Ramsay: A cyber-espionage toolkit tailored for air-gapped networks

welivesecurity.com/2020/05/13/ramsay-cyberespionage-toolkit-airgapped-networks

Ignacio Sanmillan

May 13, 2020

ESET researchers have discovered a previously unreported cyber-espionage framework that we named Ramsay and that is tailored for collection and exfiltration of sensitive documents and is capable of operating within air-gapped networks.

We initially found an instance of Ramsay in VirusTotal. That sample was uploaded from Japan and led us to the discovery of further components and versions of the framework, along with substantial evidence to conclude that this framework is at a developmental stage, with its delivery vectors still undergoing fine-tuning.

The current visibility of targets is low; based on ESET's telemetry, few victims have been discovered to date. We believe this scarcity of victims reinforces the hypothesis that this framework is under an ongoing development process, although the low visibility of victims could also be due to the nature of targeted systems being in air-gapped networks.

Shared artifacts were found alongside the Retro backdoor. This malware has been associated with Darkhotel, a notorious APT group known to have conducted cyber-espionage operations since at least 2004, having targeted government entities in China and Japan in the past.

Attack vectors

Along with the discovery of the different instances of Ramsay, we found they were leveraged using a series of attack vectors. These are:

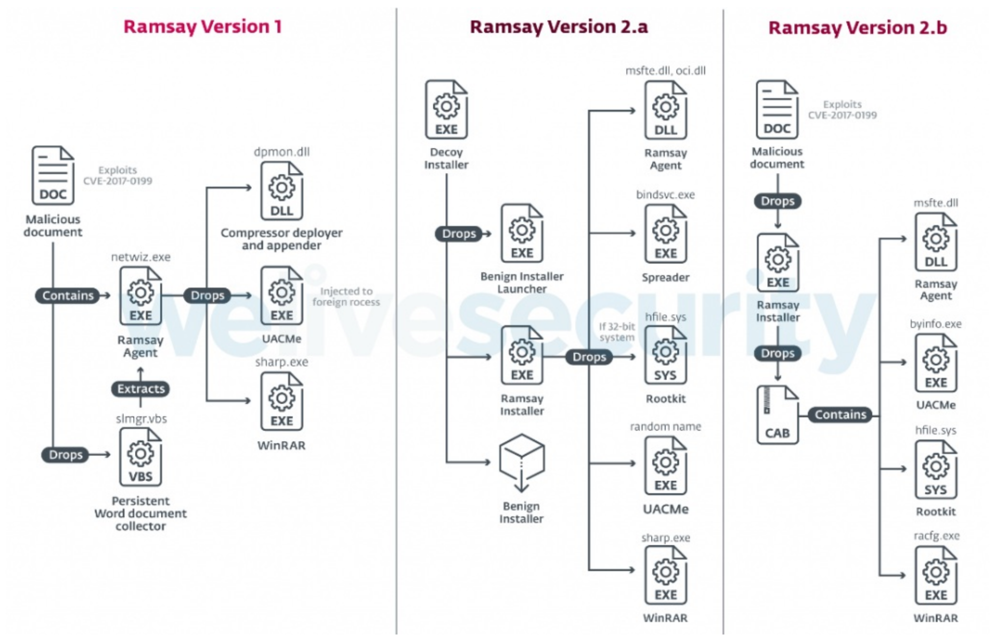


Figure 1. Overview of discovered Ramsay versions

Malicious documents dropping Ramsay version 1

This attack vector consists of malicious documents exploiting CVE-2017-0199 intended to drop an older version of Ramsay.

This document delivers an initial Visual Basic Script, shown in the screenshot below as OfficeTemporary.sct, that will extract within the document's body the Ramsay agent, masquerading as a JPG image by having a base64-encoded PE under a JPG header.

ID	Index	OLE Object
0	0x80c8	Format_id: 2 (Embedded) Class name: 'Package' Data size: 8994 OLE Package object: Filename: u'OfficeTemporary.sct' Source path: u'C:\\Intel\\OfficeTemporary.sct' Temp path = u:'C\\Intel\\OfficeTemporary.sct' MD5 = 'cf133c06180f130c471c95b3a4ebd7a5' EXECUTABLE FILE

ID	Index	OLE Object
1	0xc798	Format_id: 2 (Embedded) Class name: 'OLE2Link' Data size: 2560 MD5 = 'daee337d42fba92badbea2a4e085f73f' CLSID: 00000300-0000-0000-C000-000000000046 StdOleLink (embedded OLE object - known related to CVE-2017-0199, CVE-2017-8570, CVE-2017-8759 or CVE-2018-8174. Possibly an exploit for the OLE2Link vulnerability (VU#921560, CVE-2017-0199)

Table 1. OLE object layout contained within Ramsay version 1 RTF file as seen by oletools

We noticed that the specific Ramsay instance dropped by these documents showed low complexity in its implementation and lacked many of the more advanced features seen leveraged by later Ramsay versions.

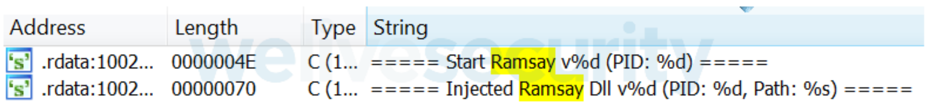
Several instances of these same malicious documents were found uploaded to public sandbox engines, labeled as testing artifacts such as *'access_test.docx*' or '*Test.docx*' denoting an ongoing effort for trial of this specific attack vector.

Based on the low complexity of the Ramsay agent delivered, the threat actors may be embedding this specific instance within these malicious documents for evaluation purposes.

Decoy installer dropping Ramsay version 2.a

We found one instance uploaded to VirusTotal of Ramsay masquerading as a 7zip installer.

The reason we named this malware Ramsay was due to some of the strings contained in this binary, such as the following:



Address	Length	Type	String
.rdata:1002...	0000004E	C (1...	===== Start Ramsay v%d (PID: %d) =====
.rdata:1002...	00000070	C (1...	===== Injected Ramsay Dll v%d (PID: %d, Path: %s) =====

Figure 2. Strings containing "Ramsay"

This version of Ramsay shows a clear refinement of its evasion and persistence tactics along with the introduction of new features such as a Spreader component and a rootkit; the Spreader component is documented more thoroughly in this part of the Capabilities section.

Malicious documents dropping Ramsay version 2.b

This attack vector consists of the delivery of a different malicious document abusing CVE-2017-11882. This document will drop a Ramsay Installer named lmsch.exe as shown in Table 2.

ID	Index	OLE Object
0	0x80c8	Format_id: 2 (Embedded) Class name: 'Package' Data size: 644790 OLE Package object: Filename: u'lmsch.exe' Source path: u'C:\\fakepath\\lmsch.exe' Temp path = u:'C:\\fakepath\\lmsch.exe' MD5 = '27cd5b330a93d891bdcbd08050a5a6e1'
1	0xc798	Format_id: 2 (Embedded) Class name: 'Equation.3' Data size: 3584 MD5 = '5ae434c951b106d63d79c98b1a95e99d' CLSID: 0002CE02-0000-0000-C000-000000000046 Microsoft Equation 3.0 (Known related to CVE-2017-11882 or CVE-2018-0802) Possibly an exploit for the Equation Editor vulnerability (VU#421280, CVE-2017-11882)

Table 2. OLE object layout contained within Ramsay version 2.b RTF file as seen by oletools

The Ramsay version leveraged by this document is a slightly modified version of Ramsay version 2.a, with the main difference of not leveraging the spreader component. The functionality of the remaining components is the same in regard to Ramsay version 2.a.

Client Execution of Infected Files

As previously mentioned, Ramsay Version 2.a delivers a Spreader component that will behave as a file infector, changing the structure of benign PE executable files held within removable and network shared drives in order to embed malicious Ramsay artifacts triggered on host file execution.

The Spreader is highly aggressive in its propagation mechanism and any PE executables residing in the targeted drives would be candidates for infection.

Based on compilation timestamps among the components of the various versions of Ramsay found, we can estimate the following development timeline of this framework:

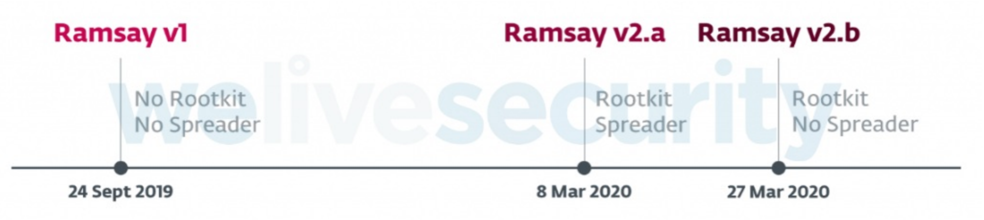


Figure 3. Estimation of Ramsay's development timeline

The analysis of the different compilation timestamps found across different components implies that this framework has been under development since late 2019, with the possibility of currently having two maintained versions tailored based on the configuration of different targets.

