SECURITY AND COMMUNICATION NETWORKS
Security Comm. Networks 2015; **8**:4213–4226
Published online 6 September 2015 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/sec.1337

RESEARCH ARTICLE

Data correlation-based analysis methods for automatic memory forensic

X. Fu[1], X. Du[2]* and B. Luo[1]*

[1] Software Institute, Nanjing University, Nanjing, China
[2] Department of Computer and Information Sciences, Temple University, Philadelphia, PA 19122, U.S.A.

ABSTRACT

Memory forensics is an important technique for protecting network security and fighting against computer crimes. It has developed greatly in the past decade, because memory can provide more reliable information that other evidence sources do not contain. However, nowadays, when investigating network criminal cases, the Gigabyte (GB) and even Terabyte (TB) level memory and many such dumps have made memory analysis a difficult task. And investigators usually have to deal with complex operating system (OS) data structures, which they have little knowledge of. So how to analyze memory evidence automatically so as to find the hidden criminal behavior and reconstruct the scenario in an understandable way has become an important problem. This paper presents an automatic memory analysis methodology based on data correlation. Through analyzing key OS data structures and utilizing a clustering algorithm, this methodology can discover the relationships among processes, files, users, Dynamic-link library (DLLs), and network connections. By describing these relationships as correlation graphs, our methods can reorganize these independent memory evidences and disclose their meanings in a high semantic level. Experiments have proved that these correlation graphs can help investigators find hidden criminal behavior and reconstruct the criminal scenarios. And as we know, now, little work is in this field. Copyright © 2015 John Wiley & Sons, Ltd.

KEYWORDS

process correlation; memory forensics; event reconstruction; memory evidences analysis; clustering

***Correspondence**

Xiaojiang Du, Department of Computer and Information Sciences, Temple University, Philadelphia, PA 19122, U. S. A.
E-mail: dxj@ieee.org
Bin Luo, Software Institute, Nanjing University, China.
E-mail: luobin@nju.edu.cn

1. INTRODUCTION

Memory forensics is the subarea of computer forensics, which is a kind of important technique for protecting security of network and fighting against computer crimes. In the past decade, memory forensics has developed greatly because memory can provide information that disks do not contain, such as running processes, network connections, and open ports. Moreover, although the binary codes can be encrypted or obfuscated, all illegal processes still have to be executed in memory and leave some footprints inevitably. So evidence from memory is more reliable. Because of these advantages, memory forensics has absorbed more and more attention recently. However, current work in this field mainly focuses on how to reliably collect evidence, such as memory dumps. Little work is about how to analyze the evidence automatically. Nowadays, with the development of computer hardware, the storage capability of memory has grown from Megabyte (MB) to TB level. Moreover, the widespread use of network and cloud computing has made investigators often face many memory dumps. They usually have to deal with a large quantity of memory data and complex OS data structures, which they have little knowledge of. So how to analyze memory evidence automatically so as to find the hidden criminal behavior and reconstruct the criminal scenario in an understandable way has become an important problem.

This paper presents an automatic memory analysis methodology based on data correlation. Through analyzing key OS data structures and utilizing a clustering algorithm, this methodology can discover the relationships among processes, files, users, DLLs, and network connections. The center of these relationships is the process. These relationships can be divided into two categories, that is, the relationship between process and process and the relationship between process and other memory data. The first kind of relationship includes father–son relation, communication relation, and service relation. And the second kind of relationship includes process–file relation, process–

DLL relation, process–user relation, and process–network relation. By describing these relationships as correlation graphs, our methods can correlate this independent memory evidence and disclose their meanings in a high semantic level. Compared with other memory analysis methods, our methods are the first way that integrates all kinds of data in memory and presents the relationships among them with correlation graphs. Some experiments have proved that these correlation graphs can help investigators find hidden criminal behaviors and reconstruct the criminal scenarios.

2. RELATED WORK

Current memory analysis methods can be divided into four categories: the string searching-based method, the memory scan-based method, the signature-based method, and the key OS data structure-based method.

The string searching-based method is the earliest memory analysis method. It was popular in earlier years because of its simplicity. However, in order to use this method, investigators must already know some keywords of the investigated case, for example, the command name or the process ID. The work of Stevens and Casey [1] is an example of this kind of method. In this paper, the historical records of command lines are searched to analyze the state of the current system. This method is fit for almost all cases and can be used as the complement of other analysis methods, but it needs much manual intervention, and the results often contain much noise.

The memory scan-based method is a brute force searching way [2]. It scans each byte in the whole memory so as to obtain the information required, so the time cost of this method is high and even intolerable in current situations.

The signature-based method has to predefine some signatures by certain rules [3]. And then it will scan the memory based on these signatures to find interesting information. In 2006, Schuster presented several such kinds of methods to analyze processes and networks. The structures of process and thread were used as the scanning signature, and the scanning results were compared with the standard process list to identify the malwares. In addition, they found the allocation of sockets based on the tag of non-paged pool, so that the socket list and network connection list could be obtained [4]. Obviously, its efficiency and accuracy are higher than the first two methods, but it cannot find the unknown behaviors that lack signatures, and its performance also cannot meet the requirement of current big data cases.

The most popular memory analysis method is based on OS data structure, that is, recovering key OS data structures from the memory dump and analyzing them to find interesting evidence. EPROCESS is one of the most important and useful structures in memory. Memparser used its ActiveProcessLinks attribute to obtain the process list [5]. Zhang and Wang [6] used the state information in EPROCESS to create the timeline of forensics. Besides EPROCESS, the registry of Windows is also a valuable database for forensics. Some information about the system could be obtained from registry [7]. In addition, Okolica and Peterson [8] presented a method to obtain the list of socket and network connection by analyzing the tcpip.sys. Ionescu *et al.* [9] presented a method to identify the DLL injection attack. This is implemented by traversing the doubly linked list of DLLs in Process Environment Block. Dolan-Gavitt [10] and Van Baar and Alink [11] obtained the DLLs and files based on Virtual Address Descriptors (VAD). In short, analysis methods based on OS data structures are accurate and widely used. However, it depends on the data structures of OS. If the structure changes in different versions of OS, this kind of method should also be changed.

The method in this paper is also based on OS data structures. However, compared with other memory analysis methods, our method is the first way that integrates all kinds of data in memory and presents the relationships among them with correlation graphs. Current methods usually aim at one kind of data structure. This kind of analysis is isolated and depends on much manual intervention, and the illegal behavior identification and the event reconstruction are still completed manually by investigators. Moreover, no method focuses on the relations between memory data. Even Volatility [12], the famous forensic framework, only provides information to investigators. Our methods can discover the relationships among processes, files, users, DLLs, and network connections; and these relationships are represented as correlation graphs, which are easy to understand. Our experiments have proved that these correlation graphs can help investigators find hidden criminal behaviors and reconstruct the criminal scenarios.

3. BACKGROUND AND OVERVIEW

3.1. Motivation

Evidence analysis is one of the key phrases in forensics. The main work in this phrase includes identifying illegal behaviors from normal ones and reconstructing the criminal process. However, with the growth of storage capability and the widespread use of network and cloud computing, current investigators often have to simultaneously face many memory dumps that have TB level data and complex OS data structure. Although some forensic tools such as Volatility can recover most memory data from memory dumps, these data are isolated. Investigators still have to identify criminal evidences and correlate them manually. In fact, if the relationships among memory data can be found and represented automatically in the early stage of analysis, not only event reconstruction but also criminal behavior identification will become easier than before. That is due to the abnormal relationship, such as outlook reading a password file, that usually implies an illegal behavior. In addition, once one piece of criminal evidence is obtained, investigators can find more evidence following the relationships among them. Then the criminal behavior will be more difficult to hide.

DOI: 10.1002/sec

3.2. Challenge

In order to find the relations among different kinds of information, some challenges have to be solved.

(1) There is a larger quantity of data and relationships in memory, how to select the useful ones?

As the core of computer systems, memory usually contains many kinds of information, for example, the key data and code of operation system, drivers, and applications. Moreover, the type, the structure, the location, and the semantics of this data are often unknown for investigators and will even change dynamically. The relationships among this data are also diverse. All these will make the analysis task more difficult.

(2) How to disclose these relationships reliably?

Even when we know there is a certain relationship between data (e.g., father–son relation between processes), how to disclose this relationship reliably and automatically is still a difficult task, because criminals could tamper with the key data, which indicate this relation so as to hide behaviors. Direct kernel object manipulation is one such example. Hackers tamper the link of process list, and then the illegal process will not be shown.

(3) How to deal with the implicit relationship?

Some relationships, for example, several processes serving for the same application, cannot be indicated by memory data. However, this kind of relationship is very important for forensics. So how to find these implicit relationships is also a challenge.

