

From Russia(?) with Code

lastline.com/labsblog/attribution-from-russia-with-code/

March 9, 2018



Posted by [Labs Team](#) ON MAR 9, 2018

The Olympic Destroyer cyberattack is a very recent and notable attack by sophisticated threat actors against a globally renowned 2-week sporting event that takes place once every four years in a different part of the world. Successfully attacking the Winter Olympics requires motivation, planning, resources and time.

Cyberattack campaigns are often a reflection of real world tensions and provide insight into the possible suspects in the attack. Much has been written about the perpetrators behind Olympic Destroyer emanating from either North Korea or Russia. Both have motivations. North Korea would like to embarrass its sibling South Korea, the holders of the 23rd Winter Olympics. Russia could be seeking revenge for the IOC ban on their team. And Russia has precedence, having previously been blamed for attacks on other sporting organizations, such as the intrusion at the World Anti Doping Agency that was targeted via a stolen International Olympic Committee account.

There has been much said about attribution, with accusations of misleading false flags and anti-forensics built into the malware. As Talos points out in their report, attribution is hard.

But attribution is not just hard, it's often a wilderness of mirrors and, more often than not, a bit anticlimactic.

The motivation of our following analysis is not to point the finger of blame about who did the attacking, but to utilize our expertise in analyzing malware code and understanding the behaviors it exhibits to highlight the heritage, evolution and commonalities we found in the code of the Olympic Destroyer malware.

Initial Samples of Code Reuse

Besides analyzing the behavior of a sample, our sandbox performs several levels of code analysis, eventually extracting all code components, regardless if they are run at run-time or not. As we described in a [blog post](#) a few years ago, this technique is essential if we are to detect any dormant functionality that might be present within the sample.

After decomposing the code components in normalized basic blocks, the sandbox computes smart code hashes that are stored and indexed in our threat intelligence knowledge base. Over the last 3 years we have been collecting code hashes for millions of files, so when we want to hunt for other samples related to the same actor, we are able to query our backend for any other binaries that have been reusing significant amounts of code.

The rationale being that actors usually build up their code base over time, and reuse it over and over again across different campaigns. Code surely might evolve, but some components are bound to remain the same. This is the intuition that drove our investigation on Olympic Destroyer further. The first results were obviously some variants of the Olympic Destroyer binaries which we have already mentioned in our [previous post](#). However, it quickly got way more interesting.

A very specific code hash led us through this process:

7CE26E95118044757D3C7A97CF9D240A (Lastline customers can use it to query our Global Threat Intelligence Network). This rare code hash surprisingly linked `21ca710ed3bc536bd5394f0bff6d6140809156cf`, a payload of the Olympic Destroyer campaign, with some other samples of a remote access trojan, "TVSpy." Though the actual internal name of the threat is TVRAT, the malware is known and labelled in VirusTotal as Trojan.Pavica or Trojan.Mezzo, none of which were previously connected to the original Olympic Destroyer campaign.

Figure 1 shows the actual code referenced by the code hash: it is a function used to read a buffer, and subsequently parse PE header from it.

```

signed int __usercall ParsePeHeader@<eax>(PE_HEADER *ctx@<ecx>, PARSING_RESULT *ctx_1)
{
    PE_HEADER *v2; // esi@1
    LONG v3; // ecx@3
    int v4; // eax@7
    SIZE_T v5; // eax@8
    LPVOID v6; // ebx@8
    int v8; // edx@10
    int v9; // ecx@10

    v2 = ctx;
    if ( ctx_1->flags >= 0x40u )
    {
        memcpy(ctx, ctx_1->size, 0x40u);
        if ( v2->dos_hdr.e_magic != 0x5A4D )
            return 2;
        v3 = v2->dos_hdr.e_lfanew;
        if ( !v3 )
            return 2;
        if ( (v3 + 0xF8) <= ctx_1->flags )
        {
            memcpy(&v2->hdr, (v3 + ctx_1->size), 0xF8u);
            if ( v2->hdr.Signature == 0x4550 && v2->hdr.OptionalHeader.Magic == 0x10B )
            {
                v4 = v2->hdr.FileHeader.NumberOfSections;
                if ( v4 )
                {
                    v5 = 40 * v4;
                    v2->size_section_headers = v5;
                    v6 = malloc(v5);
                    v2->sect = v6;
                    if ( !v6 )
                        return 3;
                    v8 = v2->hdr.FileHeader.SizeOfOptionalHeader + v2->dos_hdr.e_lfanew + 24;
                    v2->file_offset_section_headers = v8;
                    v9 = v2->size_section_headers;
                    if ( !v9 )
                        return 0;
                    if ( (v8 + v9) <= ctx_1->flags )
                    {
                        memcpy(v6, (v8 + ctx_1->size), v2->size_section_headers);
                        return 0;
                    }
                }
                return 1;
            }
        }
        return 2;
    }
    return 1;
}