Persistence mechanisms

Based on its version, Ramsay implements various persistence mechanisms of different complexity. Some of these persistence mechanisms are the following:

AppInit DLL registry key

The Windows operating system provides the functionality to allow custom DLLs to be loaded into the address space of almost all application processes via AppInit DLL registry key. This technique is not particularly complex; it is implemented in early Ramsay versions and is common in other malware families.

Scheduled Task via COM API

Scheduled tasks enable administrators to run tasks or "jobs" at designated times rather than every time the system is booted or the user logs in. This feature can be implemented via the Windows COM API, which the first versions of Ramsay have tailored. Based on the high ratio of similarity with Carberp's implementation, it's highly probable that Ramsay's implementation was adapted from Carberp's publicly available source code.

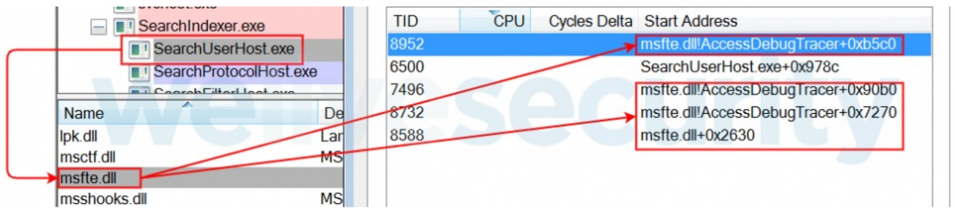
Phantom DLL Hijacking

More mature versions of Ramsay denote an increase in complexity of its persistence techniques, which include a technique sometimes referred to as "Phantom DLL Hijacking".

Phantom DLL Hijacking abuses the fact that many Windows applications use outdated dependencies not strictly necessary for the functionality of the application itself, allowing the possibility of leveraging malicious versions of these dependencies.

Two services will be targeted in order to enforce this technique. These are:

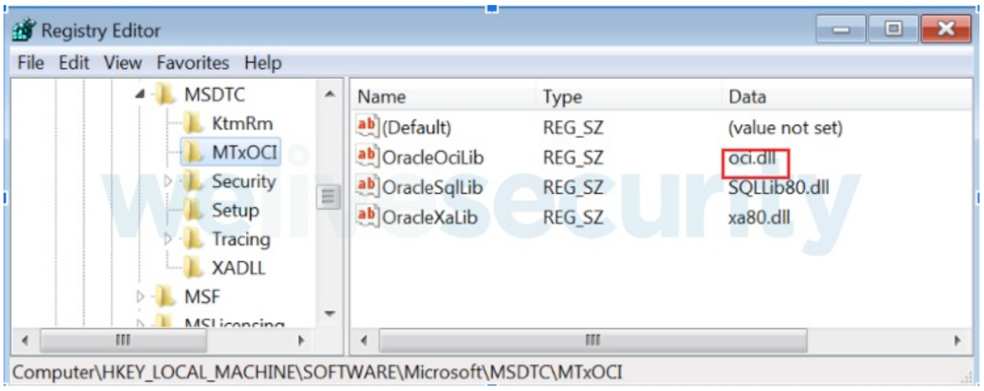
WSearch (Windows Search) hijacking msfte.dll:



TID	CPU	Cycles Delta	Start Address
8952			msfte.dll!AccessDebugTracer+0xb5c0
6500			SearchUserHost.exe+0x978c
7496			msfte.dll!AccessDebugTracer+0x90b0
8732			msfte.dll!AccessDebugTracer+0x7270
8588			msfte.dll+0x2630

Figure 4. Hijacking of Microsoft Search Service msfte.dll

MSDTC (Microsoft Distributed Transaction Coordinator) service hijacking an oracle dependency seen below as oci.dll:



Name	Type	Data
(Default)	REG_SZ	(value not set)
OracleOciLib	REG_SZ	oci.dll
OracleSqlLib	REG_SZ	SQLLib80.dll
OracleXaLib	REG_SZ	xa80.dll

Computer\HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\MSDTC\MTxOCI

Figure 5. Hijacking of MSDTC service dependency oci.dll

This persistence technique is highly versatile, enabling Ramsay agents delivered as DLLs to fragment their logic into separated sections, implementing different functionality tailored for the subject processes where the agent will be loaded. In addition, the use of this technique makes detection more difficult since the loading of these DLLs into their respective processes/services won't necessarily trigger an alert.

Capabilities

Ramsay's architecture provides a series of capabilities monitored via a logging mechanism intended to assist operators by supplying a feed of actionable intelligence to conduct exfiltration, control, and lateral movement actions, as well as providing overall behavioral and system statistics of each compromised system. The realization of these actions is possible due to the following capabilities:

File collection and covert storage

The primary goal of this framework is to collect all existing *Microsoft Word* documents within the target's filesystem. The overall collection stages are shown in Figure 6:

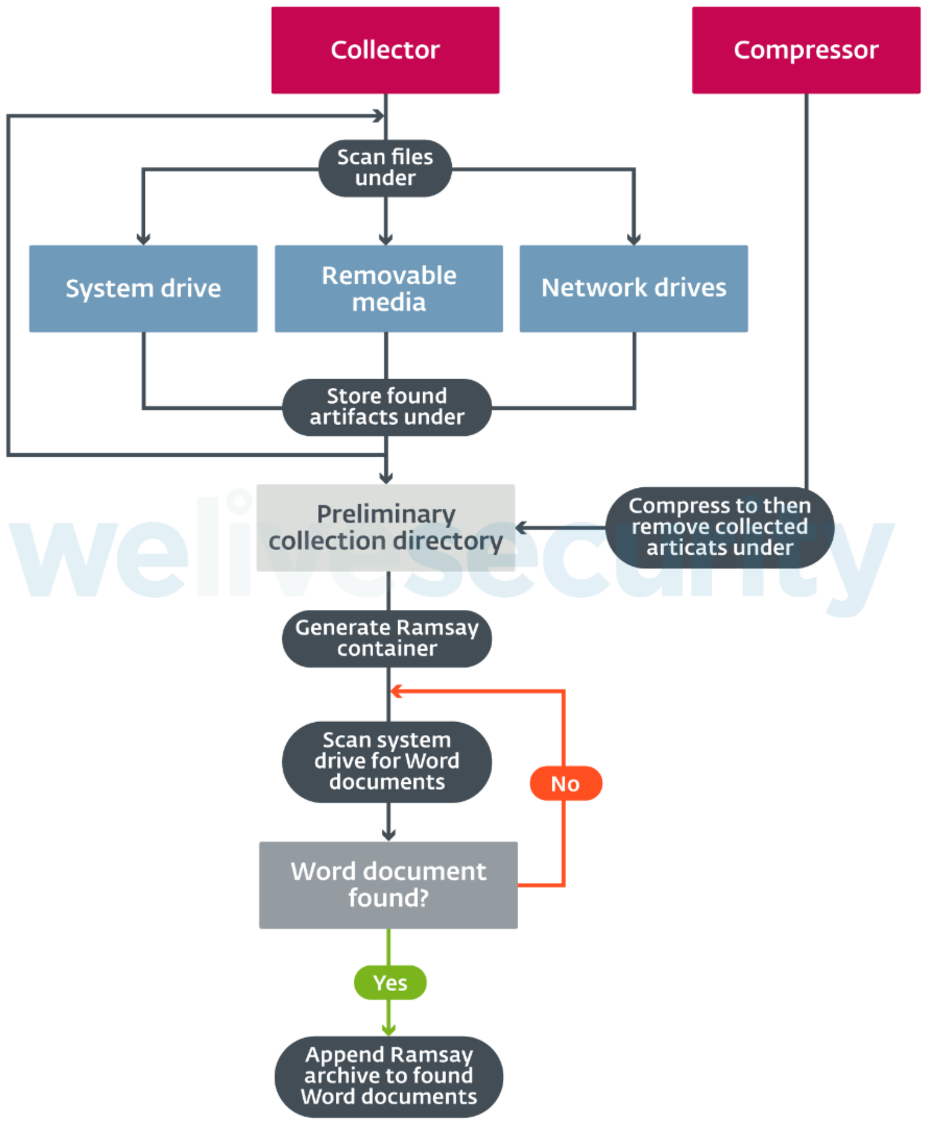
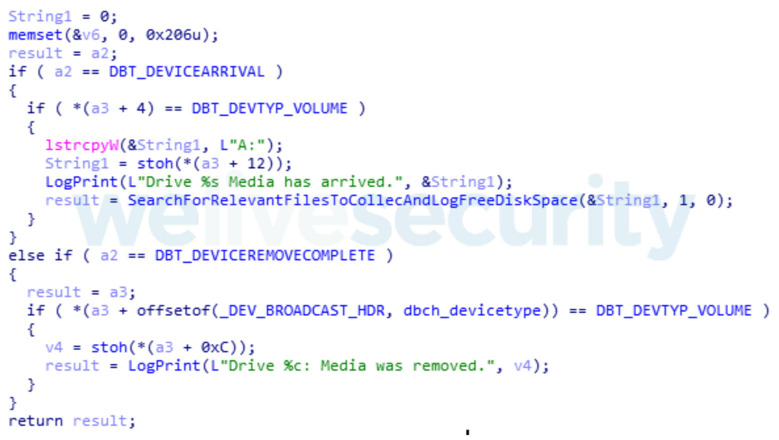