3.3. Methods overview

In computer forensics, the most important information in memory is process. It represents the software and the services running in the system. So considering challenge 1, we choose a process-centric strategy for analyzing memory evidence. The relationships can be dived into two categories, that is, the relationship between process and process and the relationship between process and other memory data. In order to show the first category of relationship from different aspects, this paper presents some methods of finding the father–son relation, the communication relation, and the service relation between processes. For the second category, this paper mainly analyzes the process–file relation, the process–DLL relation, the process–user relation, and the process–network relation. These relationships are chosen because they are useful for computer forensics. Considering challenges 2 and 3, we have designed a set of correlation methods based on both OS data structures and the clustering algorithm. For the relationships that have indicators in memory (usually certain OS data structures), we will disclose them by analyzing these data structures and choose the most reliable data structures as the indicator. As for the implicit relationships, we will discover them using the clustering algorithm. Now the most common operating system that the investigators met is still Windows XP, so our implementation and experiments are both based on Windows XP SP2.

4. CORRELATION METHODS FOR PROCESS AND OTHER MEMORY DATA

4.1. Correlating process and files

Current memory forensic methods are able to find and recover the files in memory, but they cannot correlate these files to processes. However, in order to reconstruct the criminal scenario, investigators often need to know which process creates, reads, or writes certain files. So this paper presents an _OBJECT_HEADER-based method for correlating processes and files.

In Window's memory, each file object is represented by a _FILE_OBJECT. It is an important data structure that contains much file-related information, for example, the name of file. However, it does not contain which process owns this file. Fortunately, both process and file exist as objects in memory, and every memory object has an _OBJECT_HEADER (Figure 1). We can obtain the required information from this structure. The size of _OBJECT_HEADER is always 0x018. In other words, if we look back 0x18 from the base address of each memory object, we can obtain its _OBJECT_HEADER. There is an attribute named HandleInfoOffset in _OBJECT_HEADER. It stores the offset of this file's handle info. According to this offset and the base address of each memory object, we can obtain the handle info that is represented by two words. As for file object, the first word is the address of its owner process. So we can find the relation between file and process based on this address.

For example, we can create a test file named test.docx and look for its owner process. Using the method mentioned earlier, we obtained _FILE_OBJECT and _OBJECT_HEADER of test.docx as in Figure 2. According to the value of HandleInfoOffset in _OBJECT_HEADER, we knew that the handle info was in 0x8 bytes before _OBJECT_HEADER. After reading the first word of handle

```
lkd> dt _OBJECT_HEADER
nt!_OBJECT_HEADER
   +0x000 PointerCount     : Int4B
   +0x004 HandleCount      : Int4B
   +0x004 NextToFree       : Ptr32 Void
   +0x008 Type             : Ptr32 _OBJECT_TYPE
   +0x00c NameInfoOffset   : UChar
   +0x00d HandleInfoOffset : UChar
   +0x00e QuotaInfoOffset  : UChar
   +0x00f Flags            : UChar
   +0x010 ObjectCreateInfo : Ptr32 _OBJECT_CREATE_INFORMATION
   +0x010 QuotaBlockCharged : Ptr32 Void
   +0x014 SecurityDescriptor : Ptr32 Void
   +0x018 Body             : _QUAD
```

Figure 1. _OBJECT_HEADER.

```
lkd> dt nt!_FILE_OBJECT 81bb2f90
   +0x000 Type             : 5
   +0x002 Size             : 112
   +0x004 DeviceObject     : 0x82048900 _DEVICE_OBJECT
   +0x008 Vpb              : 0x81cac0f0 _VPB
   +0x00c FsContext        : 0xe1bd20d0
   +0x010 FsContext2       : 0xe1bd2228
   +0x014 SectionObjectPointer : 0x81fd4844 _SECTION_OBJECT_POINTERS
   +0x018 PrivateCacheMap  : (null)
   +0x01c FinalStatus      : 0
   +0x020 RelatedFileObject : (null)
   +0x024 LockOperation    : 0 ''
   +0x025 DeletePending    : 0 ''
   +0x026 ReadAccess       : 0x1 ''
   +0x027 WriteAccess      : 0x1 ''
   +0x028 DeleteAccess     : 0 ''
   +0x029 SharedRead       : 0x1 ''
   +0x02a SharedWrite      : 0 ''
   +0x02b SharedDelete     : 0 ''
   +0x02c Flags            : 0x40042
   +0x030 FileName         : _UNICODE_STRING "\Documents and Settings\a\桌面\experiment\test.docx"
   +0x038 CurrentByteOffset : _LARGE_INTEGER 0x0
   +0x040 Waiters          : 0
   +0x044 Busy             : 0
   +0x048 LastLock         : (null)
   +0x04c Lock             : _KEVENT
   +0x05c Event            : _KEVENT
   +0x06c CompletionContext : (null)
lkd> dt nt!_OBJECT_HEADER 81bb2f90-0x018
   +0x000 PointerCount     : 1
   +0x004 HandleCount      : 1
   +0x004 NextToFree       : 0x00000001
   +0x008 Type             : 0x821ebe70 _OBJECT_TYPE
   +0x00c NameInfoOffset   : 0 ''
   +0x00d HandleInfoOffset : 0x8 ''
   +0x00e QuotaInfoOffset  : 0 ''
   +0x00f Flags            : 0x40 '@'
   +0x010 ObjectCreateInfo : 0x81d73e30 _OBJECT_CREATE_INFORMATION
   +0x010 QuotaBlockCharged : 0x81d73e30
   +0x014 SecurityDescriptor : (null)
   +0x018 Body             : _QUAD
```

Figure 2. _FILE_OBJECT and _OBJECT_HEADER of test.docx.

info by little endian, we obtained an object address, that is, 81bc4020. Then we read this address and obtained a process named winword (Figure 3), so we know that the owner process of test.docx is winword.

4.2. Correlating process and DLLs

In Windows, system application programming interface (APIs) are usually implemented as DLLs. So we can infer the function of a process by analyzing what DLLs it loads, which is helpful for investigating malwares in memory.

The DLLs that a process loads are also one kind of handle. So they can be found with _HANDLE_TABLE (Figure 4) of a process. _HANDLE_TABLE has an attribute named HandleTableList that points to a doubly linked list of handles. This table can be traversed by two pointers, that is, Flink and Blink. There are many types of handles, for example, files, processes, registry keys, and named pipes. In order to find DLLs, we should traverse the handle table and look for the one whose type is file and whose name ends with DLL. These handles are just DLLs the process loads. Because Volatility already has this function, we use it in our experiment directly.

```
lkd> db 81bb2f90-0x018-8 L 8
81bb2f70  20 40 bc 81 01 00 00 00                          @.....
lkd> !object 81bc4020
Object: 81bc4020  Type: (821b9e70) Process
    ObjectHeader: 81bc4008 (old version)
    HandleCount: 3  PointerCount: 90
lkd> !process 81bc4020
PROCESS 81bc4020  SessionId: 0  Cid: 08f8    Peb: 7ffdd000  ParentCid: 0268
    DirBase: 0a3c0400  ObjectTable: e1510580  HandleCount: 478.
    Image: WINWORD.EXE
```

Figure 3. The value of handle info and the object it points to.

```
lkd> dt _HANDLE_TABLE
nt!_HANDLE_TABLE
   +0x000 TableCode        : Uint4B
   +0x004 QuotaProcess     : Ptr32 _EPROCESS
   +0x008 UniqueProcessId  : Ptr32 Void
   +0x00c HandleTableLock  : [4] _EX_PUSH_LOCK
   +0x01c HandleTableList  : _LIST_ENTRY
   +0x024 HandleContentionEvent : _EX_PUSH_LOCK
   +0x028 DebugInfo        : Ptr32 _HANDLE_TRACE_DEBUG_INFO
   +0x02c ExtraInfoPages   : Int4B
   +0x030 FirstFree        : Uint4B
   +0x034 LastFree         : Uint4B
   +0x038 NextHandleNeedingPool : Uint4B
   +0x03c HandleCount      : Int4B
   +0x040 Flags            : Uint4B
   +0x040 StrictFIFO       : Pos 0, 1 Bit
```

Figure 4. _HANDLE_TABLE.

4.3. Correlating process and users

Knowing who created certain processes is important for reconstructing the criminal scenario, so it is necessary to correlate users with processes. Current memory forensic methods can only obtain the Security ID (SID) of users. They convert the SID into users' names. But the latter is more useful for forensics, and there is also no information in memory about which user an SID maps into. Fortunately, some clues can be found in register. So this paper presents a correlation method based on register.

A process is described by _EPROCESS in memory (Figure 5). _EPROCESS is one of the most important data structures in memory. It contains an attribute named Token that points to a _TOKEN structure (Figure 6). Much security-related information is contained in this structure, including the SID of users and groups. So we can obtain the SID of users that own this process using this structure.