```

Figure 1: The code referenced by the code hash 7CE26E95118044757D3C7A97CF9D240A shared by both the Olympic Destroyer sample 21ca710ed3bc536bd5394f0bff6d6140809156cf sha1 and TVSpy sample a61b8258e080857adc2d7da3bd78871f88edec2c.

This is not where code re-usage ends, as the actual function referencing and invoking the following fragment (see Figure 2) also shares almost all of the same logic. This function is responsible for loading PE file from the memory buffer and executing an entry point.

```

int __usercall LoadPeFromBuffer@<eax>(int a1@<edx>, int a2, int a3)
{
    int v3; // ebx@1
    int v4; // edi@1
    int result; // eax@2
    char v6; // [sp+1Ch] [bp-178h]@3
    int v7; // [sp+D4h] [bp-C0h]@10
    LPVOID v8; // [sp+154h] [bp-40h]@1
    int v9; // [sp+160h] [bp-34h]@10
    LPVOID lpAddress; // [sp+164h] [bp-30h]@10
    LPVOID lpMem; // [sp+168h] [bp-2Ch]@1
    int v12; // [sp+16Ch] [bp-28h]@1
    int v13; // [sp+170h] [bp-24h]@1
    int v14; // [sp+174h] [bp-20h]@1
    CPPEH_RECORD ms_exc; // [sp+17Ch] [bp-18h]@1

    v3 = a1;
    v8 = 0;
    lpMem = 0;
    v13 = 0;
    v12 = 0;
    v14 = 0;
    ms_exc.registration.TryLevel = 1;
    v4 = ParsePeHeader(a1);
    if ( v4
        || (v4 = sub_410CD0(&v6)) != 0
        || (ms_exc.registration.TryLevel = 2, (v4 = sub_410BB0(v3)) != 0)
        || (v4 = sub_410A70(&v6)) != 0
        || (v4 = sub_4108F0(&v6)) != 0
        || (v4 = sub_410830(&v6)) != 0
        || (v4 = sub_410790(&v6)) != 0 )
    {
        sub_403CC0(&__security_cookie, &ms_exc.registration, -2);
        result = v4;
    }
    else
    {
        if ( a3 )
        {
            *(_DWORD *)a3 = 32;
            *(_DWORD *) (a3 + 4) = 0;
            *(_DWORD *) (a3 + 8) = v9;
            *(_DWORD *) (a3 + 12) = lpAddress;
            *(_DWORD *) (a3 + 16) = v14;
            *(_DWORD *) (a3 + 20) = v7;
            *(_DWORD *) (a3 + 24) = lpMem;
            *(_DWORD *) (a3 + 28) = v12;
            ms_exc.registration.TryLevel = 2;
        }
        sub_403CC0(&__security_cookie, &ms_exc.registration, -2);
        result = 0;
    }
    return result;
}

```

Figure 2: Function responsible for loading PE file from memory reused in both Olympic Destroyer and TV Spy

A Deeper Dive Based on Unusual Code

We decided to further investigate this piece of code since loading PE from memory is not all that common. Its origin opened several questions:

1. Why is that piece of code the only link between the two samples?
2. Were there any other samples sharing the same code?

Our first discovery was a Remote Access trojan called TVSpy, mentioned above. This family has been the subject of a few previous research investigations, and a recent Benkow Lab [blog post](#) (from November 2017) even reported that the source code was available on github.

