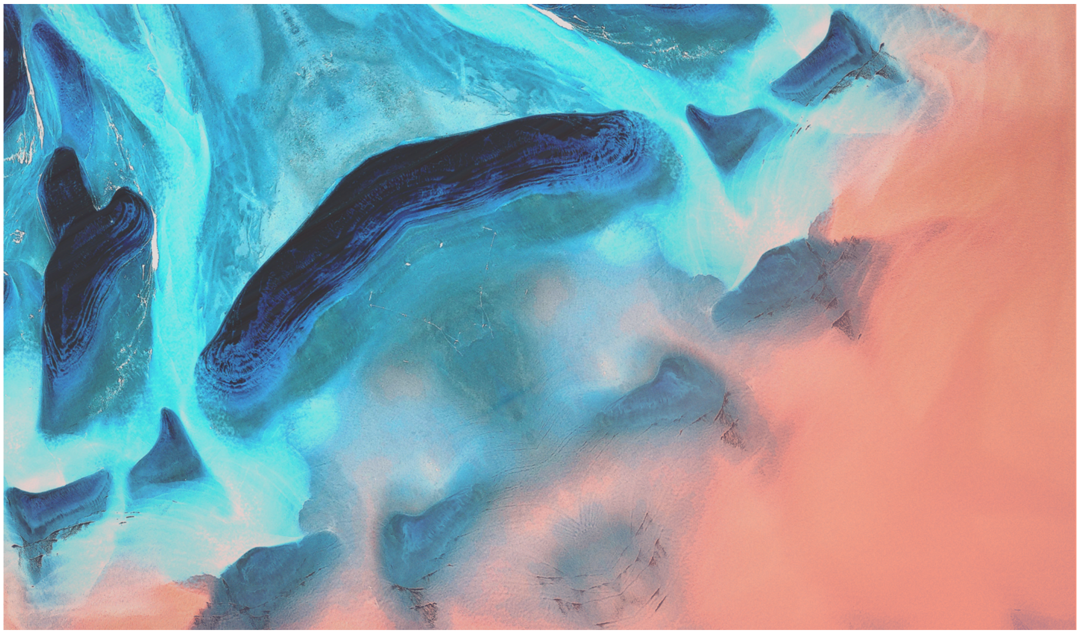
Disclosing the BLOODALCHEMY backdoor



13 October 2023•Cyril François

BLOODALCHEMY is a new, actively developed, backdoor that leverages a benign binary as an injection vehicle, and is a part of the REF5961 intrusion set.

15 min read Security research, Malware analysis

Disclosing the BLOODALCHEMY backdoor

Preamble

BLOODALCHEMY is an x86 backdoor written in C and found as shellcode injected into a signed benign process. It was discovered in our analysis and is part of the REF5961 intrusion set, which you can read about here.

BLOODALCHEMY requires a specific loader to be run because it isn't reflexive (it doesn’t have the capability to load and execute by itself). Additionally, BLOODALCHEMY isn’t compiled as position independent (when loaded at a different base address than the preferred one the binary has to be patched to take into account the new “position”).

In our analysis, the signed benign process was previously sideloaded with a malicious DLL. The DLL was missing from the sample data but was likely the container and the loader of the BLOODALCHEMY shellcode.

We believe from our research that the malware is part of a bigger toolset and is still in active development based on its current lack of capabilities, enabled debug logging of exceptions, and the existence of test strings used for persistence service setup.

Key takeaways

- BLOODALCHEMY is likely a new backdoor and is still in active development
- BLOODALCHEMY abuses a legitimate binary for loading
- BLOODALCHEMY has multiple running modes, persistence mechanisms, and communication options

Initial execution

During the initial execution phase, the adversary deployed a benign utility, `BrDifxapi.exe`, which is vulnerable to DLL side-loading. When deploying this vulnerable utility the adversary could side-load the unsigned BLOODALCHEMY loader (`BrLogAPI.dll`) and inject shellcode into the current process.

```
"command_line": "\"C:\\Windows\\System32\\cmd.EXE\" /c \"cd C:\\Windows && start C:\\Windows\\BrDifxapi.exe\"",
```

Command-line used to execute the BLOODALCHEMY loader

```
"path": "C:\\Windows\\BrLogAPI.dll",
"code_signature": {
  "exists": false
},
"name": "BrLogAPI.dll",
```

Fake BrLogApi.dll, part of BLOODALCHEMY toolset, sideloaded by BrDifxapi.exe

`BrDifxapi.exe` is a binary developed by the Japanese company Brother Industries and the version we observed has a revoked signature.