In Windows, each user has a unique SID. However, according to the information in _TOKEN, we found that many SIDs map to a process, and most of these SIDs are the universal identifiers of system, which exist in almost every process. So they should be filtered at first, and the remainders are the SIDs of users. These universal identifiers are recorded in Window's document on system universal identifiers, so they can be filtered based on this document. After obtaining the SIDs of users, we could look for the username of these SIDs in registry. HKLM\SOFTWARE

```
lkd> dt _EPROCESS
nt!_EPROCESS
   +0x000 Pcb              : _KPROCESS
   +0x06c ProcessLock      : _EX_PUSH_LOCK
   +0x070 CreateTime       : _LARGE_INTEGER
   +0x078 ExitTime         : _LARGE_INTEGER
   +0x080 RundownProtect   : _EX_RUNDOWN_REF
   +0x084 UniqueProcessId  : Ptr32 Void
   +0x088 ActiveProcessLinks : _LIST_ENTRY
   +0x090 QuotaUsage       : [3] Uint4B
   +0x09c QuotaPeak        : [3] Uint4B
   +0x0a8 CommitCharge     : Uint4B
   +0x0ac PeakVirtualSize  : Uint4B
   +0x0b0 VirtualSize      : Uint4B
   +0x0b4 SessionProcessLinks : _LIST_ENTRY
   +0x0bc DebugPort        : Ptr32 Void
   +0x0c0 ExceptionPort    : Ptr32 Void
   +0x0c4 ObjectTable      : Ptr32 _HANDLE_TABLE
   +0x0c8 Token            : _EX_FAST_REF
   +0x0cc WorkingSetLock   : _FAST_MUTEX
   +0x0ec WorkingSetPage   : Uint4B
   +0x0f0 AddressCreationLock : _FAST_MUTEX
   +0x110 HyperSpaceLock   : Uint4B
   +0x114 ForkInProgress   : Ptr32 _ETHREAD
   +0x118 HardwareTrigger  : Uint4B
   +0x11c VadRoot          : Ptr32 Void
   +0x120 VadHint          : Ptr32 Void
   +0x124 CloneRoot        : Ptr32 Void
   +0x128 NumberOfPrivatePages : Uint4B
   +0x12c NumberOfLockedPages : Uint4B
   +0x130 Win32Process     : Ptr32 Void
   +0x134 Job              : Ptr32 _EJOB
   +0x138 SectionObject    : Ptr32 Void
   +0x13c SectionBaseAddress : Ptr32 Void
   +0x140 QuotaBlock       : Ptr32 _EPROCESS_QUOTA_BLOCK
   +0x144 WorkingSetWatch  : Ptr32 _PAGEFAULT_HISTORY
   +0x148 Win32WindowStation : Ptr32 Void
   +0x14c InheritedFromUniqueProcessId : Ptr32 Void
   +0x150 LdtInformation   : Ptr32 Void
   +0x154 VadFreeHint      : Ptr32 Void
   +0x158 VdmObjects       : Ptr32 Void
   +0x15c DeviceMap        : Ptr32 Void
   +0x160 PhysicalVadList  : _LIST_ENTRY
   +0x168 PageDirectoryPte : _HARDWARE_PTE
   +0x168 Filler           : Uint8B
```

Figure 5. _EPROCESS.

```
lkd> dt _TOKEN
nt!_TOKEN
   +0x000 TokenSource      : _TOKEN_SOURCE
   +0x010 TokenId          : _LUID
   +0x018 AuthenticationId : _LUID
   +0x020 ParentTokenId    : _LUID
   +0x028 ExpirationTime   : _LARGE_INTEGER
   +0x030 TokenLock        : Ptr32 _ERESOURCE
   +0x038 AuditPolicy      : _SEP_AUDIT_POLICY
   +0x040 ModifiedId       : _LUID
   +0x048 SessionId        : Uint4B
   +0x04c UserAndGroupCount : Uint4B
   +0x050 RestrictedSidCount : Uint4B
   +0x054 PrivilegeCount   : Uint4B
   +0x058 VariableLength   : Uint4B
   +0x05c DynamicCharged   : Uint4B
   +0x060 DynamicAvailable : Uint4B
   +0x064 DefaultOwnerIndex : Uint4B
   +0x068 UserAndGroups    : Ptr32 _SID_AND_ATTRIBUTES
   +0x06c RestrictedSids   : Ptr32 _SID_AND_ATTRIBUTES
   +0x070 PrimaryGroup     : Ptr32 Void
   +0x074 Privileges       : Ptr32 _LUID_AND_ATTRIBUTES
   +0x078 DynamicPart      : Ptr32 Uint4B
   +0x07c DefaultDacl      : Ptr32 _ACL
   +0x080 TokenType        : _TOKEN_TYPE
   +0x084 ImpersonationLevel : _SECURITY_IMPERSONATION_LEVEL
   +0x088 TokenFlags       : Uint4B
   +0x08c TokenInUse       : UChar
   +0x090 ProxyData        : Ptr32 _SECURITY_TOKEN_PROXY_DATA
   +0x094 AuditData        : Ptr32 _SECURITY_TOKEN_AUDIT_DATA
   +0x098 OriginatingLogonSession : _LUID
   +0x0a0 VariablePart     : Uint4B
```

Figure 6. _TOKEN.

\Microsoft\ WindowsNT\CurrentVersion\ProfileList records all SIDs in this system. If we search the ProfileImagePath attribute of users' SIDs in the ProfileList, we can extract the username. So based on the aforementioned method, the user can be correlated to the process.

For example, we can obtain the SIDs of some processes as in Figure 7. Among these SIDs, only S-1-5-21-2052111302-1085031214-682003330-1003 is not the universal identifier. When we look for this SID in the ProfileList of registry, the result is as Figures 8 and 9. From the ProfileImagePath, we can know that the username is a. So user a can be correlated with all processes that have the SID S-1-5-21-2052111302-1085031214-682003330-1003.

4.4. Correlating process and network

Criminals often need to transmit information and instructions through a network. So knowing which process is related to certain network information is important for investigators. Based on this relationship, investigators can know the IP of the host that communicates with a certain process and even obtain the content of the communication.

Windows uses _TCPT_OBJECT (Figure 10) to describe a TCP connection. _TCPT_OBJECT contains plenty of network-related information, such as the pointer to next TCP object, the destination IP of the TCP connection, the local IP, the remote port, the local port, and the Pid of the process that creates this connection. So we can correlate the process and TCP connection based on Pid. _TCPT_OBJECT has a special tag TCPT (0x54435054), so we can locate this data structures with a signature-based scanning method. In addition, the next TCP object can be obtained following the next attribute in _TCPT_OBJECT. So we can find all the TCP connections and correlate them according to processes. Because Volatility already has this function, so we use it in our experiment.

```
ThunderPlatform (2608): S-1-5-21-2052111302-1085031214-682003330-1003
ThunderPlatform (2608): S-1-5-21-2052111302-1085031214-682003330-513 (Domain Users)
ThunderPlatform (2608): S-1-1-0 (Everyone)
ThunderPlatform (2608): S-1-5-32-544 (Administrators)
ThunderPlatform (2608): S-1-5-32-545 (Users)
ThunderPlatform (2608): S-1-5-4 (Interactive)
ThunderPlatform (2608): S-1-5-11 (Authenticated Users)
ThunderPlatform (2608): S-1-5-5-0-63753 (Logon Session)
ThunderPlatform (2608): S-1-2-0 (Local (Users with the ability to log in locally))
XLLiveUD.exe (4036): S-1-5-21-2052111302-1085031214-682003330-1003
XLLiveUD.exe (4036): S-1-5-21-2052111302-1085031214-682003330-513 (Domain Users)
XLLiveUD.exe (4036): S-1-1-0 (Everyone)
XLLiveUD.exe (4036): S-1-5-32-544 (Administrators)
XLLiveUD.exe (4036): S-1-5-32-545 (Users)
XLLiveUD.exe (4036): S-1-5-4 (Interactive)
XLLiveUD.exe (4036): S-1-5-11 (Authenticated Users)
XLLiveUD.exe (4036): S-1-5-5-0-63753 (Logon Session)
XLLiveUD.exe (4036): S-1-2-0 (Local (Users with the ability to log in locally))
WINWORD.EXE (2296): S-1-5-21-2052111302-1085031214-682003330-1003
WINWORD.EXE (2296): S-1-5-21-2052111302-1085031214-682003330-513 (Domain Users)
WINWORD.EXE (2296): S-1-1-0 (Everyone)
WINWORD.EXE (2296): S-1-5-32-544 (Administrators)
WINWORD.EXE (2296): S-1-5-32-545 (Users)
WINWORD.EXE (2296): S-1-5-4 (Interactive)
WINWORD.EXE (2296): S-1-5-11 (Authenticated Users)
WINWORD.EXE (2296): S-1-5-5-0-63753 (Logon Session)
WINWORD.EXE (2296): S-1-2-0 (Local (Users with the ability to log in locally))
IEXPLORE.EXE (3268): S-1-5-21-2052111302-1085031214-682003330-1003
IEXPLORE.EXE (3268): S-1-5-21-2052111302-1085031214-682003330-513 (Domain Users)
IEXPLORE.EXE (3268): S-1-1-0 (Everyone)
IEXPLORE.EXE (3268): S-1-5-32-544 (Administrators)
IEXPLORE.EXE (3268): S-1-5-32-545 (Users)
IEXPLORE.EXE (3268): S-1-5-4 (Interactive)
IEXPLORE.EXE (3268): S-1-5-11 (Authenticated Users)
IEXPLORE.EXE (3268): S-1-5-5-0-63753 (Logon Session)
IEXPLORE.EXE (3268): S-1-2-0 (Local (Users with the ability to log in locally))
QQExternal.exe (1928): S-1-5-21-2052111302-1085031214-682003330-1003
QQExternal.exe (1928): S-1-5-21-2052111302-1085031214-682003330-513 (Domain Users)
QQExternal.exe (1928): S-1-1-0 (Everyone)
QQExternal.exe (1928): S-1-5-32-544 (Administrators)
QQExternal.exe (1928): S-1-5-32-545 (Users)
QQExternal.exe (1928): S-1-5-4 (Interactive)
```

Figure 7. The SIDs of some processes.