Unfortunately, all links to github are now dead. But that didn't stop us from finding the actual source code (or at least evidence that it was indeed published at some point). Apparently it was sold for \$US500 on an underground Russian forum in 2015. Even though the original post and links are gone, a Russian information security forum kept a copy of the source code package alongside a description of the original sale announcement (see Figure 3).

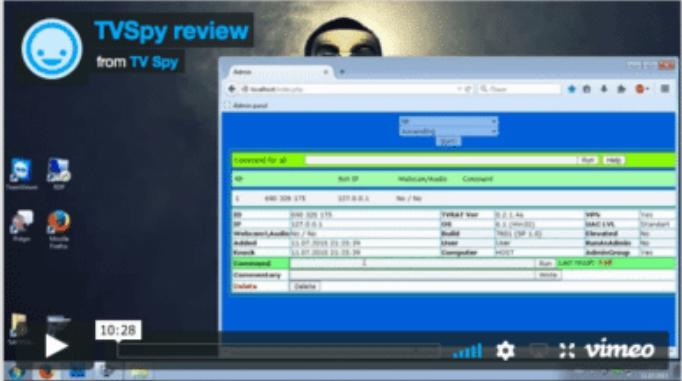
15.10.2015 #1

Описание:
 Исходный код скрытого TeamViewer бота с админ панелью. Бот представляет из себя dll весом 94 кб, которая управляет TeamViewer'ом. Билдится в Visual Studio 2010.
 Файл Microsoft Excel с расширением .xls с запросом о включении макроса вес 121 кб подгружает и устанавливает бота
 Скрипт сканирующий компьютер жертвы, флешки, сетевые шары, закладки и историю браузеров на ключевые слова и ссылки например "bank.ru, bitcoin, wallet.dat, webmoney" - результаты поиска высылает на почту в текстовом файле с путями к вкуснятине, вместе с ID бота в заголовке письма для удобства идентификации жертв.
 Кейлоггер на основе Punto Switcher

Возможности TeamViewer бота:

- + Админ панель с GeolP, с информацией о системе - микрофон, вебкамера, отправка команд ботам, возможность сортировки ботов, показ всех ботов - время онлайн, офлайн, время заражения
- + Загрузка и запуск dll из памяти TeamViewer, автозагрузка с ними
- + Шифрованная конфигурация, файлы бота скрытые и системные
- + Хорошие новости для новичка! Больше не нужно мучатся с хостингом, динамическим dns и тп. Бот отправляет свой ID на email и работает через сервера TeamViewer очень стабильно и быстро (обход NAT, BackConnect)
- + Скрытая загрузка бота после рестарта компьютера
- + Скрытая установка TeamViewer VPN (если права позволяют)
- + RDP через TeamViewer VPN. Работает без патчинга termsrv.dll - это очень стабильно можно не бояться обновлений Windows
- + Создает скрытый RDP аккаунт в системе
- + Создает возможность скрытой RDP сессии в одной учетке с жертвой
- + Скрытый TeamViewer бот не конфликтует с ранее установленной TeamViewer
- + Просмотр экрана, управление, файл менеджер, cmd, возможность запуска файлов прямо из файл менеджера (скрытый и обычный)
- + Отправка команд из TeamViewer чата
- + Стабильно работает на всех win32 / x64 в том числе Windows 10
- + Оригинальные файлы TeamViewer не изменены цифровые подписи не тронуты, что дает лояльность АВ и проактивов
- + Просмотр вебкамеры, прослушка микрофона
- + Бот работает даже в гостевой учетке, UAC молчит
- + многие другие фишки по мелочи

Видео обзор старой версии без GeolP в админке, скрипта и других плюшек



Продажник:чел продаёт за 500 баксов

Пароль на архив:111

Figure 3: TVSpy code as sold in an underground forum (according to researchers from ru-sfera.org)