Collector
Compressor
Scan files under
System drive
Removable media
Network drives
Store found artifacts under
Preliminary collection directory
Compress to then remove collected articats under
Generate Ramsay container
Scan system drive for Word documents
No
Word document found?
Yes
Append Ramsay archive to found Word documents
welivesecurity

Figure 6. Mechanism of document collection

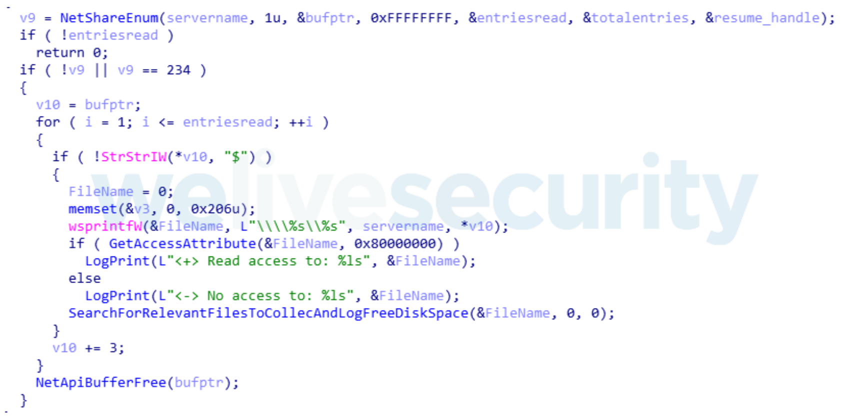
Word documents will first be collected and stored in a preliminary collection directory. The location of this directory may vary depending on the Ramsay version. Two of the directories we observed being used for this purpose were %APPDATA%\Microsoft\UserSetting and %APPDATA%\Microsoft\UserSetting\MediaCache.

Depending on the Ramsay version, file collection won't be restricted to the local system drive, but also will search additional drives such as network or removable drives:



```
String1 = 0;
memset(&v6, 0, 0x206u);
result = a2;
if ( a2 == DBT_DEVICEARRIVAL )
{
  if ( *(a3 + 4) == DBT_DEVTYP_VOLUME )
  {
    lstrcpyW(&String1, L"A:");
    String1 = stoh(*(a3 + 12));
    LogPrint(L"Drive %s Media has arrived.", &String1);
    result = SearchForRelevantFilesToCollecAndLogFreeDiskSpace(&String1, 1, 0);
  }
}
else if ( a2 == DBT_DEVICEREMOVECOMPLETE )
{
  result = a3;
  if ( *(a3 + offsetof(_DEV_BROADCAST_HDR, dbch_devicetype)) == DBT_DEVTYP_VOLUME )
  {
    v4 = stoh(*(a3 + 0xC));
    result = LogPrint(L"Drive %c: Media was removed.", v4);
  }
}
return result;
```

Figure 7. Hex-Rays output of procedure to scan removable drives for collection

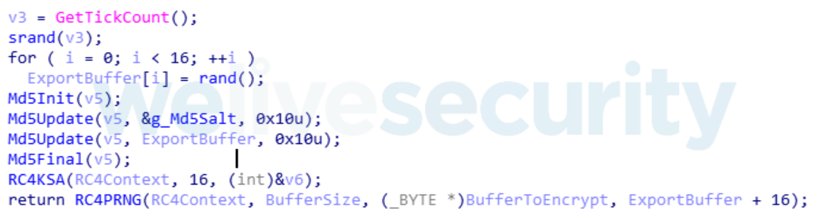


```
v9 = NetShareEnum(servername, 1u, &bufptr, 0xFFFFFFFF, &entriesread, &totalentries, &resume_handle);
if ( !entriesread )
  return 0;
if ( !v9 || v9 == 234 )
{
  v10 = bufptr;
  for ( i = 1; i <= entriesread; ++i )
  {
    if ( !StrStrIW(*v10, "$") )
    {
      FileName = 0;
      memset(&v3, 0, 0x206u);
      wsprintfW(&FileName, L"\\\\%s\\%s", servername, *v10);
      if ( GetAccessAttribute(&FileName, 0x80000000) )
        LogPrint(L"<+> Read access to: %ls", &FileName);
      else
        LogPrint(L"<-> No access to: %ls", &FileName);
      SearchForRelevantFilesToCollecAndLogFreeDiskSpace(&FileName, 0, 0);
    }
    v10 += 3;
  }
  NetApiBufferFree(bufptr);
}
```

Figure 8. Hex-Rays output of procedure to scan network drives for collection

Collected documents are encrypted using the RC4 Stream Cipher Algorithm.

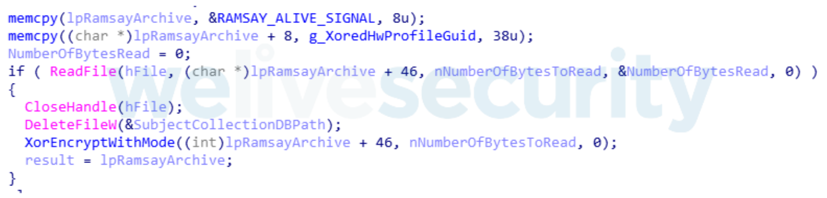
The RC4 key used to encrypt each file will be a computed MD5 hash of a randomly generated sequence of 16 bytes, salted with 16 bytes hardcoded in the malware sample. The first 16 bytes of the buffer where the encrypted file will be held will correspond to the actual RC4 key used:



```
v3 = GetTickCount();
srand(v3);
for ( i = 0; i < 16; ++i )
  ExportBuffer[i] = rand();
Md5Init(v5);
Md5Update(v5, &g_Md5Salt, 0x10u);
Md5Update(v5, ExportBuffer, 0x10u);
Md5Final(v5);
RC4KSA(RC4Context, 16, (int)&v6);
return RC4PRNG(RC4Context, BufferSize, (_BYTE *)BufferToEncrypt, ExportBuffer + 16);
```

Figure 9. Hex-Rays output of RC4 key generation and storage

Collected files under the preliminary collection directory will be compressed using a WinRAR instance that the Ramsay Installer drops. This compressed archive will be saved within the preliminary collection directory and then generate a Ramsay container artifact:

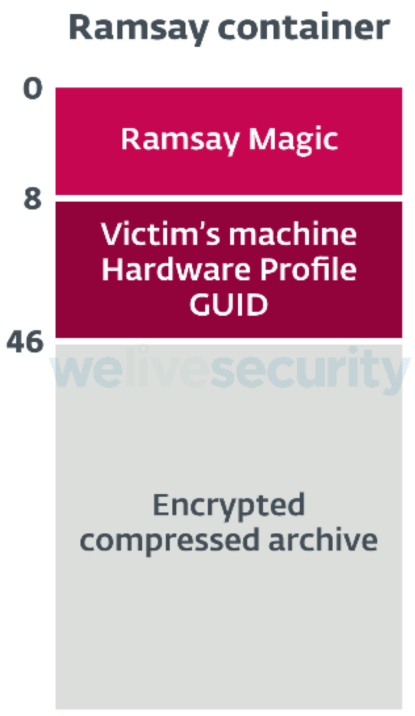


```
memcpy(lpRamsayArchive, &RAMSAY_ALIVE_SIGNAL, 8u);
memcpy((char *)lpRamsayArchive + 8, g_XoredHwProfileGuid, 38u);
NumberOfBytesRead = 0;
if ( ReadFile(hFile, (char *)lpRamsayArchive + 46, nNumberOfBytesToRead, &NumberOfBytesRead, 0) )
{
  CloseHandle(hFile);
  DeleteFileW(&SubjectCollectionDBPath);
  XorEncryptWithMode((int)lpRamsayArchive + 46, nNumberOfBytesToRead, 0);
  result = lpRamsayArchive;
}
```