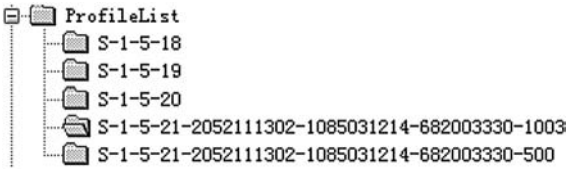

Figure 8. The SIDs in ProfileList.

5. CORRELATION METHODS FOR PROCESS AND PROCESS

5.1. Communication relation between processes

The communication relation between processes is an important relationship for reconstructing a criminal scenario. In Windows, processes are able to communicate in different ways. This paper chooses three most popular communication methods to correlate processes, that is, shared memory, file mapping, and pipeline.

5.1.1. Correlating processes by shared memory.

Shared memory means certain memory blocks exist in the virtual address space of several processes, so they can communicate by sharing these blocks. As we know, although this relation is useful for forensics, there are still no memory forensic tools that consider it. This is probably due to _EPROCESS having no knowledge of this relation, and other data structures also cannot indicate it directly. In this paper, we present a way to identify this relation. It mainly includes three steps.

名称	类型	数据
(默认)	REG_SZ	(数值未设置)
CentralProfile	REG_SZ	
Flags	REG_DWORD	0x00000000 (0)
OptimizedLogo...	REG_DWORD	0x0000000b (11)
ProfileImagePath	REG_EXPAND_SZ	%SystemDrive%\Documents and Settings\a
ProfileLoadTi...	REG_DWORD	0x01cf577f (30365567)
ProfileLoadTi...	REG_DWORD	0x7229dfe6 (1915346918)
RefCount	REG_DWORD	0x00000017 (23)
RunLogonScrip...	REG_DWORD	0x00000000 (0)
Sid	REG_BINARY	01 05 00 00 00 00 00 05 15 00 00 00 c6 bb 50 7a
State	REG_DWORD	0x00000100 (256)

Figure 9. THE detailed information of S-1-5-21-2052111302-1085031214-682003330-1003.

```
'_TCPT_OBJECT' : [ 0x20, { \
    'Next' : [ 0x0, ['pointer', ['_TCPT_OBJECT']]], \
    'RemoteIpAddress' : [ 0xc, ['unsigned long']], \
    'LocalIpAddress' : [ 0x10, ['unsigned long']], \
    'RemotePort' : [ 0x14, ['unsigned short']], \
    'LocalPort' : [ 0x16, ['unsigned short']], \
    'Pid' : [ 0x18, ['unsigned long']], \
} ], \
```

Figure 10. _TCPT_OBJECT.

Step 1. Obtaining the VAD tree of the process being investigated. _EPROCESS has an attribute named VadRoot that points to the address of VAD tree's root node. Every node in VAD tree is an _MMVAD structure (Figure 11), which describes one block of virtual address space assigned to this process. The StartingVpn and EndingVpn attributes are the starting and ending addresses of this block. The LeftChild and RightChild attributes point to the left and right child nodes of the current node. So the VAD tree can be built using these pointers. It should be noted that every VAD node is either private or mapped. A private VAD node means that the virtual address space this node describes is private to the process. A mapped VAD node means that the virtual address space this node describes is shared with other processes, so when we correlate processes by shared memory, we only consider this kind of VAD node.

Step 2. Obtaining the shared memory block by the File Pointer attribute of VAD node. From Figure 11, we can find that _MMVAD has a ControlArea attribute that points to a _CONTROL_AREA structure. _CONTROL_AREA (Figure 12) describes the control information of this VAD node. It includes the pointers to files, shared memory, buffered data, and so on. The FilePointer attribute in _CONTROL_AREA is a pointer to _FILE_OBJECT. After observing all the VAD nodes a process contains, we find that the value of FilePointer falls into two categories. Firstly, it is a normal hex address in most cases. That means it points to a file that can be obtained with this address. In rare cases, FilePointer is null. According to the Windows official documents, that means _CONTROL_AREA describes a block of shared memory, and this block is used to communicate between processes. In other words, if we find that the _FILE_OBJECT attribute of the VAD node points to a block of shared memory, we can infer that the owner process of this VAD node is communicating or waiting to communicate with other processes with this shared memory.

```
lkd> dt _MMVAD
nt!_MMVAD
   +0x000 StartingVpn      : Uint4B
   +0x004 EndingVpn        : Uint4B
   +0x008 Parent           : Ptr32 _MMVAD
   +0x00c LeftChild        : Ptr32 _MMVAD
   +0x010 RightChild       : Ptr32 _MMVAD
   +0x014 u                : __unnamed
   +0x018 ControlArea      : Ptr32 _CONTROL_AREA
   +0x01c FirstPrototypePte : Ptr32 _MMPTE
   +0x020 LastContiguousPte : Ptr32 _MMPTE
   +0x024 u2               : __unnamed
```

Figure 11. _MMVAD.

```
lkd> dt _CONTROL_AREA
nt!_CONTROL_AREA
   +0x000 Segment          : Ptr32 _SEGMENT
   +0x004 DereferenceList  : _LIST_ENTRY
   +0x00c NumberOfSectionReferences : Uint4B
   +0x010 NumberOfPfnReferences : Uint4B
   +0x014 NumberOfMappedViews : Uint4B
   +0x018 NumberOfSubsections : Uint2B
   +0x01a FlushInProgressCount : Uint2B
   +0x01c NumberOfUserReferences : Uint4B
   +0x020 u                : __unnamed
   +0x024 FilePointer      : Ptr32 _FILE_OBJECT
   +0x028 WaitingForDeletion : Ptr32 _EVENT_COUNTER
   +0x02c ModifiedWriteCount : Uint2B
   +0x02e NumberOfSystemCacheViews : Uint2B
```

Figure 12. _CONTROL_AREA.

Step 3. Correlating processes based on _SEGMENT attribute. From Figure 12, we can find that _CONTROL_AREA has a Segment attribute that points to a _SEGMENT structure (Figure 13). _SEGMENT has two attributes named u1 and u2. We find that if FilePointer is null, then u1 points to the process that created this shared memory. We also observe that the values of u1 can be divided into two kinds. (i) It is equal to the address of the VAD node's owner process, which means that this process is the creator of this shared memory. (ii) It is different from the address of the VAD node's owner process. This means that it points to a process that creates this shared memory and communicates with the VAD node's owner process with this shared memory. So we can correlate the two processes.

For example, we want to test which processes the process Word communicates with by shared memory. Firstly, we obtain the VAD node of the process Word (Figure 14).

Then for all mapped VAD nodes, we choose the one whose address is 820e1c80 and obtain its _MMVAD and

```
nt!_SEGMENT
   +0x000 ControlArea      : Ptr32 _CONTROL_AREA
   +0x004 TotalNumberOfPtes : Uint4B
   +0x008 NonExtendedPtes  : Uint4B
   +0x00c WritableUserReferences : Uint4B
   +0x010 SizeOfSegment    : Uint8B
   +0x018 SegmentPteTemplate : _MMPTE
   +0x020 NumberOfCommittedPages : Uint4B
   +0x024 ExtendInfo       : Ptr32 _MMEXTEND_INFO
   +0x028 SystemImageBase  : Ptr32 Void
   +0x02c BasedAddress     : Ptr32 Void
   +0x030 u1               : __unnamed
   +0x034 u2               : __unnamed
   +0x038 PrototypePte     : Ptr32 _MMPTE
   +0x040 ThePtes          : [1] _MMPTE
```

Figure 13. _SEGMENT.

```
lkd> !vad 8206bcf0
VAD     level      start      end    commit
81c8bc48 (19)         10       10         1 Private      READWRITE
820a3eb8 (20)         20       20         1 Private      READWRITE
81f524e0 (18)         30      12f        30 Private      READWRITE
820e1c80 (20)        130      132         0 Mapped       READONLY
820a7348 (19)        140      140         0 Mapped       READONLY
81d8dea0 (20)        150      24f       199 Private      READWRITE
81b66ce0 (17)        250      25f         7 Private      READWRITE
8212bbc8 (20)        260      26f         0 Mapped       READWRITE
81b96138 (19)        270      285         0 Mapped       READONLY
81b9e6a0 (20)        290      2cc         0 Mapped       READONLY
81f51b60 (18)        2d0      310         0 Mapped       READONLY
81ef9550 (20)        320      325         0 Mapped       READONLY
820c3fd8 (19)        330      370         0 Mapped       READONLY
81d70038 (16)        380      38f         9 Private      READWRITE
81a0e240 (19)        390      392         0 Mapped       READONLY
81cd8270 (18)        3a0      3af        16 Private      READWRITE
81c87a80 (19)        3b0      3b1         0 Mapped       READONLY
81a0f248 (17)        3c0      3c1         0 Mapped       READONLY
81f22288 (19)        3d0      497         0 Mapped       EXECUTE_REA
820c4a68 (18)        4a0      5a2         0 Mapped       READONLY
820a9f30 (19)        5b0      5bf        16 Private      READWRITE
81f3ecc8 (15)        5c0      8bf         0 Mapped       EXECUTE_REA
81e684d0 (18)        8c0      8c0         1 Private      READWRITE
81fe9c18 (17)        8d0      8d0         1 Private      READWRITE
81b9e1b8 (18)        8e0      8ef        16 Private      READWRITE
82083be8 (16)        8f0      91a        43 Private      READWRITE
81f9a158 (18)        920      921         0 Mapped       READONLY
```

Figure 14. The VAD node of process word.