Not Enough – The Investigation Continued

Although interesting, this connection was eventually not enough to connect Olympic Destroyer to Russia or to TVSpy. So we kept digging. Further research finally identified the code in Figures 1 and 2 to be part of an open source project called LoadDLL (see Figure 4) and available on codeproject.com (first published back in March 2014).

```

ELoadDLLResult LoadDLL(LOAD_DLL_READPROC read_proc, void* read_proc_param, int flags, LOAD_DLL_INFO* info)
{
    LOAD_DLL_CONTEXT ctx;
    ELoadDLLResult res;
    BOOL [finished successfully] = FALSE;
    unsigned i;

    if (!read_proc)
        return ELoadDLLResult_WrongFunctionParameters;

    ctx.sect = NULL;
    ctx.loaded_import_modules_array = NULL;
    ctx.import_modules_array_capacity = 0;
    ctx.num_import_modules = 0;
    ctx.dll_main = NULL;
    _try
    {
        _try
        {
            res = LoadDLL_LoadHeaders(&ctx, read_proc, read_proc_param);
            if (res != ELoadDLLResult_OK)
                return res;

            res = LoadDLL_AllocateMemory(&ctx, flags);
            if (res != ELoadDLLResult_OK)
                return res;

            _try
            {
                res = LoadDLL_LoadSections(&ctx, read_proc, read_proc_param, flags);
                if (res != ELoadDLLResult_OK)
                    return res;

                res = LoadDLL_PerformRelocation(&ctx);
                if (res != ELoadDLLResult_OK)
                    return res;

                res = LoadDLL_ResolveImports(&ctx);
                if (res != ELoadDLLResult_OK)
                    return res;

                res = LoadDLL_SetSectionMemoryProtection(&ctx);
                if (res != ELoadDLLResult_OK)
                    return res;
            }
        }
    }
}

```

Figure 4: Fragment of LoadDLL source code from LoadDLL project

However, a couple things still didn't add up: why had we only managed to identify samples from 2017 even if the source code was released in 2014? What about older versions of TVSpy? How come our search didn't return any of those samples? Were Olympic Destroyer and TVSpy samples from 2017 sharing more than just the LoadDLL code?

Apparently TVSpy went through a few transformations. Samples from 2015 did embed and use the LoadDLL code, but the compiler did some specific optimizations that made the code unique (see Figure 5). In particular the compiler optimized out both "flags" (not used in the function) and "read_proc" (statically link function) from the parameters of LoadDll, but it couldn't optimize out a "if (read_proc)" check even though it is useless since "read_proc" is not passed as a parameter anymore.


```

int __usercall LoadDLL@eax>(LOAD_DLL_FROM_MEMORY_STRUCT *read_proc_param, LOAD_DLL_INFO *info)
{
    int result; // eax@2
    int res; // esi@3
    LOAD_DLL_CONTEXT ctx; // [sp+Ch] [bp-17Ch]@3
    int v5; // [sp+160h] [bp-28h]@3
    int dll_main; // [sp+164h] [bp-24h]@3
    int finished_successfully; // [sp+16Ch] [bp-1Ch]@1
    CPPEH_RECORD ms_exc; // [sp+170h] [bp-18h]@3

    finished_successfully = 0;
    if ( read_proc )
    {
        ctx.sect = 0;
        ctx.loaded_import_modules_array = 0;
        v5 = 0;
        *(_DWORD *)&ctx.num_import_modules = 0;
        dll_main = 0;
        ms_exc.registration.TryLevel = 1;
        res = LoadDLL_LoadHeaders(&ctx, read_proc_param);
        if ( res
            || (res = LoadDLL_AllocateMemory(&ctx)) != 0
            || (ms_exc.registration.TryLevel = 2, (res = LoadDLL_LoadSections(&ctx)) != 0)
            || (res = LoadDLL_PerformRelocation(&ctx)) != 0
            || (res = LoadDLL_ResolveImports(&ctx)) != 0
            || (res = LoadDLL_SetSectionMemoryProtection(&ctx)) != 0
            || (res = LoadDLL_CallDLLEntryPoint(&ctx)) != 0 )
        {
            _local_unwind2((int)&ms_exc.registration, 0xFFFFFFFF);
            result = res;
        }
        else
        {
            if ( info )
            {
                info->size = 0xA0;
                info->flags = 0;
                info->image_base = ctx.image_base;
                info->mem_block = ctx.image;
                info->dll_main = dll_main;
                info->export_dir_rva = ctx.hdr.OptionalHeader.DataDirectory[0].VirtualAddress;
                info->loaded_import_modules_array = ctx.loaded_import_modules_array;
                info->num_import_modules = *(_DWORD *)&ctx.num_import_modules;
                ms_exc.registration.TryLevel = 2;
            }
            finished_successfully = 1;
            _local_unwind2((int)&ms_exc.registration, 0xFFFFFFFF);
            result = 0; // ELoadDLLResult_OK
        }
    }
    else
    {
        result = -2u; // ELoadDLLResult_WrongFunctionParameters
    }
    return result;
}