Figure 10. Hex-Rays output of Ramsay container generation

As shown in the previous screenshot, these Ramsay containers contain a magic value at the beginning of the file, along with a Hardware Profile

GUID denoting an identifier of the victim's machine; an additional XOR-based encryption layer will be applied to the generated compressed archive. The following diagram shows the structure of these artifacts:



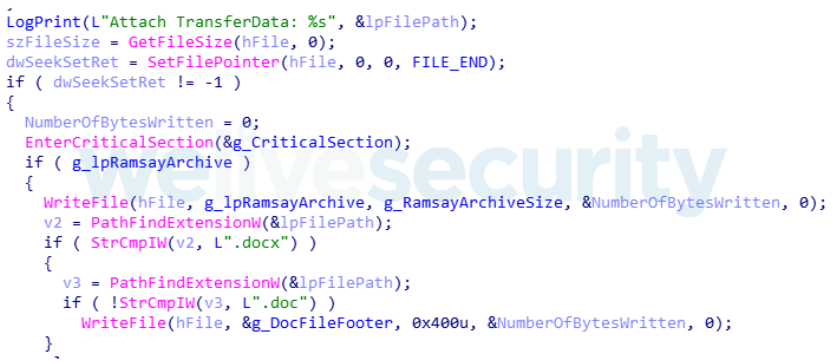
Ramsay container
0
Ramsay Magic
8
Victim's machine Hardware Profile GUID
46
Encrypted compressed archive
welivesecurity

Figure 11. Ramsay Container Structure<{i>

Ramsay implements a decentralized way of storing these artifacts among the victim's file system by using inline hooks applied on two Windows API functions, WriteFile and CloseHandle.

The hooked WriteFile procedure's main purpose is to save the file handle of the subject file to write and install another hook in the CloseHandle API function. The CloseHandle hooked procedure will then check whether the subject file name has a .doc extension; if that's the case, it will then append at the end of the subject document the Ramsay container artifact followed by a stream of 1024 bytes denoting a Microsoft Word document footer.

This is done as an evasion measure in order to provide a means to hide the embedded artifact within the subject document from the naked eye:



```
LogPrint(L"Attach TransferData: %s", &lpFilePath);
szFileSize = GetFileSize(hFile, 0);
dwSeekSetRet = SetFilePointer(hFile, 0, 0, FILE_END);
if ( dwSeekSetRet != -1 )
{
  NumberOfBytesWritten = 0;
  EnterCriticalSection(&g_CriticalSection);
  if ( g_lpRamsayArchive )
  {
    WriteFile(hFile, g_lpRamsayArchive, g_RamsayArchiveSize, &NumberOfBytesWritten, 0);
    v2 = PathFindExtensionW(&lpFilePath);
    if ( StrCmpIW(v2, L".docx") )
    {
      v3 = PathFindExtensionW(&lpFilePath);
      if ( !StrCmpIW(v3, L".doc") )
        WriteFile(hFile, &g_DocFileFooter, 0x400u, &NumberOfBytesWritten, 0);
    }
```

Figure 12. Hex-Rays output of code for appending Word document footer at the end of the target document

Ramsay containers appended to Word documents will be marked in order to avoid redundant artifacts being appended to already affected documents and the preliminary storage directory will be cleared in order to generate a brand-new Ramsay artifact in intervals.

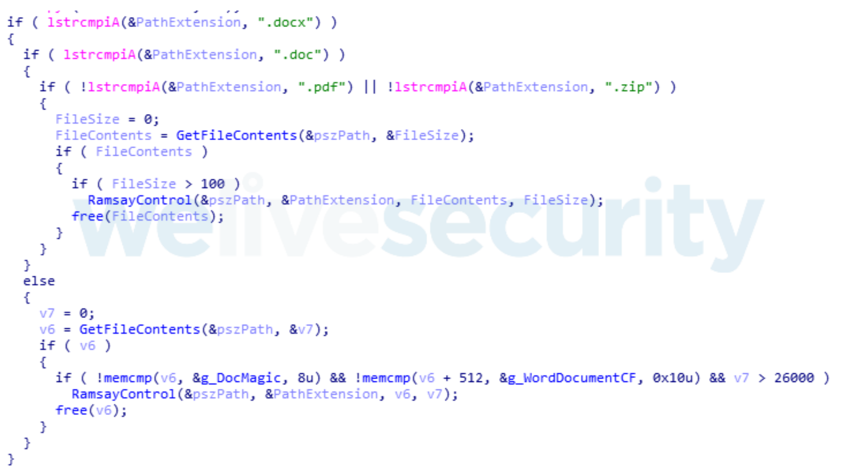
Even though affected documents will be modified, it won't impact their integrity; each affected Word document remains fully operational after artifact appending has taken place.

Exfiltration of these artifacts is done via an external component that we haven't been able to retrieve. However, based on the decentralized methodology Ramsay implements for storage of collected artifacts, we believe this component would scan the victim's file system in search for the Ramsay container's magic values, in order to identify the location of artifacts to exfiltrate.

Command execution

Unlike most conventional malware, Ramsay does not have a network-based C&C communication protocol nor does it make any attempt to connect to a remote host for communication purposes. Ramsay's control protocol follows the same decentralized philosophy implemented for collected artifact storage.

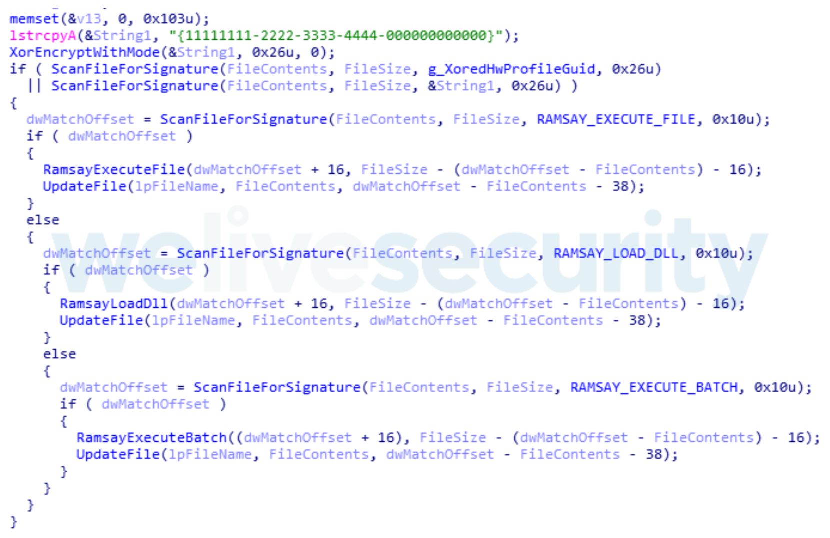
Ramsay will scan all the network shares and removable drives *(*excluding A: and B: drives usually reserved for floppy disks) for potential control files. First, Ramsay looks for Word documents and also, in more recent versions, for PDFs and ZIP archives:



```
if ( lstrcmpiA(&PathExtension, ".docx") )
{
  if ( lstrcmpiA(&PathExtension, ".doc") )
  {
    if ( !lstrcmpiA(&PathExtension, ".pdf") || !lstrcmpiA(&PathExtension, ".zip") )
    {
      FileSize = 0;
      FileContents = GetFileContents(&pszPath, &FileSize);
      if ( FileContents )
      {
        if ( FileSize > 100 )
          RamsayControl(&pszPath, &PathExtension, FileContents, FileSize);
        free(FileContents);
      }
    }
  }
  else
  {
    v7 = 0;
    v6 = GetFileContents(&pszPath, &v7);
    if ( v6 )
    {
      if ( !memcmp(v6, &g_DocMagic, 8u) && !memcmp(v6 + 512, &g_WordDocumentCF, 0x10u) && v7 > 26000 )
        RamsayControl(&pszPath, &PathExtension, v6, v7);
      free(v6);
    }
  }
}
```