Signature Verification

⚠ Signed file, valid signature. Revoked.

File Version Information

Copyright	Copyright(C) 2009-2013 Brother Industries, Ltd.
Product	BrDifxapi.exe
Description	BrDifxapi
Original Name	BrDifxapi.exe
Internal Name	BrDifxapi
File Version	1, 1, 1, 0
Date signed	2015-02-03 08:42:00 UTC

BrDifxapi.exe with revoked signature

The legitimate DLL named `BrLogApi.dll` is an unsigned DLL also by Brother Industries. BLOODALCHEMY uses the same DLL name.

Signature Verification

⚠ File is not signed

File Version Information

Copyright	Copyright (C) 2004-2008 Brother Industries, Ltd.
Product	Brother MFC Windows Software Standard Debug Log Send DLL
Description	Brother MFC Windows Software Standard Debug Log Send DLL
Original Name	BrLogAPI.dll
Internal Name	Brother MFC Windows Software Standard Debug Log Send DLL

The legitimate BrLogApi.dll is an unsigned DLL file

Code analysis

Data Obfuscation

To hide its strings the BLOODALCHEMY malware uses a classic technique where each string is encrypted, preceded by a single-byte decryption key, and finally, all concatenated together to form what we call an encrypted blob.

While the strings are not null-terminated, the offset from the beginning of the blob, the string, and the size are passed as a parameter to the decryption function. Here is the encrypted blob format:

Blob = Key0 :EncryptedString0 + Key1:EncryptedString1 + ... + KeyN:EncryptedStringN

The implementation in Python of the string decryption algorithm is given below:

```
def decrypt_bytes(encrypted_data: bytes, offset: int, size: int) -> bytes:
    decrypted_size = size - 1
    decrypted_data = bytearray(decrypted_size)

    encrypted_data_ = encrypted_data[offset : offset + size]
    key = encrypted_data_[0]

    i = 0
    while i != decrypted_size:
            decrypted_data[i] = key ^ encrypted_data_[i + 1]
           key = (key + ((key << ((i % 5) + 1)) | (key >> (7 - (i % 5))))) & 0xFF
           i += 1

    return bytes(decrypted_data)
```

The strings contained in the configuration blob are encrypted using the same scheme, however the ids (or offsets) of each string are obfuscated; it adds two additional layers of obfuscation that must be resolved. Below, we can resolve additional obfuscation layers to decrypt strings from the configuration:

```
def decrypt_configuration_string(id: int) -> bytes:
        return decrypt_bytes(
                *get_configuration_encrypted_string(
                        get_configuration_dword(id)))
```

Each function is given below:

The `get_configuration_dword` function

```
def get_configuration_dword(id: int) -> int:
        b = ida_bytes.get_bytes(CONFIGURATION_VA + id, 4)
        return b[0] + (b[1] + (b[2] + (b[3] << 8) << 8) << 8)
```

The `get_configuration_encrypted_strng` function

```
def get_configuration_encrypted_string(id: int) -> tuple[int, int]:
         ea = CONFIGURATION_VA + id

        v2 = 0
        i = 0

        while i <= 63:
            c = ida_bytes.get_byte(ea)

            v6 = (c & 127) << i
            v2 = (v2 | v6) & 0xFFFFFFFF

            ea += 1

            if c >= 0:
                break
```

```
            i += 7
            return ea, v2
```

Persistence

BLOODALCHEMY maintains persistence by copying itself into its persistence folder with the path suffix `\Test\test.exe`,

```
LODWORD(v6) = 40;
if ( !ctf::configuration::GeWStrBuffer1(v6, v7) )// %AUTOPATH%\\Test\\test.exe
{
```

BLOODALCHEMY folder and binary name

The root directory of the persistence folder is chosen based on its current privilege level, it can be either:

- `%ProgramFiles%`
- `%ProgramFiles(x86)%`
- `%Appdata%`
- `%LocalAppData%\Programs`

```
    g_fp_GetNativeSystemInfo(&SystemInfo);
    if ( SystemInfo.wProcessorArchitecture == 9
      || SystemInfo.wProcessorArchitecture == 6
      || SystemInfo.wProcessorArchitecture == 7 )
    {
      v4 = (uint8_t *)&g_encrypted_string_blob_1[9];// %ProgramFiles(x86)%
    }
    else
    {
      v4 = (uint8_t *)&unk_731000;          // %ProgramFiles%
    }
    ctf::WStrBuffer::DecryptExpandVariableString(p_file_path, v4, v5);
  }
  else
  {
    v8 = 284;
    v9 = 5;
    sub_72DC72(&v8);
    if ( v9 <= 5 )
      ctf::WStrBuffer::DecryptExpandVariableString(p_file_path, (uint8_t *)&g_encrypted_string_blob_1[14], 0xAu);// %AppData%
    else
      ctf::WStrBuffer::DecryptExpandVariableString(p_file_path, (uint8_t *)&g_encrypted_string_blob_1[3], 0x18u);// %LocalAppData%\\Programs
  }
```

BLOODALCHEMY root persistence folder choice

Persistence is achieved via different methods depending on the configuration:

- As a service
- As a registry key
- As a scheduled task
- Using COM interfaces

To identify the persistence mechanisms, we can use the uninstall command to observe the different ways that the malware removes persistence.

As a service named `Test`.

```
g_fp_4(56, (ctf::WStrBuffer *)&v3);            // Test
ctf::DeleteServiceIfStopped(*(wchar_t **)&v3);
```

BLOODALCHEMY deleting previously installed service

As a registry key at `CurrentVersion\Run`

```
g_fp_4(72, &p_output);                   // SOFTWARE\\Microsoft\\Windows\\CurrentVersion\\Run
g_fp_4(68, (ctf::WStrBuffer *)&v3);      // Test
ctf::DeleteRegistryValue(HKEY_LOCAL_MACHINE, p_output.p_w_data, *(_DWORD *)&v3);
ctf::DeleteRegistryValue(HKEY_CURRENT_USER, p_output.p_w_data, *(_DWORD *)&v3);
```

BLOODALCHEMY deleting "CurrentVersion\Run" persistence registry key

As a scheduled task, running with SYSTEM privilege via `schtask.exe`:

```
b'schtasks.exe /CREATE /SC %s /TN "%s" /TR "\'%s\'" /RU "NT AUTHORITY\\SYSTEM" /Fb'
```

Using the `TaskScheduler::ITaskService` COM interface. The intent of this persistence mechanism is currently unknown.

```
  _result = g_fp_CoCreateInstance(g_clsid_TaskScheduler, 0, (void *)7, g_iid_ITaskService, (void **)&p_itask_service);
  if ( _result < 0 )
  {
LABEL_8:
    v5 = _result;
```

Instantiation of the ITaskService COM interface

Running modes

The malware has different running modes depending on its configuration:

- Within the main or separate process thread
- Create a Windows process and inject a shellcode into it
- As a service

The malware can either work within the main process thread.

```
    }
    p_this = 0;
    ctf::thread::DoTheActualJob();
LABEL_17:
```

Capability function called within the main function

Or run in a separate thread.

```
Thread = g_fp_CreateThread(0, 0, ctf::thread::DoTheActualJob, 0, 0, 0);
if ( Thread )
  g_fp_CloseHandle(Thread);
```

Capability function called in a new thread

Or create a Windows process from a hardcoded list and inject a shellcode passed by parameter to the entry point using the WriteProcessMemory+QueueUserAPC+ResumeThread method.

```
  if ( g_fp_ConfigurationGetDword(84) )
    ctf::CreateProcessInjectShellcodeIfConfigValue84ThenTerminateIself();
```

Process injection running method

```
g_fp_ConfigurationGetWstrBuffer0(index, &s);// b'%windir%\\system32\\SearchIndexer.exe'
                                            // b'%windir%\\system32\\wininit.exe'
                                            // b'%windir%\\system32\\taskhost.exe'
                                            // b'%windir%\\system32\\svchost.exe'
```

List of target binaries for process injection

The shellcode is contained in the parameters we call `p_interesting_data`. This parameter is actually a pointer to a structure containing both the malware configuration and executable binary data.

```
int __stdcall ctf::Main(
        uint32_t p_file_path,
        uint32_t running_mode,
        uint8_t *p_interesting_data,
        uint32_t shellcode_size,
        uint32_t size_1,
        size_t arg_0)
{
```