```

Figure 5. Reconstructed source code of LoadDll from TVSpy dated back to 2015

The “read_proc” function itself is also identical to one from source code (see Figures 6 and 7) and as you can see in Figure 8, it also gets called exactly the same way as the original source code from codeproject.com.


```

int __cdecl read_proc(void *buff, int data, size_t size, LOAD_DLL_FROM_MEMORY_STRUCT *a4)
{
    if ( size )
    {
        if ( data + size > a4->dll_size )
            return 0;
        memcpy(buff, (const void *) (data + a4->dll_data), size);
    }
    return 1;
}

```

Figure 6: read_proc function implementation

```

static BOOL LoadDLLFromMemoryCallback(
    void* ,
    size_t position,
    size_t size,
    LOAD_DLL_FROM_MEMORY_STRUCT* param)
{
    if (!size)
        return TRUE;
    if ((position + size) > param->dll_size)
        return FALSE;
    memcpy(buff, (char*)param->dll_data + position, size);
    return TRUE;
}

```

Figure 7: read_proc function implementation

The most interesting aspect for us is in fact the version of TVSpy that dates back to 2017-2018 and shares with Olympic Destroyer almost the exact binary code of LoadDLL. You can see LoadDll_LoadHeaders for those samples in Figure 9: as you might notice the function looks different then the one from the older version (see Figure 8).

```

signed int __usercall LoadDLL_LoadHeaders@<eax>(LOAD_DLL_CONTEXT *ctx@<ecx>, LOAD_DLL_FROM_MEMORY_STRUCT *info)
{
    LOAD_DLL_CONTEXT *ctx_1; // esi@1
    int v3; // eax@3
    signed int result; // eax@5
    int v5; // eax@8
    size_t size; // edi@9
    IMAGE_SECTION_HEADER *sect; // eax@9
    int file_offset_section_headers; // ecx@11

    ctx_1 = ctx;
    if ( read_proc(ctx, 0, 0x40u, info) )
    {
        if ( ctx_1->dos_hdr.e_magic != 0x5A4D )
            return 2;
        v3 = ctx_1->dos_hdr.e_lfanew;
        if ( !v3 )
            return 2;
        if ( !read_proc(&ctx_1->hdr, v3, 0xF8u, info) )
            return 1;
        if ( ctx_1->hdr.Signature != 0x4550 )
            return 2;
        if ( ctx_1->hdr.OptionalHeader.Magic != 0x10B )
            return 2;
        v5 = ctx_1->hdr.FileHeader.NumberOfSections;
        if ( !(_WORD)v5 )
            return 2;
        size = 40 * v5;
        ctx_1->size_section_headers = 40 * v5;
        sect = (IMAGE_SECTION_HEADER *)malloc(40 * v5);
        ctx_1->sect = sect;
        if ( !sect )
            return 3;
        file_offset_section_headers = ctx_1->hdr.FileHeader.SizeOfOptionalHeader + ctx_1->dos_hdr.e_lfanew + 24;
        ctx_1->file_offset_section_headers = file_offset_section_headers;
        if ( !read_proc(sect, file_offset_section_headers, size, info) )
            return 1;
        result = 0;
    }
    else
    {
        result = 1;
    }
    return result;
}

```

Figure 8. Reconstructed source code of LoadDLL_LoadHeaders function from TVSpy dated back to 2015