Figure 13. Hex-Rays output of Ramsay Scan procedure for Control File retrieval

These files are parsed for the presence of a magic marker specific to the control file format. More specifically, Ramsay looks for any of two given encoded Hardware Profile GUIDs. One of these GUIDs is hardcoded as shown in Figure 14, while the other is dynamically generated based on the compromised victim's machine. If any of the subject identifiers are found, parsing for a command signature will be attempted.



```
memset(&v13, 0, 0x103u);
lstrcpyA(&String1, "{11111111-2222-3333-4444-000000000000}");
XorEncryptWithMode(&String1, 0x26u, 0);
if ( ScanFileForSignature(FileContents, FileSize, g_XoredHwProfileGuid, 0x26u)
  || ScanFileForSignature(FileContents, FileSize, &String1, 0x26u) )
{
  dwMatchOffset = ScanFileForSignature(FileContents, FileSize, RAMSAY_EXECUTE_FILE, 0x10u);
  if ( dwMatchOffset )
  {
    RamsayExecuteFile(dwMatchOffset + 16, FileSize - (dwMatchOffset - FileContents) - 16);
    UpdateFile(lpFileName, FileContents, dwMatchOffset - FileContents - 38);
  }
  else
  {
    dwMatchOffset = ScanFileForSignature(FileContents, FileSize, RAMSAY_LOAD_DLL, 0x10u);
    if ( dwMatchOffset )
    {
      RamsayLoadDll(dwMatchOffset + 16, FileSize - (dwMatchOffset - FileContents) - 16);
      UpdateFile(lpFileName, FileContents, dwMatchOffset - FileContents - 38);
    }
    else
    {
      dwMatchOffset = ScanFileForSignature(FileContents, FileSize, RAMSAY_EXECUTE_BATCH, 0x10u);
      if ( dwMatchOffset )
      {
        RamsayExecuteBatch((dwMatchOffset + 16), FileSize - (dwMatchOffset - FileContents) - 16);
        UpdateFile(lpFileName, FileContents, dwMatchOffset - FileContents - 38);
      }
    }
  }
}
```

Figure 14. Hex-Rays output of Ramsay Control File Parsing

The search for these two GUID instances implies that Ramsay's control documents can be deliberately crafted to be "victim agnostic", capable of deploying the same control document instance across a number of victims by leveraging a "global" GUID within control documents. On the other hand, control documents can be crafted by embedding a specific GUID intended to be delivered exclusively on a single victim's machine. This indicator of Ramsay's control protocol implementation implies that its backend counterpart may be somewhat automated.

Ramsay control protocol supports three different commands:

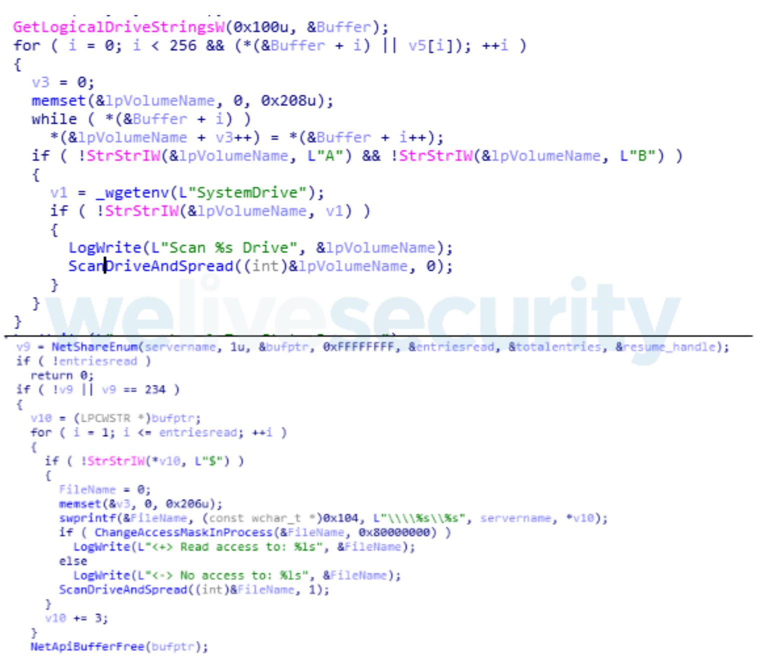
Signature	Command
Rr*e#R79m3QNU3Sy	File Execution
CNDkS_&pgaU#7Yg9	DLL Load
2DWcdSqcv3?(XYqT	Batch Execution

Table 3. Ramsay's control commands

After a given command signature is retrieved, the contained artifact to execute will be extracted within the control document's body to then be restored, modifying the subject control document to its original form after command execution.

Spreading

Among the different files dropped by the latest versions of Ramsay we find a *Spreader* component. This executable will attempt to scan for network shares and removable drives excluding A: and B: drives:



```
GetLogicalDriveStringsW(0x100u, &Buffer);
for ( i = 0; i < 256 && (*(&Buffer + i) || v5[i]); ++i )
{
  v3 = 0;
  memset(&lpVolumeName, 0, 0x208u);
  while ( *(&Buffer + i) )
    *(&lpVolumeName + v3++) = *(&Buffer + i++);
  if ( !StrStrIW(&lpVolumeName, L"A") && !StrStrIW(&lpVolumeName, L"B") )
  {
    v1 = _wgetenv(L"SystemDrive");
    if ( !StrStrIW(&lpVolumeName, v1) )
    {
      LogWrite(L"Scan %s Drive", &lpVolumeName);
      ScanDriveAndSpread((int)&lpVolumeName, 0);
    }
  }
}
```

```
v9 = NetShareEnum(servername, 1u, &bufptr, 0xFFFFFFFF, &entriesread, &totalentries, &resume_handle);
if ( !entriesread )
  return 0;
if ( !v9 || v9 == 234 )
{
  v10 = (LPCWSTR *)bufptr;
  for ( i = 1; i <= entriesread; ++i )
  {
    if ( !StrStrIW(*v10, L"$") )
    {
      FileName = 0;
      memset(&v3, 0, 0x206u);
      swprintf(&FileName, (const wchar_t *)0x104, L"\\\\%s\\%s", servername, *v10);
      if ( ChangeAccessMaskInProcess(&FileName, 0x80000000) )
        LogWrite(L"<+> Read access to: %ls", &FileName);
      else
        LogWrite(L"<-> No access to: %ls", &FileName);
      ScanDriveAndSpread((int)&FileName, 1);
    }
    v10 += 3;
  }
  NetApiBufferFree(bufptr);
```

Figure 15. Hex-Rays output of spreader scanning routines

It is important to notice that there is a correlation between the target drives Ramsay scans for propagation and control document retrieval. This assesses the relationship between Ramsay's spreading and control capabilities showing how Ramsay's operators leverage the framework for lateral movement, denoting the likelihood that this framework has been designed to operate within air-gapped networks.

The propagation technique mainly consists of file infection much like a prepender file infector in order to generate executables similar in structure to Ramsay's decoy installers for every accessible PE file within the aforementioned targeted drives. The following diagram illustrates the changes applied to targeted executables after infection has taken place and how these components interact on execution:

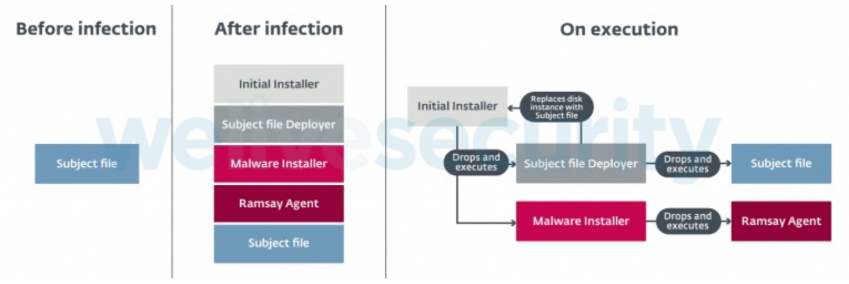
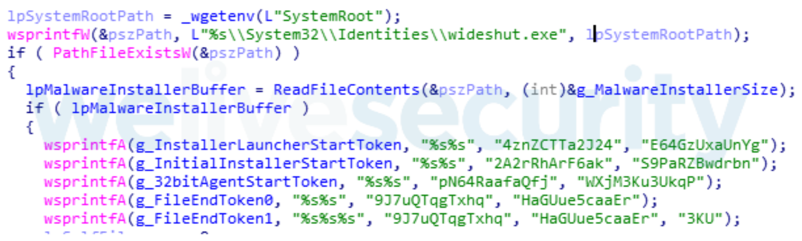