_CONTROL_AREA based on the aforementioned method. The result is shown in Figure 15. We find that the value of FilePointer is null, which means that this node is a block of shared memory.

Then we check the u1 attribute in _SEGMENT. The result is shown in Figure 16. We obtain the object according to the value of u1 and find that the type of this object is a process, and the name of this process is csrss. The detailed information is shown in Figures 17 and 18.

After checking all mapped VAD nodes of Word, we can obtain all the processes that communicate with Word by shared memory. They can be presented using the correlation graph in Figure 19.

5.1.2. Correlating processes by file mapping.

File mapping communication means that a process regards a file as memory blocks in its address space, so it can read and write the contents of this file by simple pointer operations instead of complex file Input/Output (I/O) operations. Windows allows several processes to access the same file mapping object. Each process can

```
lkd> dt nt!_MMVAD 820e1c80
   +0x000 StartingVpn      : 0x130
   +0x004 EndingVpn        : 0x132
   +0x008 Parent           : 0x820a7348 _MMVAD
   +0x00c LeftChild        : (null)
   +0x010 RightChild       : (null)
   +0x014 u                : __unnamed
   +0x018 ControlArea      : 0x81d73d88 _CONTROL_AREA
   +0x01c FirstPrototypePte : 0xe1b01e20 _MMPTE
   +0x020 LastContiguousPte : 0xe1b01e30 _MMPTE
   +0x024 u2               : __unnamed
lkd> dt nt!_CONTROL_AREA 0x81d73d88
   +0x000 Segment          : 0xe1b01de0 _SEGMENT
   +0x004 DereferenceList  : _LIST_ENTRY [ 0x0 - 0x0 ]
   +0x00c NumberOfSectionReferences : 1
   +0x010 NumberOfPfnReferences : 0
   +0x014 NumberOfMappedViews : 0x11
   +0x018 NumberOfSubsections : 1
   +0x01a FlushInProgressCount : 0
   +0x01c NumberOfUserReferences : 0x12
   +0x020 u                : __unnamed
   +0x024 FilePointer      : (null)
   +0x028 WaitingForDeletion : (null)
   +0x02c ModifiedWriteCount : 0
   +0x02e NumberOfSystemCacheViews : 0
```

Figure 15. _MMVAD and _CONTROL_AREA of selected VAD node.

```
lkd> dt nt!_SEGMENT 0xe1b01de0
   +0x000 ControlArea      : 0x81d73d88 _CONTROL_AREA
   +0x004 TotalNumberOfPtes : 3
   +0x008 NonExtendedPtes  : 3
   +0x00c WritableUserReferences : 0
   +0x010 SizeOfSegment    : 0x3000
   +0x018 SegmentPteTemplate : _MMPTE
   +0x020 NumberOfCommittedPages : 3
   +0x024 ExtendInfo       : (null)
   +0x028 SystemImageBase  : (null)
   +0x02c BasedAddress     : (null)
   +0x030 u1               : __unnamed
   +0x034 u2               : __unnamed
   +0x038 PrototypePte     : 0xe1b01e20 _MMPTE
   +0x040 ThePtes          : [1] _MMPTE
lkd> dd 0xe1b01de0 + 0x030
e1b01e10  81b142c0 01760000 e1b01e20 00000000
e1b01e20  181bd163 80000000 181be163 80000000
e1b01e30  17794123 80000000 0001040c 61626453
e1b01e40  1c060401 8208d9a8 e1624cf8 e1624cf8
e1b01e50  e25c3ea8 e16c4748 e1b01e58 e1b01e58
e1b01e60  00000000 00010002 81c3dc80 00000000
e1b01e70  00010406 6d695346 0c070401 61564d43
e1b01e80  002c0000 00166b76 80000004 00000000
```

Figure 16. _SEGMENT and the value of u1.

```
lkd> !object 81b142c0
Object: 81b142c0  Type: (821b9e70) Process
    ObjectHeader: 81b142a8 (old version)
    HandleCount: 3  PointerCount: 205
```

Figure 17. The object type in address 81b142c0.

receive a pointer in its address space and use this pointer to read or write the same file. By this way, several processes can communicate with each other. As far as we know, there is no memory forensic tool considering this kind of communication. In order to identify this communication, this paper presents a correlating method based on the process–file relation we mentioned in Section 4.1. Firstly, the processes and files in memory are correlated by the method in Section 4.1. Secondly, if several processes are correlated with the same files, there should be file mapping-based communication between them.

5.1.3. Correlating processes by named pipelines.

Pipeline is a communication channel that has two ends. The processes holding the handle of two ends can communicate through this channel. There are two types of pipelines, that is, named pipelines and anonymous pipelines. The latter has no name and cannot be obtained from the memory, so this paper only focuses on named pipelines. We can obtain the pipeline based on the handle information of the process and then correlate them. We have not found other forensic methods that consider this relation.

A name is given to the named pipeline after it is created. Then any process using this pipeline could open it using this name to obtain the handle. So at first, we obtain the handle list of each process and then obtain the name of the pipeline in this list. The handle list of each process is described by _HANDLE_TABLE, which is pointed to by the ObjectTable attribute in _EPROCESS. The correlating method is similar to the method in Section 4.2. Firstly, the handle list of each process is obtained and traversed. All handles of type named pipeline will be filtered. Then based on the information of these handles, we can extract the name of the pipeline. Finally, the processes that use the same named pipeline can be found and correlated.

```
lkd> !process 81b142c0
PROCESS 81b142c0  SessionId: 0  Cid: 0270    Peb: 7ffdf000  ParentCid: 0228
    DirBase: 0a3c0040  ObjectTable: e1645858  HandleCount: 545.
    Image: csrss.exe
    VadRoot 81f521a8 Vads 159 Clone 0 Private 408. Modified 11344. Locked 0.
    DeviceMap e1004428
    Token                             e18d4908
    ElapsedTime                       06:48:05.479
    UserTime                          00:00:02.484
    KernelTime                        00:00:07.140
    QuotaPoolUsage[PagedPool]         208996
    QuotaPoolUsage[NonPagedPool]      9000
    Working Set Sizes (now,min,max)  (442, 50, 345) (1768KB, 200KB, 1380KB)
    PeakWorkingSetSize                2830
    VirtualSize                       92 Mb
    PeakVirtualSize                   190 Mb
    PageFaultCount                    13970
    MemoryPriority                    BACKGROUND
    BasePriority                      13
    CommitCharge                      572
```

Figure 18. The detailed information of process in address 81b142c0.

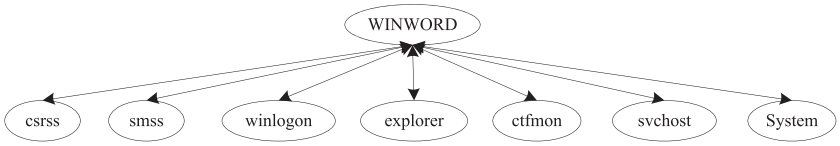

Figure 19. The shared memory correlation graph of word and other processes.

```
0x820c2f90   1044     0x13f4   0x12019f File            \Device\NamedPipe\lsarpc
```

Figure 20. NamedPipe.

For example, a handle of type named pipeline is shown in Figure 20.

And a correlation graph of processes communicating by named pipeline is shown in Figure 21.

5.2. Father–son relation between processes

In Windows, every process has one father and several sons except the idle process. Now almost all memory forensic tools can list the running process in memory. However, most of them can only provide the list but never correlate these processes. Although Volatility has a pstree command, which can present processes in father–son relation, it misses the hidden processes. So this method is not complete for forensics. This paper presents a novel method that not only can correlate father and son processes but also will not miss hidden processes. It mainly includes two steps.

Step 1. Finding all running processes in memory. A simple way to obtain the process list is based on the ActiveProcessLinks attribute of _EPROCESS. It is a doubly linked list that can point to the next and the previous processes in the list. However, criminals can remove their processes from this list with direct kernel object manipulation technique, so it is unreliable to obtain the process list this way. Fortunately, every process obtains its space from the memory pool with a pool tag, that is, Proc, so we can find all processes more reliably by scanning this tag. This paper obtains the process list using this method.