First, we thought that the authors added new checks before calling read_proc function, making clear link between Olympic Destroyer and TVSpy (how, after all, could there be the same code modifications if the authors were not the same?). However, after further review we figured that read_proc didn't exist anymore. Instead it was compiled inline resulting in a statically linked memcpy function.

```

signed int __usercall LoadDLL_LoadHeaders@<eax>(LOAD_DLL_CONTEXT *ctx@<ecx>, LOAD_DLL_FILE_TO_LOAD *file_data)
{
    LOAD_DLL_CONTEXT *ctx_1; // esi@1
    LONG e_lfanew; // ecx@23
    int v4; // eax@7
    SIZE_T sectionTableSize; // eax@88
    IMAGE_SECTION_HEADER *sect; // ebx@88
    int file_offset_section_headers; // edx@10
    int size_section_headers; // ecx@10

    ctx_1 = ctx;
    if ( file_data->size >= 0x40u ) // inline check from read_proc
    {
        memcpy_static((unsigned int)ctx, (const void *)file_data->buffer, 0x40u);
        if ( ctx_1->dos_hdr.e_magic != 0x5A4D )
            return 2;
        e_lfanew = ctx_1->dos_hdr.e_lfanew;
        if ( !e_lfanew )
            return 2;
        if ( (unsigned int)(e_lfanew + 0xF8) <= file_data->size ) // inline check from read_proc
        {
            memcpy_static((unsigned int)&ctx_1->hdr, (const void *){e_lfanew + file_data->buffer}, 0xF8u);
            if ( ctx_1->hdr.Signature == 0x4550 && ctx_1->hdr.OptionalHeader.Magic == 0x10B )
            {
                v4 = ctx_1->hdr.FileHeader.NumberOfSections;
                if ( (_WORD)v4 )
                {
                    sectionTableSize = 40 * v4;
                    ctx_1->size_section_headers = sectionTableSize;
                    sect = (IMAGE_SECTION_HEADER *)malloc_static(sectionTableSize);
                    ctx_1->sect = sect;
                    if ( !sect )
                        return 3;
                    file_offset_section_headers = ctx_1->hdr.FileHeader.SizeOfOptionalHeader + ctx_1->dos_hdr.e_lfanew + 24;
                    ctx_1->file_offset_section_headers = file_offset_section_headers;
                    size_section_headers = ctx_1->size_section_headers;
                    if ( !size_section_headers )
                        return 0;
                    // inline check from read_proc
                    if ( (unsigned int)(file_offset_section_headers + size_section_headers) <= file_data->size )
                    {
                        memcpy_static(
                            (unsigned int)sect,
                            (const void *){file_offset_section_headers + file_data->buffer},
                            ctx_1->size_section_headers);
                        return 0;
                    }
                }
                return 1;
            }
        }
        return 2;
    }
    return 1;
}

```

Figure 9. Reconstructed LoadDLL_LoadHeaders from TVSpy and OlympicDestroyer samples, including additional check due to inlining of the read_proc function.