Figure 16. File structure changes during an infection and execution

All of the different artifacts involved in the infection stage are either within the context of the spreader or dropped previously by another Ramsay component. Some of the artifacts are parsed for the following tokens:



```
lpSystemRootPath = _wgetenv(L"SystemRoot");
wsprintfW(&pszPath, L"%s\\System32\\Identities\\wideshut.exe", lpSystemRootPath);
if ( PathFileExistsW(&pszPath) )
{
  lpMalwareInstallerBuffer = ReadFileContents(&pszPath, (int)&g_MalwareInstallerSize);
  if ( lpMalwareInstallerBuffer )
  {
    wsprintfA(g_InstallerLauncherStartToken, "%s%s", "4znZCTTa2J24", "E64GzUxaUnYg");
    wsprintfA(g_InitialInstallerStartToken, "%s%s", "2A2rRhArF6ak", "S9PaRZBwdrbn");
    wsprintfA(g_32bitAgentStartToken, "%s%s", "pN64RaafaQfj", "WXjM3Ku3UkqP");
    wsprintfA(g_FileEndToken0, "%s%s", "9J7uQTqgTxhq", "HaGUue5caaEr");
    wsprintfA(g_FileEndToken1, "%s%s%s", "9J7uQTqgTxhq", "HaGUue5caaEr", "3KU");
```

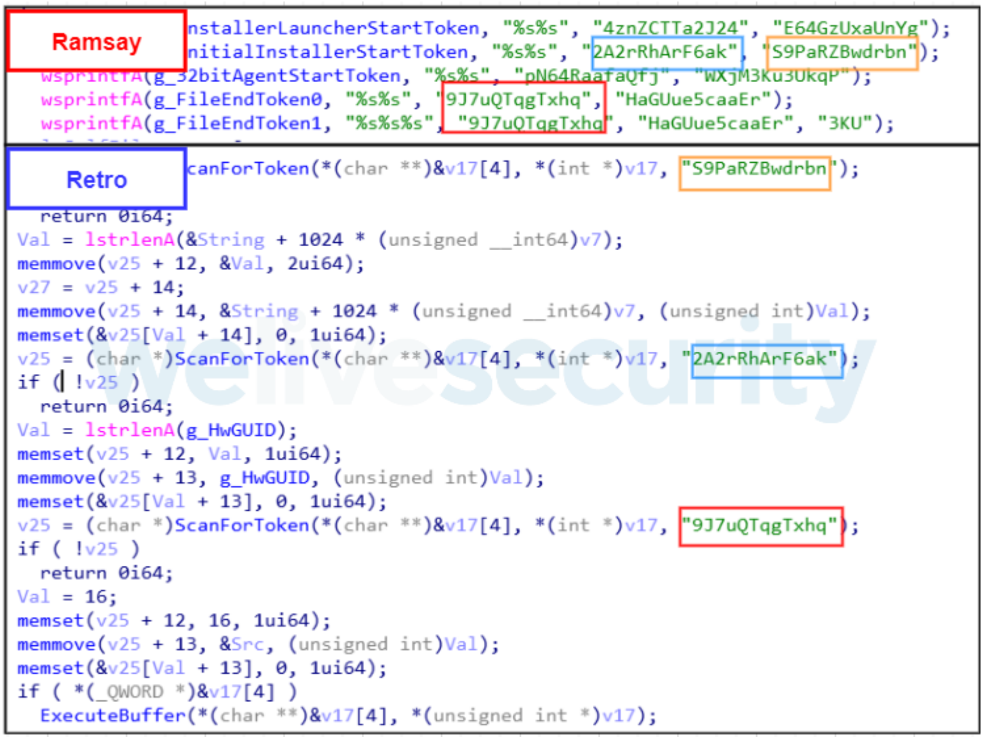
Figure 17. Hex-Rays output of tokens to search for different artifacts within the spreader context

After a given file has been infected, it will be marked by writing a specific token at the end of it in order to provide the spreader an identifier to prevent redundant infection.

In addition, some components of Ramsay have implemented a network scanner intended for the discovery of machines within the compromised host's subnet that are susceptible to the EternalBlue SMBv1 vulnerability. This information will be contained within all logged information Ramsay collects and may be leveraged by operators in order to do further lateral movement over the network in a later stage via a different channel.

Further remarks

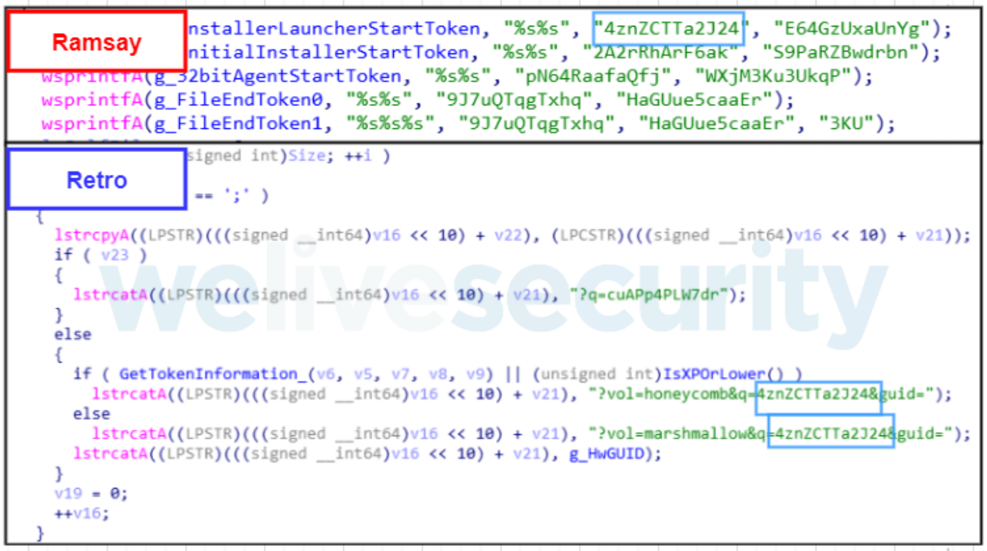
Ramsay's version 2.a Spreader component was found to have reused a series of tokens seen before in Darkhotel's Retro Backdoor. These tokens are the following:



```
nstallerLauncherStartToken, "%s%s", "4znZCTTa2J24", "E64GzUxaUnYg");
nitialInstallerStartToken, "%s%s", "2A2rRhArF6ak", "S9PaRZBwdrbn");
wsprintfA(g_32bitAgentStartToken, "%s%s", "pN64RaafaQfj", "WXjM3Ku3UkqP");
wsprintfA(g_FileEndToken0, "%s%s", "9J7uQTqgTxhq", "HaGUue5caaEr");
wsprintfA(g_FileEndToken1, "%s%s%s", "9J7uQTqgTxhq", "HaGUue5caaEr", "3KU");
```

```
canForToken(*(char **)&v17[4], *(int *)v17, "S9PaRZBwdrbn");
  return 0i64;
Val = lstrlenA(&String + 1024 * (unsigned __int64)v7);
memmove(v25 + 12, &Val, 2ui64);
v27 = v25 + 14;
memmove(v25 + 14, &String + 1024 * (unsigned __int64)v7, (unsigned int)Val);
memset(&v25[Val + 14], 0, 1ui64);
v25 = (char *)ScanForToken(*(char **)&v17[4], *(int *)v17, "2A2rRhArF6ak");
if ( !v25 )
  return 0i64;
Val = lstrlenA(g_HwGUID);
memset(v25 + 12, Val, 1ui64);
memmove(v25 + 13, g_HwGUID, (unsigned int)Val);
memset(&v25[Val + 13], 0, 1ui64);
v25 = (char *)ScanForToken(*(char **)&v17[4], *(int *)v17, "9J7uQTqgTxhq");
if ( !v25 )
  return 0i64;
Val = 16;
memset(v25 + 12, 16, 1ui64);
memmove(v25 + 13, &Src, (unsigned int)Val);
memset(&v25[Val + 13], 0, 1ui64);
if ( *(_QWORD *)&v17[4] )
  ExecuteBuffer(*(char **)&v17[4], *(unsigned int *)v17);
```