Entrypoint prototype

```
int __stdcall ctf::QueueUserAPCProcessInjection(HANDLE h_process, HANDLE h_thread, ULONG_PTR a3)
{
  void *remote_address; // edi
  int _error; // esi
  SIZE_T v5; // eax
  DWORD flOldProtect; // [esp+8h] [ebp-8h] BYREF
  SIZE_T n_bytes; // [esp+Ch] [ebp-4h] BYREF

  n_bytes = g_shellcode_size + g_size_1 + g_size_0;

  remote_address = g_fp_VirtualAllocEx(h_process, 0, n_bytes, 0x3000u, PAGE_READWRITE);
  if ( !remote_address )
    return g_fp_GetLastError();

  if ( !g_fp_WriteProcessMemory(h_process, remote_address, g_p_interesting_data, n_bytes, &n_bytes) )
    goto LABEL_9;
```

Provided shellcode copied in the remote process

```
if ( g_fp_VirtualProtectEx(h_process, remote_address, v5, PAGE_EXECUTE_READ, &flOldProtect)
  && g_fp_QueueUserAPC((PAPCFUNC)remote_address, h_thread, a3)
  && g_fp_ResumeThread(h_thread) != -1 )
{
```

Final part of the process injection procedure

Or install and run itself as a service. In this scenario, the service name and description will be `Test` and `Digital Imaging System`:

```
g_fp_4(56, &p_output);                    // Test
g_fp_4(60, (ctf::WStrBuffer *)&v6);       // Digital Imaging System
g_fp_4(64, (ctf::WStrBuffer *)&v3);       // Digital Imaging System
```

Name and description strings used to install the BLOODALCHEMY service

Also when running as a service and started by the service manager the malware will masquerade itself as stopped by first setting the service status to “SERVICE_RUNNING” then setting the status to “SERVICE_STOPPED” while in fact the malware is still running.

```
g_h_service_status_handle = g_fp_RegisterServiceCtrlHandlerW(*(LPCWSTR *)service_name, ctf::callback::ServiceHandler);
if ( g_h_service_status_handle )
{
  ctf::SetServiceStatus(4u);                    // 4 = SERVICE_RUNNING
  g_fp_Sleep(0x3E8u);
  ctf::SetServiceStatus(1u);                    // 1 = SERVICE_STOPPED
}
```

BLOODALCHEMY’s service entry point masquerading service status

Communication

The malware communicates using either the HTTP protocol, named pipes, or sockets.

When using the HTTP protocol the malware requests the following URI `/Inform/logger/`.

```
ctf::DecryptCpyString(uri, 0x100073157Cui64, (uint8_t *)0xC8, v18);// /Inform/logger
```

URI used to connect to C2

In this scenario, BLOODALCHEMY will try to use any proxy server found in the registry key `SOFTWARE\\Microsoft\\Windows\\CurrentVersion\\Internet Settings`.

```
ctf::crypto::DecryptCStr(&key_path, &g_encrypted_string_blob_0[224], v9);// SOFTWARE\\Microsoft\\Windows\\CurrentVersion\\Internet Settings
if ( g_fp_RegOpenKeyExA(h_registry, (LPCSTR)key_path.p_data, 0, 0x20019u, (PHKEY)&key_path.capacity + 1) )
  goto LABEL_2;
LODWORD(v8) = 12;

v15 = 4;
h_proxy_enable_value = 0;
ctf::crypto::DecryptCStr(&key_path, &g_encrypted_string_blob_0[284], v8);// ProxyEnable
if ( g_fp_RegQueryValueExA(HIDWORD(key_path.capacity), key_path.p_data, 0, 0, &h_proxy_enable_value) )
{
LABEL_2:
  LastError = g_fp_GetLastError();
}
else if ( h_proxy_enable_value )
{
  j_g_fp_memset(h_proxy_server_value, 0, 1024);
  LODWORD(v6) = 12;
  v15 = 1024;
  ctf::crypto::DecryptCStr(&key_path, &g_encrypted_string_blob_0[96], v6);// ProxyServer
  if ( !g_fp_RegQueryValueExA(HIDWORD(key_path.capacity), key_path.p_data, 0, 0, h_proxy_server_value) )
    ctf::CStrBuffer::FromCStr(_p_proxy_server, h_proxy_server_value);

  j_g_fp_memset(h_proxy_ovveride_value, 0, 1024);
  LODWORD(v7) = 14;
  v15 = 1024;
  ctf::crypto::DecryptCStr(&key_path, &g_encrypted_string_blob_0[24], v7);// ProxyOverride
  if ( !g_fp_RegQueryValueExA(HIDWORD(key_path.capacity), key_path.p_data, 0, 0, h_proxy_ovveride_value) )
    ctf::CStrBuffer::FromCStr(p_proxy_override, h_proxy_ovveride_value);
```