Also the meaningless check in LoadDll (“if (read_proc)”) we mentioned before has disappeared in the new version of the code (see Figure 10).

```

int __usercall LoadDLL@<eax>(LOAD_DLL_FILE_TO_LOAD *a1@<edx>, int a2, LOAD_DLL_INFO *info)
{
    int v3; // ebp@0
    LOAD_DLL_FILE_TO_LOAD *file_data; // ebx@1
    int res; // edi@1
    int result; // eax@2
    LOAD_DLL_CONTEXT ctx; // [sp+1Ch] [bp-178h]@1
    int dll_main; // [sp+174h] [bp-20h]@1
    CPPEH_RECORD ms_exc; // [sp+17Ch] [bp-18h]@1
    // no "if (read_proc)" check anymore,
    // since read_proc function doesn't exist
    file_data = a1;
    ctx.sect = 0;
    ctx.loaded_import_modules_array = 0;
    ctx.dll_main = 0;
    *(_DWORD *)&ctx.num_import_modules = 0;
    dll_main = 0;
    ms_exc.registration.TryLevel = 1;
    res = LoadDLL_LoadHeaders(&ctx, a1);
    if ( res
        || (res = LoadDLL_AllocateMemory(&ctx)) != 0
        || (ms_exc.registration.TryLevel = 2,
            (res = LoadDLL_LoadSections(&ctx, (LOAD_DLL_FROM_MEMORY_STRUCT *)file_data)) != 0)
        || (res = LoadDLL_PerformRelocation(&ctx)) != 0
        || (res = LoadDLL_ResolveImports(&ctx)) != 0
        || (res = LoadDLL_SetSectionMemoryProtection(&ctx)) != 0
        || (res = LoadDLL_CallDLLEntryPoint(&ctx)) != 0 )
    {
        local_unwind2(v3, &__security_cookie, (int)&ms_exc.registration, 0xFFFFFFFF);
        result = res;
    }
    else
    {
        if ( info )
        {
            info->size = 32;
            info->flags = 0;
            info->image_base = ctx.image_base;
            info->mem_block = ctx.image;
            info->dll_main = dll_main;
            info->export_dir_rva = ctx.hdr.OptionalHeader.DataDirectory[0].VirtualAddress;
            info->loaded_import_modules_array = ctx.loaded_import_modules_array;
            info->num_import_modules = *(_DWORD *)&ctx.num_import_modules;
            ms_exc.registration.TryLevel = 2;
        }
        local_unwind2(v3, &__security_cookie, (int)&ms_exc.registration, 0xFFFFFFFF);
        result = 0;
    }
    return result;
}

```

Figure 10. Reconstructed LoadDLL_LoadHeaders from TVSpy and Olympic Destroyer samples, including additional check due to inlining of the read_proc function.

The Bottom Line – Evidence is Inconclusive

In conclusion, we believe that this is not enough evidence to substantiate a claim that Olympic Destroyer and new versions of TVSpy using the same modified source code are built by the same author.

The more probable version for us is that the sample was built on a new compiler that further optimized the code. It would still mean that both new version of TVSpy and Olympic Destroyer are built using the same toolchain configured in the very same way (to enable full

optimization and link C++ runtime statically). We actually went to the extent of compiling the LoadDLL on MS Visual Studio 2017 with C++ runtime statically linked, and we managed to get the very same code as the one included in both Olympic Destroyer and TVSpy.

Although we would have liked to finally solve the dilemma, and unveil which were the actors behind the Olympic Destroyer attack, we ended up with more questions than answers, but admittedly, that's what research sometimes is about.

First, why would the authors of an allegedly state sponsored malware use an old LoadDLL project from an open source project from 2014? It is hard to believe that they could not come up with their own implementation or use much more advanced open-source projects for that, and definitely not relying on an educational prototype buried way beyond the first page of results in Google.

Or maybe the actors were not that much advanced as we would like to think, maybe seeing this as a one-time job, without enough resources to avoid using publicly available source code to quickly build their malware? Or maybe it's just another red flag, and the real authors decided to use the TVSpy source code as released in 2015 to leave a "Russian fingerprint"?

Maybe all of the above?

At the beginning of this article we stated that attribution is not just hard, it's often a wilderness of mirrors and more often than not, a bit anticlimactic. As a matter of fact, that was quite a precise prediction.

- [About](#)
- [Latest Posts](#)



Labs Team

Lastline Labs is where some of the most brilliant minds in the threat prevention community collaborate to develop advanced cyber security solutions. Our research team tracks the evolution, proliferation, and impact of advanced malware. The Lastline Labs Team continually monitors the threat landscape and analyzes new security threats and vulnerabilities.



Latest posts by Labs Team ([see all](#))

- [Threat Actor “Cold River”: Network Traffic Analysis and a Deep Dive on Agent Drable](#) - January 11, 2019
- [Tales From the Field: The Surge of Agent Tesla](#) - August 28, 2018
- [From Russia\(?\) with Code](#) - March 9, 2018

Tags:

[APT](#), [attribution](#), [code hash](#), [code re-usage](#), [code re-use](#), [Global Threat Intelligence Network](#), [Olympic Destroyer](#), [TVSpy](#)