Figure 18. Hex-Rays output of Token Reuse with Retro

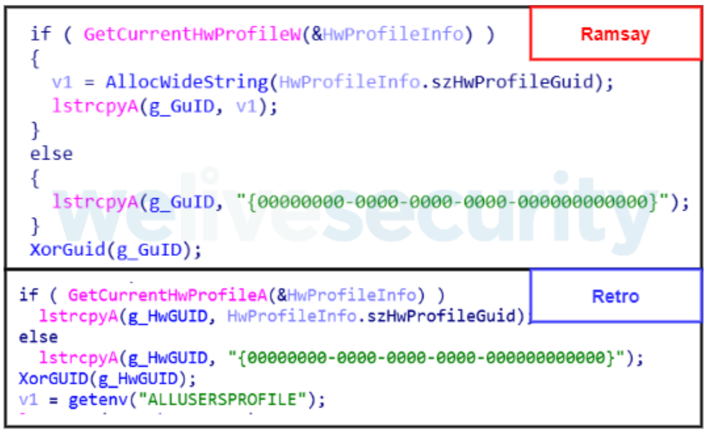


```
nstallerLauncherStartToken, "%s%s", "4znZCTTa2J24", "E64GzUxaUnYg");
nitialInstallerStartToken, "%s%s", "2A2rRhArF6ak", "S9PaRZBwdrbn");
wsprintfA(g_32bitAgentStartToken, "%s%s", "pN64RaafaQfj", "WXjM3Ku3UkqP");
wsprintfA(g_FileEndToken0, "%s%s", "9J7uQTqgTxhq", "HaGUue5caaEr");
wsprintfA(g_FileEndToken1, "%s%s%s", "9J7uQTqgTxhq", "HaGUue5caaEr", "3KU");
```

```
signed int)Size; ++i )
 == ';' )
{
  lstrcpyA((LPSTR)(((signed __int64)v16 << 10) + v22), (LPCSTR)(((signed __int64)v16 << 10) + v21));
  if ( v23 )
  {
    lstrcatA((LPSTR)(((signed __int64)v16 << 10) + v21), "?q=cuAPp4PLW7dr");
  }
  else
  {
    if ( GetTokenInformation_(v6, v5, v7, v8, v9) || (unsigned int)IsXPOrLower() )
      lstrcatA((LPSTR)(((signed __int64)v16 << 10) + v21), "?vol=honeycomb&q=4znZCTTa2J24&guid=");
    else
      lstrcatA((LPSTR)(((signed __int64)v16 << 10) + v21), "?vol=marshmallow&q=4znZCTTa2J24&guid=");
    lstrcatA((LPSTR)(((signed __int64)v16 << 10) + v21), g_HwGUID);
  }
  v19 = 0;
  ++v16;
}
```

Figure 19. Token Reuse on Retro URL Crafting

Ramsay serializes victims using the GetCurrentHwProfile API to then retrieve a GUID for the specific victim's machine. This is also seen implemented in Retro. They both use the same default GUID in case the API call fails:

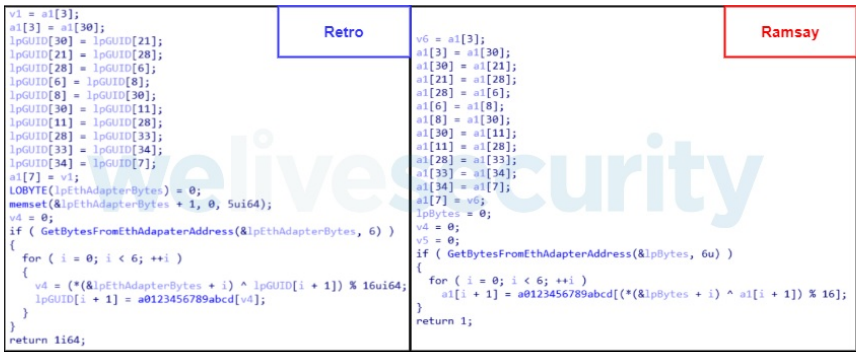


```
if ( GetCurrentHwProfileW(&HwProfileInfo) )
{
  v1 = AllocWideString(HwProfileInfo.szHwProfileGuid);
  lstrcpyA(g_GuID, v1);
}
else
{
  lstrcpyA(g_GuID, "{00000000-0000-0000-0000-000000000000}");
}
XorGuid(g_GuID);
```

```
if ( GetCurrentHwProfileA(&HwProfileInfo) )
  lstrcpyA(g_HwGUID, HwProfileInfo.szHwProfileGuid);
else
  lstrcpyA(g_HwGUID, "{00000000-0000-0000-0000-000000000000}");
XorGUID(g_HwGUID);
v1 = getenv("ALLUSERSPROFILE");
```

Figure 20. Ramsay and Retro GUID generation

Both Ramsay and Retro share the same encoding algorithm to encode the retrieved GUID.



```
v1 = a1[3];
a1[3] = a1[30];
lpGUID[30] = lpGUID[21];
lpGUID[21] = lpGUID[28];
lpGUID[28] = lpGUID[6];
lpGUID[6] = lpGUID[8];
lpGUID[8] = lpGUID[30];
lpGUID[30] = lpGUID[11];
lpGUID[11] = lpGUID[28];
lpGUID[28] = lpGUID[33];
lpGUID[33] = lpGUID[34];
lpGUID[34] = lpGUID[7];
a1[7] = v1;
LOBYTE(lpEthAdapterBytes) = 0;
memset(&lpEthAdapterBytes + 1, 0, 5ui64);
v4 = 0;
if ( GetBytesFromEthAdapaterAddress(&lpEthAdapterBytes, 6) )
{
  for ( i = 0; i < 6; ++i )
  {
    v4 = (*(&lpEthAdapterBytes + i) ^ lpGUID[i + 1]) % 16ui64;
    lpGUID[i + 1] = a0123456789abcd[v4];
  }
}
return 1i64;
```

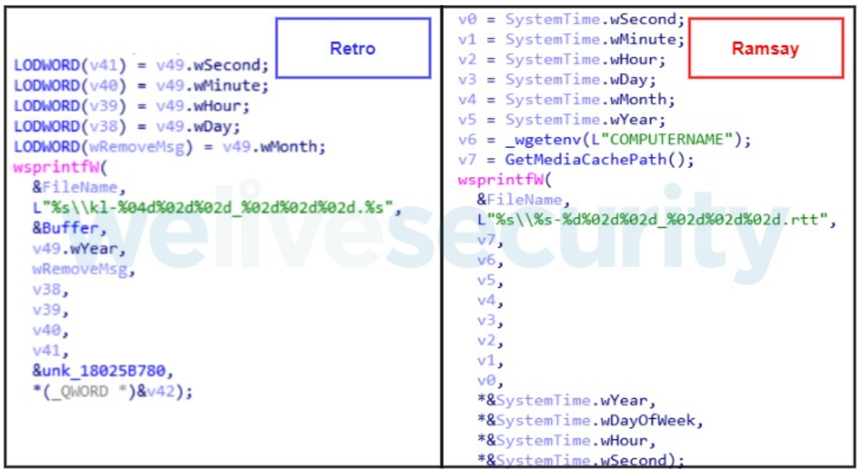
```
v6 = a1[3];
a1[3] = a1[30];
a1[30] = a1[21];
a1[21] = a1[28];
a1[28] = a1[6];
a1[6] = a1[8];
a1[8] = a1[30];
a1[30] = a1[11];
a1[11] = a1[28];
a1[28] = a1[33];
a1[33] = a1[34];
a1[34] = a1[7];
a1[7] = v6;
lpBytes = 0;
v4 = 0;
v5 = 0;
if ( GetBytesFromEthAdapterAddress(&lpBytes, 6u) )
{
  for ( i = 0; i < 6; ++i )
    a1[i + 1] = a0123456789abcd[(*(&lpBytes + i) ^ a1[i + 1]) % 16];
}
return 1;
```

Figure 21. Ramsay and Retro GUID encoding scheme

The GUID retrieved by GetCurrentHwProfile is specific for the system's hardware but not for the user or PC instance. Therefore, it is likely that by just leveraging this GUID operators may encounter duplicates intended to serialize different victims.

The purpose of this scheme is to generate a GUID that is less likely to be duplicate-prone by 'salting' it with the machine's ethernet adapter address. This implies that Retro and Ramsay share the same scheme to generate unique identifiers.