Step 2. The UniqueProcessId and InheritedFromUniqueProcessId in_EPROCESS represent the ID of the current process and its father. So according to the two IDs, father and son can be correlated. For example, for the process list in Figure 22, we can generate a father–son correlation graph shown in Figure 23.

5.3. Service relation between processes

In addition to the obvious relationships such as father–son and communication relations, there are also some implicit relationships between processes. For example, some processes seem to be independent, but in fact, they cooperate with and serve the same application. We call this kind of relationship a service relation. It has not been used in other

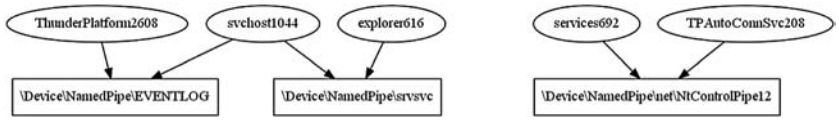

Figure 21. A correlation graph of processes communicating by named pipeline.

Offset(V)	Name	PID	PPID	Thds	Hnds	Sess	Wow64	Start	Exit
0x810b1660	System	4	0	58	379	------	0		
0xff2ab020	smss.exe	544	4	3	21	------	0	2010-08-11 06:06:21 UTC+0000	
0xff1ecda0	csrss.exe	608	544	10	410	0	0	2010-08-11 06:06:23 UTC+0000	
0xff1ec978	winlogon.exe	632	544	24	536	0	0	2010-08-11 06:06:23 UTC+0000	
0xff247020	services.exe	676	632	16	288	0	0	2010-08-11 06:06:24 UTC+0000	
0xff255020	lsass.exe	688	632	21	405	0	0	2010-08-11 06:06:24 UTC+0000	
0xff218230	vmacthlp.exe	844	676	1	37	0	0	2010-08-11 06:06:24 UTC+0000	
0x80ff88d8	svchost.exe	856	676	29	336	0	0	2010-08-11 06:06:24 UTC+0000	
0xff217560	svchost.exe	936	676	11	288	0	0	2010-08-11 06:06:24 UTC+0000	
0x80fbf910	svchost.exe	1028	676	88	1424	0	0	2010-08-11 06:06:24 UTC+0000	
0xff22d558	svchost.exe	1088	676	7	93	0	0	2010-08-11 06:06:25 UTC+0000	
0xff203b80	svchost.exe	1148	676	15	217	0	0	2010-08-11 06:06:26 UTC+0000	
0xff1d7da0	spoolsv.exe	1432	676	14	145	0	0	2010-08-11 06:06:26 UTC+0000	
0xff1b8b28	vmtoolsd.exe	1668	676	5	225	0	0	2010-08-11 06:06:35 UTC+0000	
0xff1fdc88	VMUpgradeHelper	1788	676	5	112	0	0	2010-08-11 06:06:38 UTC+0000	
0xff143b28	TPAutoConnSvc.e	1968	676	5	106	0	0	2010-08-11 06:06:39 UTC+0000	
0xff25a7e0	alg.exe	216	676	8	120	0	0	2010-08-11 06:06:39 UTC+0000	
0xff364310	wscntfy.exe	888	1028	1	40	0	0	2010-08-11 06:06:49 UTC+0000	
0xff38b5f8	TPAutoConnect.e	1084	1968	1	68	0	0	2010-08-11 06:06:52 UTC+0000	
0x80f60da0	wuauclt.exe	1732	1028	7	189	0	0	2010-08-11 06:07:44 UTC+0000	
0xff3865d0	explorer.exe	1724	1708	13	326	0	0	2010-08-11 06:09:29 UTC+0000	
0xff3667e8	VMwareTray.exe	432	1724	1	60	0	0	2010-08-11 06:09:31 UTC+0000	
0xff374980	VMwareUser.exe	452	1724	8	207	0	0	2010-08-11 06:09:32 UTC+0000	
0x80f94588	wuauclt.exe	468	1028	4	142	0	0	2010-08-11 06:09:37 UTC+0000	
0xff224020	cmd.exe	124	1668	0	--------	0	0	2010-08-15 19:17:55 UTC+0000	2010-08-15 19:17:56 UTC+0000

Figure 22. A process list obtained by Scanning Proc tag.

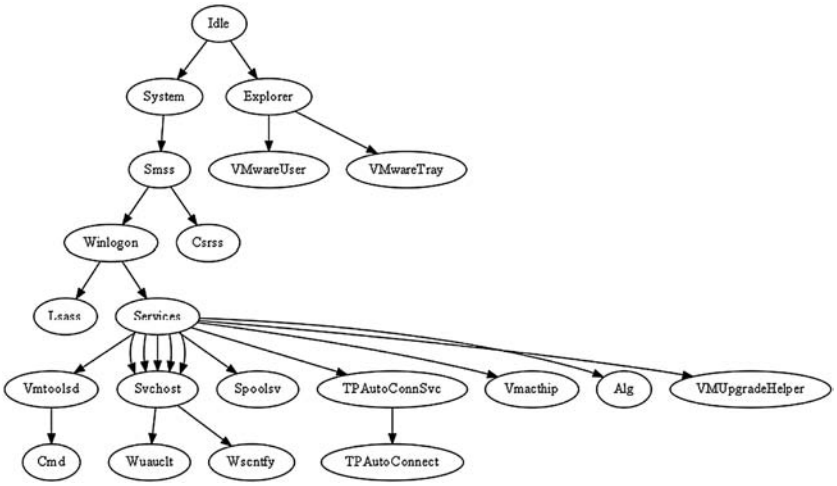

Figure 23. A father–son correlation graph.

memory forensic methods. However, based on this relationship, investigators can classify processes, infer the goal of certain unknown process, and even identify criminal behaviors.

5.3.1. Correlating processes based on name.

At first, we find this kind of relation using a simple method, that is, the similarity of process name. Generally, application developers will give their processes similar names, such as the same prefix or suffix. For example, when QQ, a famous instant communication tool in China, is running, there are four processes in memory, that is, QQ, QQProtect, QQExternal, and TxPlatform. Three of them have the same prefix. So we can correlate them by the similarity of name. However, the process TxPlatform is missed by this method. So correlating based on name is simple but not accurate. Another more accurate method is still needed.

5.3.2. Correlating processes based on DLLs.

Processes often load some DLLs to obtain certain system services or execute certain tasks. In other word, the list of DLLs that a process loads can tell us what this process could probably do. Moreover, after analyzing about 100 applications, we find that the processes serve for the same application that usually loads similar DLLs, so we can cluster processes based on DLLs. In this paper, we cluster processes using the Density-based spatial clustering of applications with noise (DBSCAN) algorithm, which is an efficient density-based clustering algorithm. The main steps are as follows.

Firstly, for each process, we should obtain the list of DLLs it loaded. This can be implemented by the method in Section 4.2.

Secondly, the processes are clustered using the DBSCAN based on their DLLs. DBSCAN is a density-based clustering algorithm because it finds a number of clusters starting from the estimated density distribution of the corresponding nodes. DBSCAN is also one of the most common clustering algorithms. We choose this algorithm because of its high efficiency and the number of clusters need not be predefined. The framework of our clustering method is shown in Figure 24. It mainly includes four steps (More details can be found in our paper [13]).

Step 1. Selecting key DLLs. In order to improve the performance of cluster, the DLL set is refined.

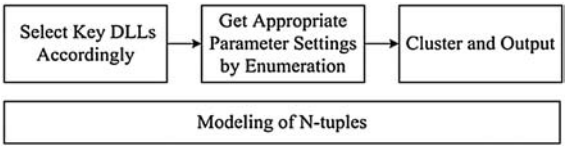

Figure 24. The framework of our clustering method.

The total number of DLLs in Windows is almost 2000, so if all of them are chosen as clustering attributes, the performance would be unbearable. So we select some key DLLs as the base of clustering. This selection follows two rules: (i) the DLLs loaded by processes frequently should be selected and (ii) DBSCAN requires that all samples should be independent. However, there are many dependencies between DLLs. So if DLL A depends on DLL B, that is, if A is loaded then B must be loaded, then only A is selected as representation. Based on the two rules and after verifying with a training set, about 300 DLLs are selected as key DLLs.

Step 2. Modeling each process into an *N*-tuple based on the key DLLs. The *N*-tuple is defined as follows. Experiments show that after modeling each process into an *N*-tuple, the speed of clustering is improved greatly.

Definition 1. An *N*-tuple is a sequence of 1 s or 0 s, where *n* is a non-negative integer and indicates the number of 1 s or 0 s in the tuple.

Definition 2. The *N*-tuple of a process means *N* key DLLs are organized as an ordered set. If any key DLL is found in this process, the corresponding bit in the tuple is set to 1; otherwise, it is set to 0.

Step 3. Obtaining appropriate parameter settings by enumeration. DBSCAN's definition of a cluster is based on the notion of density reachability. Basically, a point q is directly density reachable from a point p if it is not farther away than a given distance ε (i.e., it is part of its ε-neighborhood) and if p is surrounded by sufficiently many points such that one may consider p and q to be part of a cluster. DBSCAN requires two parameters: ε (eps) and the minimum number of points required to form a dense region (minPts). It starts with an arbitrary starting point that has not been visited. This point's ε-neighborhood is retrieved, and if it contains sufficiently many points, a cluster is started. Otherwise, the point is labeled as noise. In our implementation, the ε is set to 4 and minPts is set to 2. After enumerating almost all possible parameters' value and testing the effect, we found that this is the best choice for our methods.