Host proxy information gathered from registry

We did not uncover any C2 infrastructure with our sample, but the URL could look something like this: `https://malwa[.]re/Inform/logger`

When using a named pipe, the name is randomly generated using the current PID as seed.

```
memset(w_random_pipe_name, 0, sizeof(w_random_pipe_name));
g_fp_GenerateRandomWStr(current_pid, w_random_pipe_name, 17u);
```

Random pipe name generation seeded with current PID

While waiting for a client to connect to this named pipe the malware scans the running processes and checks that its parent process is still running, this may be to limit access to the named pipe. That said, the malware is not checking that the pipe client is the correct parent process, only that the parent process is running. This introduces flawed logic in protecting the named pipe.

```
ctf::GetParentPid(current_pid, &parent_pid);
```

Retrieve parent PID

```
for ( i = g_fp_EnumProcesses(p_pid_list, 0x2000u, &n_bytes); i; i = g_fp_EnumProcesses(p_pid_list, 0x2000u, &n_bytes) )
{
  j = 0;
  n_pids = n_bytes >> 2;
  n_bytes = n_pids;
  if ( n_pids )
  {
    do
    {
      // Seems that parent process must exist else it won't accept client. Also it's not testing that the client is indeed the parent process ...
      if ( (PVOID)p_pid_list[j] == parent_pid )
        break;
      ++j;
    }
    while ( j < n_pids );
  }

  if ( j >= n_pids )
    goto end;
```

Flawed check for restricting pipe access to parent process

From the malware strings and imports we know that the malware can also operate using TCP/UDP sockets.

```
g_fp_connect         00735310
g_fp_gethostbyname   00735350
g_fp_getpeername     00735308
g_fp_getsockname     0073532C
g_fp_inet_ntoa       00735334
g_fp_ioctlsocket     00735340
g_fp_recv   00735318
g_fp_recvfrom        00735338
g_fp_select 00735304
g_fp_send   0073530C
g_fp_sendto 00735300
```

Usage of the socket API in one of the implementations of the "communication" interface

While we haven't made any conclusions about their usage, we list all the protocols found in the encrypted strings.

- DNS://
- HTTP://
- HTTPS://
- MUX://
- UDP://
- SMB://
- SOCKS5://
- SOCKS4://
- TCP://

For all protocols the data can be encrypted, LZNT1 compressed, and/or Base64-encoded.

Commands

The malware only contains a few commands with actual effects:

- Write/overwrite the malware toolset
- Launch its malware binary `Test.exe`
- Uninstall and terminate
- Gather host information

There are three commands that write (or overwrite) the malware tool set with the received Base64-encoded binary data:

- Either the malware binary (`Test.exe`)
- the sideloaded DLL (`BrLogAPI.dll`)
- or the main trusted binary (`BrDifxapi.exe`)

```
if ( !v5 )
  return ctf::command::OverwriteMalwareBinaryInPersistenceFolder((int)a1, p_struc_16->field_0);
v6 = v5 - 1;
if ( !v6 )
  return ctf::command::WriteBrLogAPIDLLFileInPersistenceFolder((int)a1, p_struc_16->field_0);
v7 = v6 - 1;
if ( !v7 )
  return ctf::command::WriteDIFXFileInPersistenceFolder((int)a1, p_struc_16->field_0);
```

BLOODALCHEMY tool set overwrite commands

One command that launches the `Test.exe` binary in the persistence folder.

```
    return ctf::command::LaunchMalwareInPersistenceFolder(a1);
  return 50;
```

BLOODALCHEMY command to run the malware executable binary

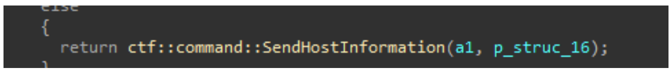
The uninstall and terminate itself command will first delete all its files at specific locations then remove any persistence registry key or scheduled task, then remove installed service and finish by terminating itself.

```
if ( !v8 )
  ctf::command::UninstallAndTerminateItself(a1);
```

Command to uninstall and terminate itself

Uninstall function

One host information gathering command: CPU, OS, display, network, etc.