We also found similarities in the way Ramsay and Retro saved some of their log files, sharing a similar filename convention:



```
LODWORD(v41) = v49.wSecond;
LODWORD(v40) = v49.wMinute;
LODWORD(v39) = v49.wHour;
LODWORD(v38) = v49.wDay;
LODWORD(wRemoveMsg) = v49.wMonth;
wsprintfW(
  &FileName,
  L"%s\\kl-%04d%02d%02d_%02d%02d%02d.%s",
  &Buffer,
  v49.wYear,
  wRemoveMsg,
  v38,
  v39,
  v40,
  v41,
  &unk_18025B780,
  *(_QWORD *)&v42);
```

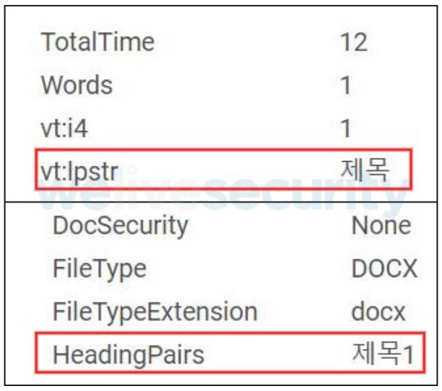
```
v0 = SystemTime.wSecond;
v1 = SystemTime.wMinute;
v2 = SystemTime.wHour;
v3 = SystemTime.wDay;
v4 = SystemTime.wMonth;
v5 = SystemTime.wYear;
v6 = _wgetenv(L"COMPUTERNAME");
v7 = GetMediaCachePath();
wsprintfW(
  &FileName,
  L"%s\\%s-%d%02d%02d_%02d%02d%02d.rtt",
  v7,
  v6,
  v5,
  v4,
  v3,
  v2,
  v1,
  v0,
  *&SystemTime.wYear,
  *&SystemTime.wDayOfWeek,
  *&SystemTime.wHour,
  *&SystemTime.wSecond);
```

Figure 22. Some of Ramsay and Retro filename convention

Is important to highlight that among Retro's documented techniques, it leverages malicious instances of msfte.dll, oci.dll and lame_enc.dll, and via Phantom DLL Hijacking. As previously documented, Ramsay also uses this technique in some of its versions also using msfte.dll and oci.dll.

In addition, we also observed similarities among Ramsay and Retro in regard to the open-source tools used among their toolsets, such as leveraging UACMe for privilege escalation and ImprovedReflectiveDLLInjection for deploying some of their components.

Finally, we noticed Korean language metadata within the malicious documents leveraged by Ramsay, denoting the use of Korean-based templates.



TotalTime	12
Words	1
vt:i4	1
vt:lpstr	제목
DocSecurity	None
FileType	DOCX
FileTypeExtension	docx
HeadingPairs	제목1

Figure 23. Malicious document metadata showing the Korean word "title"

Conclusion

Based on the different instances of the framework found, Ramsay has gone through various development stages, denoting an increasing progression in the number and complexity of its capabilities.

Developers in charge of attack vectors seem to be trying various approaches such as old exploits for Word vulnerabilities from 2017 as well as deploying trojanized applications.

We interpret this as that developers have a prior understanding of the victims' environment and are tailoring attack vectors that would successfully intrude into targeted systems without the need to waste unnecessary resources.

Some stages of Ramsay's framework are still under evaluation, which could explain the current low visibility of victims, having in mind that Ramsay's intended targets may be under air-gapped networks, which would also impact victim visibility.

We will continue monitoring new Ramsay activities and will publish relevant information on our blog. For any inquiries, contact us as *threatintel@eset.com*. Indicators of Compromise can also be found in our GitHub repository.

Indicators of Compromise (IoCs)

SHA-1	ESET detection name	Comments
f79da0d8bb1267f9906fad1111bd929a41b18c03	Win32/TrojanDropper.Agent.SHN	Initial Installer
62d2cc1f6eedba2f35a55beb96cd59a0a6c66880	Win32/Ramsay.A	Installer Launcher
baa20ce99089fc35179802a0cc1149f929bdf0fa	Win32/HackTool.UACMe.T	UAC Bypass Module
5c482bb8623329d4764492ff78b4fbc673b2ef23	Win32/HackTool.UACMe.T	UAC Bypass Module
e7987627200d542bb30d6f2386997f668b8a928c	Win32/TrojanDropper.Agent.SHM	Spreader
3bb205698e89955b4bd07a8a7de3fc75f1cb5cde	Win32/TrojanDropper.Agent.SHN	Malware Installer
bd8d0143ec75ef4c369f341c2786facbd9f73256	Win32/HideProc.M	HideDriver Rootkit
7d85b163d19942bb8d047793ff78ea728da19870	Win32/HideProc.M	HideDriver Rootkit
3849e01bff610d155a3153c897bb662f5527c04c	Win64/HackTool.Inject.A	Darkhotel Retro Backdoor Loader
50eb291fc37fe05f9e55140b98b68d77bd61149e	Win32/Ramsay.B	Ramsay Initial Installer (version 2.b)
87ef7bf00fe6aa928c111c472e2472d2cb047eae	Win32/Exploit.CVE-2017-11882.H	RTF file that drops 50eb291fc37fe05f9e55140b98b68d77bd61149e
5a5738e2ec8af9f5400952be923e55a5780a8c55	Win32/Ramsay.C	Ramsay Agent DLL (32bits)
19bf019fc0bf44828378f008332430a080871274	Win32/Ramsay.C	Ramsay Agent EXE (32bits)
bd97b31998e9d673661ea5697fe436efe026cba1	Win32/Ramsay.C	Ramsay Agent DLL (32bits)
eb69b45faf3be0135f44293bc95f06dad73bc562	Win32/Ramsay.C	Ramsay Agent DLL (32bits)
f74d86b6e9bd105ab65f2af10d60c4074b8044c9	Win64/Ramsay.C	Ramsay Agent DLL (64bits)
ae722a90098d1c95829480e056ef8fd4a98eedd7	Win64/Ramsay.C	Ramsay Agent DLL (64bits)

MITRE ATT&CK techniques

Tactic	ID	Name	Description
Initial Access	T1091	Replication Through Removable Media	Ramsay's spreading mechanism is done via removable drives.
Execution	T1106	Execution through API	Ramsay's embedded components are executed via CreateProcessA and ShellExecute .
	T1129	Execution through Module Load	Ramsay agent can be delivered as a DLL.
	T1203	Exploitation for Client Execution	Ramsay attack vectors exploit CVE-2017-1188 or CVE-2017-0199.
	T1035	Service Execution	Ramsay components can be executed as service dependencies.

Tactic	ID	Name	Description	
	T1204	User Execution	Ramsay Spreader component infects files within the file system.	
Persistence	T1103	AppInit DLLs	Ramsay can use this registry key for persistence.	
	T1050	New Service	Ramsay components can be executed as service dependencies.	
	T1053	Scheduled Task	Ramsay sets a scheduled task to persist after reboot.	
Privilege Escalation	T1088	Bypass User Account Control	Ramsay drops UACMe instances for privilege escalation.	
Defense Evasion	T1038	DLL Order Hijacking	Ramsay agents will masquerade as service dependencies leveraging Phantom DLL Hijacking.	
	T1107	File Deletion	Ramsay installer is deleted after execution.	
	T1055	Process Injection	Ramsay's agent is injected into various processes.	
	T1045	Software Packing	Ramsay installer may be packed with UPX.	
Discovery	T1083	File and Directory Discovery	Ramsay agent scans for files and directories on the system drive.	
	T1057	Process Discovery	Ramsay will attempt to find if host is already compromised by checking the existence of specific processes.	
	Lateral Movement	T1210	Exploitation of Remote Services	Ramsay network scanner may scan the host's subnet to find targets vulnerable to EternalBlue.
T1105		Remote File Copy	Ramsay attempts to infect files on network shares.	
T1091		Replication Through Removable Media	Ramsay attempts to infect files on removable drives.	
Collection	T1119	Automated Collection	Ramsay agent collects files in intervals.	
	T1005	Data from Local System	Ramsay agent scans files on system drive.	
	T1039	Data from Network Shared Drive	Ramsay agent scans files on network shares.	
	T1025	Data from Removable Media	Ramsay agent scans files on removable drives.	
	T1113	Screen Capture	Ramsay agent may generate and collect screenshots.	
Command and Control	T1092	Communication Through Removable Media	Ramsay agent scans for control files for its file-based communication protocol on removable drives.	
	T1094	Custom Command and Control Protocol	Ramsay implements a custom, file-based C&C protocol.	

Tactic	ID	Name	Description
Exfiltration	T1002	Data Compressed	Ramsay agent compresses its collection directory.

Ignacio Sanmillan 13 May 2020 - 11:30AM

Newsletter