Step 4. Clustering and outputting the result. For example, after clustering the processes in our test system, we obtained a cluster shown in Figure 25. We find two processes, that is, AlipaySecSvc and TaobaoProtect, that belong to this cluster. In fact, although they have different names, they both serve for the same application Alipay. From this example, we can see that our clustering method can overcome the shortage of the method in Section 5.3.1. So they can be used together in real investigation.

6. EVALUATION

In order to evaluate the effectiveness of our method, we simulated two different investigation cases. Both of them ran in VMware, so the memory dumps we obtained in the experiments were all vmem files. The operating system we used in the experiments was Windows XP SP2. Now the goal of our evaluations is only proving the feasibility and effectiveness of our methods; so in our experiments, we used both the prototype developed by ourselves and some functions of current forensic tools (such as Volatility). In the future, we will develop our independent plug based on Volatility so as to analyze memory evidence automatically.

6.1. Investigation case 1

This scenario was a simulation of the spy or identity theft on a computer. Suppose that a criminal has obtained the username and password of a certain user on a computer, and he logs onto this computer using this account. He already knows that there is a file that contains all kinds of passwords of this user on this computer. So he searches this computer and finds a file named password, then he sends this file to his email address using outlook.

In order to investigate this case, we will use the father–son relation, process–user relation, and process–file relation mentioned before.

```
1:
AlipaySecSvc.e pid:   1876
Command line : "C:\Program Files\alipay\alieditplus\AlipaySecSvc.exe"

TaobaoProtect. pid:   3712
Command line : "C:\Program Files\alipay\SafeTransaction\TaobaoProtect.exe"
```

Figure 25. An example of clustering.

Firstly, the processes were correlated by father–son relation. We obtained the complete process list, which includes the hidden ones using the psscan command in Volatility. Then we correlated these processes based on father–son correlating method in prototype. Part of the result is shown in Figure 26.

Then the processes were correlated with users. We first obtained the relation between processes and SID using the getsids command in Volatility. In order to find the true user, we filtered the universal identifiers of system. S-1-5-21-2052111302-1085031214-682003330-1003 is the only remaining SID. After searching this SID in registry, we found that the username is A. The final result of process–user correlating is shown in Figure 27. We find that many processes are correlated with A.

Finally, the processes were correlated with files. This can be implemented with the help of windbg, which is a debug tool provided by Microsoft. According to the correlation graph in Figure 27, the processes created by user A (such as word, QQ, and outlook) will be analyzed first. Using the method mentioned in Section 4.1, we found that process outlook was correlated with file password (Figure 28).

Based on the aforementioned correlation graphs, we can infer that file password was sent as an attachment by outlook. Because user A is the owner of process outlook, and A is the only account in system, we can infer that this account may be misused by the criminal.

6.2. Investigation case 2

This scenario was a simulation of the malware installation on a computer. Suppose that a criminal has obtained the username and password of a certain user on a computer and logs onto this computer using this account. Then he

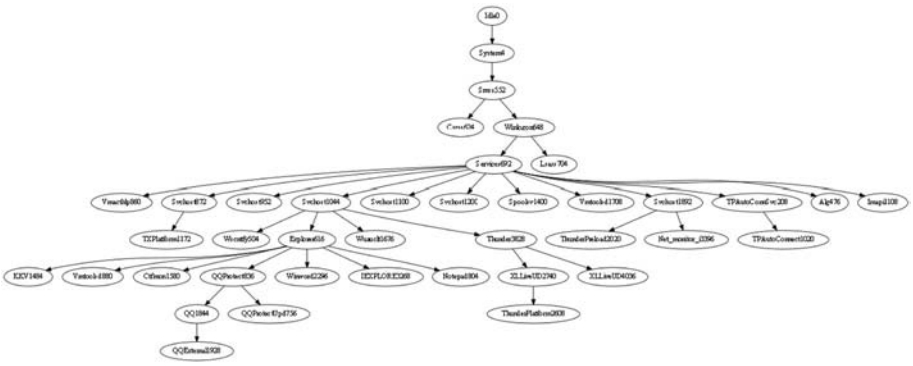

Figure 26. The father–son relation graph in case 1.

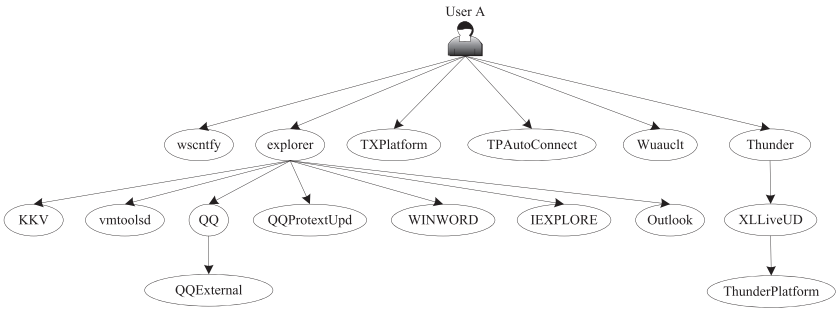

Figure 27. The user–process correlation graph in case 1.

```
processor number 0, process 81a19da0
PROCESS 81a19da0  SessionId: 0  Cid: 02f4    Peb: 7ffd7000  ParentCid: 0268
    DirBase: 0a3c0240  ObjectTable: e23ce2f8  HandleCount: 1738.
    Image: OUTLOOK.EXE

0fac: Object: 81d773d0  GrantedAccess: 00100001 Entry: e10fbf58
Object: 81d773d0  Type: (821ebe70) File
    ObjectHeader: 81d773b8 (old version)
        HandleCount: 1  PointerCount: 3
        Directory Object: 00000000  Name: \my\password.docx
```

Figure 28. The correlation between process outlook and file password.

```
000004F0   00 00 00 00 53 46 52 2D  00 00 00 00 53 46 52 41      SFR-    SFRA
00000500   00 00 00 68 00 74 00 74  00 70 00 3A 00 2F 00 2F     h t t p : / /
00000510   00 64 00 6C 00 73 00 77  00 2E 00 62 00 61 00 69   d l s w . b a i
00000520   00 64 00 75 00 2E 00 63  00 6F 00 6D 00 2F 00 73   d u . c o m / s
00000530   00 77 00 2D 00 73 00 65  00 61 00 72 00 63 00 68   w - s e a r c h
00000540   00 2D 00 73 00 70 00 2F  00 73 00 6F 00 66 00 74   - s p / s o f t
00000550   00 2F 00 39 00 64 00 2F  00 31 00 34 00 37 00 34   / 9 d / 1 4 7 4
00000560   00 34 00 2F 00 63 00 68  00 72 00 6F 00 6D 00 65   4 / A d W a r e
00000570   00 33 00 34 00 2E 00 30  00 2E 00 31 00 38 00 34   . W i n 3 2 . U
00000580   00 37 00 2E 00 31 00 31  00 36 00 6D 00 2E 00 31   n d e f . f v r
00000590   00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
```

Figure 29. A record in Taskdb.

downloads a malware and installs it on this computer. This malware will run whenever the system is started.

In order to investigate this case, we will use the communication relation, process–user relation, process–file relation, and process–network relation.

Firstly, the processes are correlated by father–son relation and then they are correlated with users. The method is the same as case 1, so we will not provide the details here. In the next step, the processes were correlated with files. We found that process Thunder was correlated with a file taskdb. We obtained this file on this computer and opened it using Winhex. The result is shown in Figure 29. We found that taskdb was the database for Thunder. It stored all the downloading tasks of Thunder. Moreover, from these records, we found that a file named adware. win32.undef.fvr was downloaded by Thunder.

