x1303	delete registry value
0x1304	unknown
0x1401	get proxy info
0x1402	update proxy info
0x1501	gather victim info
0x1502	echo 0x1502

```

21  command_id = *(a2 + 12);
22  if ( command_id <= 0x1301 )
23  {
24      if ( command_id == 0x1301 )
25          return bc_1301(buf, a2);
26      command_1101_1 = command_id - 0x1101;
27      if ( !command_1101_1 )
28          return bc_1101_update_config_(buf, a2);
29      command_1102_1 = command_1101_1 - 1;
30      if ( !command_1102_1 )
31      {
32          p_ntdll_RtlEnterCriticalSection(&dword_2652644);
33          v10 = bc_1102get_current_config_(buf, 4354, &offset_config, offset_config);
34          p_ntdll_RtlLeaveCriticalSection(&dword_2652644);
35          return v10;
36      }
37      v5 = command_1102_1 - 0xFF;
38      if ( !v5 )
39          return bc_1201_update_test_exe(buf);
40      v6 = v5 - 1;
41      if ( !v6 )
42          return bc_1202_update_BrLogAPI_dll(buf);
43      v7 = v6 - 1;
44      if ( !v7 )
45          return bc_1203_update_DIFX(buf);
46      v8 = v7 - 1;
47      if ( !v8 )
48          bc_1204_remove_persistence(buf);
49      if ( v8 == 1 )
50          return bc_1205_exec_test_exe(buf);
51      return 50;
52  }
53  command_1302_1 = command_id - 0x1302;
54  if ( !command_1302_1 )
55      return bc_1302_load_payload_create_regvalue(buf);

```

Figure 17. Branching of processing based on the backdoor command ID

The code similarities with Deed RAT

Based on our reversing results, we have discovered multiple similarities between BloodAlchemy and Deed RAT. Here are some examples of code similarities that we consider particularly significant:

The first remarkably similar point is the unique data structures of the payload header in both BloodAlchemy and Deed RAT. Although there are differences in values such as magic number and plugin ID and other values. This data structure is designed based on the PE header which maps the payload into memory based on its respective values.

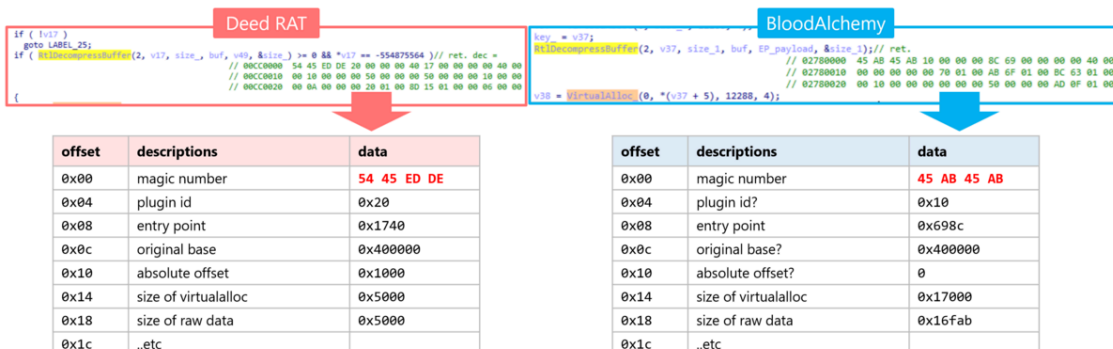


Figure 18. Comparison of custom data structures between Deed RAT and BloodAlchemy

In relation to above example, some similarities have been found in the loading process of shellcode, and the DLL file used to read the shellcode as well. Regarding the payload, various similarities have been confirmed with high confidence:

- Exception handling after the entry point
- Loading start functions for each plugin
- Plugin names
- Plugin information
- Structure of the malware configuration (offset of encrypted data)
- Hardcoded directories and a specific file name used for persistence

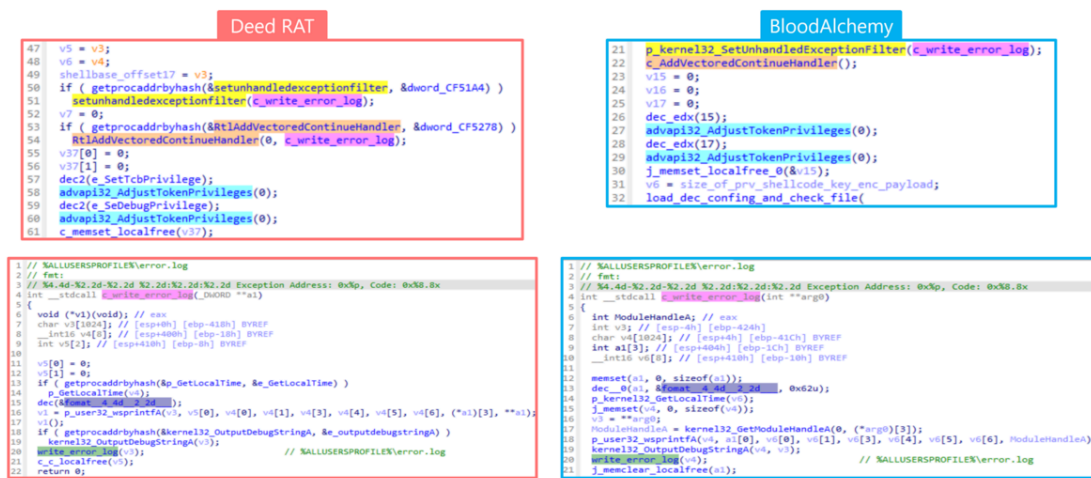


Figure 19. Comparison of exception handling after the entry point.

We have concluded that BloodAlchemy is highly likely to be a variant of Deed RAT, based on our deeply analysis and comparison results.

Summary

In this article, we have explained the analysis results of BloodAlchemy. The origin of BloodAlchemy and Deed RAT is ShadowPad and given the history of ShadowPad being utilized in numerous APT campaigns, it is crucial to pay special attention to the usage trend of this malware.

One more thing, our experts presented a talk titled "Into the Vapor to Tracking Down Unknown Panda's Claw Marks" at the Botconf 2024 held in Nice, France, discussing the analysis of BloodAlchemy.

The slide of presentation is available here, if you interested in the BloodAlchemy research, please check it.

- [\[Slide\] Into the Vapor to Tracking Down Unknown Panda's Claw Marks](#)

Appendix

Source: <https://blog-en.itochuci.co.jp/entry/2024/05/23/090000>