Information gathering command

Summary

BLOODALCHEMY is a backdoor shellcode containing only original code(no statically linked libraries). This code appears to be crafted by experienced malware developers.

The backdoor contains modular capabilities based on its configuration. These capabilities include multiple persistence, C2, and execution mechanisms.

While unconfirmed, the presence of so few effective commands indicates that the malware may be a subfeature of a larger intrusion set or malware package, still in development, or an extremely focused piece of malware for a specific tactical usage.

BLOODALCHEMY and MITRE ATT&CK

Elastic uses the MITRE ATT&CK framework to document common tactics, techniques, and procedures that advanced persistent threats used against enterprise networks.

Tactics

Tactics represent the why of a technique or sub-technique. It is the adversary's tactical goal: the reason for performing an action.

Malware prevention capabilities

- BLOODALCHEMY

YARA

Elastic Security has created YARA rules to identify this activity. Below are YARA rules to identify the BLOODALCHEMY malware:

```
BLOODALCHEMY
rule Windows_Trojan_BloodAlchemy_1 {
    meta:
        author = "Elastic Security"
        creation_date = "2023-05-09"
        last_modified = "2023-06-13"
        threat_name = "Windows.Trojan.BloodAlchemy"
        license = "Elastic License v2"
        os = "windows"

    strings:
```

```
        $a1 = { 55 8B EC 51 83 65 FC 00 53 56 57 BF 00 20 00 00 57 6A 40 FF 15 }
        $a2 = { 55 8B EC 81 EC 80 00 00 00 53 56 57 33 FF 8D 45 80 6A 64 57 50 89 7D
E4 89 7D EC 89 7D F0 89 7D }

    condition:
        all of them
}

rule Windows_Trojan_BloodAlchemy_2 {
    meta:
        author = "Elastic Security"
        creation_date = "2023-05-09"
        last_modified = "2023-06-13"
        threat_name = "Windows.Trojan.BloodAlchemy"
        license = "Elastic License v2"
        os = "windows"

    strings:
        $a1 = { 55 8B EC 83 EC 54 53 8B 5D 08 56 57 33 FF 89 55 F4 89 4D F0 BE 00 00
00 02 89 7D F8 89 7D FC 85 DB }
        $a2 = { 55 8B EC 83 EC 0C 56 57 33 C0 8D 7D F4 AB 8D 4D F4 AB AB E8 42 10 00
00 8B 7D F4 33 F6 85 FF 74 03 8B 77 08 }

    condition:
        any of them
}

rule Windows_Trojan_BloodAlchemy_3 {
    meta:
        author = "Elastic Security"
        creation_date = "2023-05-10"
        last_modified = "2023-06-13"
        threat_name = "Windows.Trojan.BloodAlchemy"
        license = "Elastic License v2"
        os = "windows"

    strings:
        $a = { 55 8B EC 83 EC 38 53 56 57 8B 75 08 8D 7D F0 33 C0 33 DB AB 89 5D C8
89 5D D0 89 5D D4 AB 89 5D }

    condition:
        all of them
}

rule Windows_Trojan_BloodAlchemy_4 {
    meta:
        author = "Elastic Security"
        creation_date = "2023-05-10"
        last_modified = "2023-06-13"
        threat_name = "Windows.Trojan.BloodAlchemy"
        license = "Elastic License v2"
        os = "windows"

    strings:
        $a = { 55 8B EC 83 EC 30 53 56 57 33 C0 8D 7D F0 AB 33 DB 68 02 80 00 00 6A
40 89 5D FC AB AB FF 15 28 }

    condition:
        all of them
}
```

Observations

All observables are also available for download in both ECS and STIX format in a combined zip bundle.

The following observables were discussed in this research.

Observable	Type	Name	Reference
e14ee3e2ce0010110c409f119d56f6151fdca64e20d902412db46406ed89009a	SHA-256	BrLogAPI.dll	BLOODALCHEMY loader
25268bc07b64d0d1df441eb6f4b40dc44a6af568be0657533088d3bfd2a05455	SHA-256	NA	BLOODALCHEMY payload