Then we correlated processes by shared memory communication. We found that several processes communicate with Process IEXPLORER, including Thunder. The correlating steps are as follows. Firstly, the VAD nodes of IEXPLORER were obtained (Figure 30). Secondly, for mapped VAD nodes, we found the node whose FilePointer was null by checking its _MMVAD and _CONTROL_AREA (Figure 31). That means this node

```
process:IEXPLORER
lkd> !vad 81b3f0e0
VAD     level      start      end    commit
81e651e8 ( 3)       54b0     55af         2 Private      READWRITE
81e80450 ( 2)       55b0     56af         2 Private      READWRITE
81f04588 ( 1)       56b0     56d1        34 Private      READWRITE
81a1ff10 ( 2)       56e0     572c        77 Private      READWRITE
81c917e0 ( 3)       5730     5740         0 Mapped       READONLY
81cd20b8 ( 4)       5750     5750         1 Private      READWRITE
81fd9b48 ( 0)       5770     5b6f       925 Private      READWRITE
81c8ac68 ( 1)       5b70     5b71         0 Mapped       READONLY
81f3eed0 ( 3)       5b80     5c7f        12 Private      READWRITE
81f5dda0 ( 4)       5c80     5c88         9 Private      READWRITE
81bc1d68 ( 5)       5c90     5cd7        72 Private      READWRITE
81c328e0 ( 6)       5ce0     5ce1         2 Private      READWRITE
820dc6a8 ( 2)       5cf0     5cf1         0 Mapped       READONLY
81cd9d98 (-1)       5d00     5f8b       652 Private      READWRITE
81c4a358 ( 4)       5f90     608f         2 Private      READWRITE
81e85e88 ( 3)       6090     618f         2 Private      READWRITE
81d85bf8 ( 2)       6190     628f       255 Private      READWRITE
81d882c8 ( 6)       6290     638f         2 Private      READWRITE
81c2d8c8 ( 5)       6390     640c         0 Mapped       READWRITE
81bcba30 ( 6)       6410     648c         0 Mapped       READWRITE
82039418 ( 4)       6490     658f         2 Private      READWRITE
81cf70e0 ( 3)       6590     668f        12 Private      READWRITE
81b384c8 ( 4)       6690     66c0         0 Mapped       READONLY
81b38468 ( 1)       66d0     66e0         0 Mapped       READONLY
81a18480 ( 3)       66f0     66f0         1 Private      READWRITE
8209f3b0 ( 2)       6700     67ff         3 Private      READWRITE
81b2dce8 ( 3)       6800     6800         0 Mapped       READONLY
81c8da68 ( 4)       6810     684f         6 Private      READWRITE
81e768c8 ( 0)       6850     6850         0 Mapped       READONLY
81f57698 ( 4)       6860     6860         0 Mapped       READONLY
```

Figure 30. The VAD node of IEXPLORER.

```
lkd> dt nt!_MMVAD 820e1c80
   +0x000 StartingVpn      : 0x130
   +0x004 EndingVpn        : 0x132
   +0x008 Parent           : 0x820a7348 _MMVAD
   +0x00c LeftChild        : (null)
   +0x010 RightChild       : (null)
   +0x014 u                : __unnamed
   +0x018 ControlArea      : 0x81d73d88 _CONTROL_AREA
   +0x01c FirstPrototypePte : 0xe1b01e20 _MMPTE
   +0x020 LastContiguousPte : 0xe1b01e30 _MMPTE
   +0x024 u2               : __unnamed
lkd> dt nt!_CONTROL_AREA 0x81d73d88
   +0x000 Segment          : 0xe1b01de0 _SEGMENT
   +0x004 DereferenceList  : _LIST_ENTRY [ 0x0 - 0x0 ]
   +0x00c NumberOfSectionReferences : 1
   +0x010 NumberOfPfnReferences : 0
   +0x014 NumberOfMappedViews : 0x11
   +0x018 NumberOfSubsections : 1
   +0x01a FlushInProgressCount : 0
   +0x01c NumberOfUserReferences : 0x12
   +0x020 u                : __unnamed
   +0x024 FilePointer      : (null)
   +0x028 WaitingForDeletion : (null)
   +0x02c ModifiedWriteCount : 0
   +0x02e NumberOfSystemCacheViews : 0
```

Figure 31. VAD and ControlArea.

represents a block of shared memory. Thirdly, the _SEGMENT structure was checked to obtain the value of u1attribute (Figure 32). As mentioned in Section 5.1.1, u1 points to the process that created this shared memory. So we obtained an address 81b142c0. After executing the

```
lkd> dt nt!_SEGMENT 0xe1b01de0
   +0x000 ControlArea      : 0x81d73d88 _CONTROL_AREA
   +0x004 TotalNumberOfPtes : 3
   +0x008 NonExtendedPtes  : 3
   +0x00c WritableUserReferences : 0
   +0x010 SizeOfSegment    : 0x3000
   +0x018 SegmentPteTemplate : _MMPTE
   +0x020 NumberOfCommittedPages : 3
   +0x024 ExtendInfo       : (null)
   +0x028 SystemImageBase  : (null)
   +0x02c BasedAddress     : (null)
   +0x030 u1               : __unnamed
   +0x034 u2               : __unnamed
   +0x038 PrototypePte     : 0xe1b01e20 _MMPTE
   +0x040 ThePtes          : [1] _MMPTE
lkd> dd 0xe1b01de0 + 0x030
e1b01e10  81b142c0 01760000 e1b01e20 00000000
e1b01e20  181bd163 80000000 181be163 80000000
e1b01e30  17794123 80000000 0001040c 61626453
e1b01e40  1c060401 8208d9a8 e1624cf8 e1624cf8
e1b01e50  e25c3ea8 e16c4748 e1b01e58 e1b01e58
e1b01e60  00000000 00010002 81c3dc80 00000000
e1b01e70  00010406 6d695346 0c070401 61564d43
e1b01e80  002c0000 00166b76 80000004 00000000
```

Figure 32. The value of u1 in _SEGMENT.

```
lkd> !object 81b142c0
Object: 81b142c0  Type: (821b9e70) Process
    ObjectHeader: 81b142a8 (old version)
    HandleCount: 3  PointerCount: 205
```

Figure 33. The result of !object.

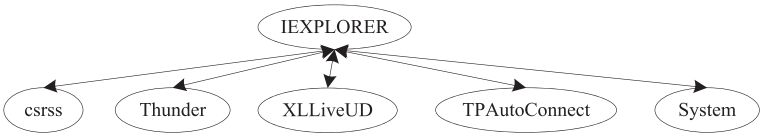

Figure 34. The communication correlation graph based on shared memory for IEXPLORER.

command !object in windbg, we found that this address really stored a process (Figure 33), which was Thunder. That means IEXPLORER communicated with Thunder using shared memory. So we can infer that the criminal may have opened a website using IE and downloaded a malware adware.win32.undef.fvr using Thunder.

In fact, many processes communicated with IEXPLORER using shared memory. The correlation graph is shown in Figure 34.

Then the processes are correlated with the network. We obtained the TCP connections created by Thunder using the connscan command in Volatility. From the result, we can obtain the IP address of the downloading tasks.

Finally, in order to confirm whether adware.win32.undef.fvr was a malware, we check the Task Scheduler in this system, and we found that this program will run whenever this system starts. After searching for the introduction of adware.win32.undef.fvr on the Internet, we confirm it is a malware.

7. CONCLUSION

This paper presents an automatic memory analysis methodology based on data correlation. Through analyzing key OS data structures and utilizing a clustering algorithm, this methodology can discover the relationships among processes, files, users, DLLs, and network connections. Then it organizes this data into correlation graphs so as to disclose the meaning of evidence in a high semantic level. Compared with only providing information to investigators, our experiments have proven that these correlation graphs can help investigators find hidden criminal behaviors and reconstruct the criminal scenarios. The limitation of our methodology is that it will be affected by the change of OS versions, because the data structures we based our methods on may change in different versions. So in the future, we will try to find another platform-independent methods.

ACKNOWLEDGEMENTS

This work was supported by the National Natural Science Foundation of China (61100198/F0207, 61100197/F0207), as well as the US Army Research Office under the grant WF911NF-14-1-0518.

REFERENCES

1. Stevens RM, Casey E. Extracting windows command line details from physical memory. *Digital Investigation* 2010; **7**(5):57–63.
2. Hargreaves C, Chivers H. Recovery of encryption keys from memory using a linear scan, *Proceedings of the 2008 third international conference on availability, reliability and security*(*ARES'2008*) 2008; 1369–1376.
3. Schuster A. Searching for processes and threads in Microsoft Windows memory dumps. *Digital Investigation* 2006; **3**(1):10–16.
4. Schuster A. Pool allocations as an information source in windows memory forensics, *Proceedings of IT-incident management & IT-forensics*(*IMF'2006*) 2006; 104–115.
5. Bilby D. Low down and dirty: anti-forensic rootkits, *Proceedings of Ruxcon'2006*, 2006; 34–41.
6. Zhang R, Wang L, Zhang S. Windows memory analysis based on KPCR, *Proceedings of Information Assurance and Security* (*IAS'2009*) 2009; 677–680.
7. Dolan-Gavitt B. Forensic analysis of the windows registry in memory. *Digital Investigation* 2008; **5**(1):26–32.
8. Okolica J, Peterson GL. Windows operating systems agnostic memory analysis. *Digital Investigation* 2010; **7**(1):48–56.
9. Ionescu A, Russinovich ME, Solomon DA. *Microsoft Windows Internals*. USA: Microsoft Press, 2009; 56–89.
10. Dolan-Gavitt B. The VAD tree: a process-eye view of physical memory. *Digital Investigation* 2007; **4**(2): 62–64.
11. Van Baar RB, Alink W, Van Ballegooij AR. Forensic memory analysis: files mapped in memory. *Digital Investigation* 2008; **5**(3):52–57.
12. http://code.google.com/p/volatility/, Volatility maintained by Volatility Foundation, 2014.
13. Duan YH, Fu X, Luo B, Wang ZQ, Shi J. Detective: automatically identify and analyze malware processes in forensic scenarios via dynamic-link libraries. *Proceedings of ICC 2015*, 2015; 1–6.