The Quarterly Magazine for Digital Forensics Practitioners

Issue 52 • Q1 2023

DIGITAL FORENSICS Magazine

FORENSIC ANALYSIS of the NETWIRE STACK

They said you couldn't determine that an attacker using the NetWire RAT delivered particular files to a victim's computer. They were wrong.

PLUS

- Rethinking Remote DFIR
- Challenges Detecting Email Scams
- Empowering Digital Forensics in Policing
- **Regulars:** News, Legal, IRQ & More!

ALSO AVAILABLE FROM

ANCHORS IN R

FEATURED EXCLUSIVELY IN

DIGITAL FORENSICS M

/ LEAD FEATURE

BEYOND TIMELINES ANCHORS IN RELATIVE TIME

What happens to our timelines when we have reason to believe that critical dates and times from file systems cannot be trusted, not even to be consistently inaccurate? Mark Spencer explains...

/ INTERMEDIATE

Originally published in Issue 18, February 2014

Digital forensics and incident response (DFIR) practitioners use timelines to efficiently identify and better understand suspicious activity. The use of timelines has always been a core component of DFIR analysis (see Clifford Stoll's work in The Cuckoo's Egg), but over the last few years the importance of timelines has been increasingly highlighted in research [1], software [2] and training [3].

The foundations of timelines are obviously built on dates and times. We prefer timelines built on accurate dates and times, but consistently inaccurate dates and times, even in small clusters, may be extremely valuable. What happens to our timelines when we have reason to believe that critical dates and times from file systems, logs, embedded in documents, etc. cannot be trusted, not even to be consistently inaccurate?

We have confronted cases involving such widespread date and time tampering that the utility of "traditional" timeline analysis came into question. We realized that we had to dig deeper in these cases, and began formalizing our practice of identifying both legitimate and illegitimate anchors in relative time.

Let's break down the concept of identifying legitimate and illegitimate anchors in relative time. Legitimate and illegitimate anchors, ('anchors' are simply solid events we can rely on) are events that we can be confident are either legitimate or illegitimate, and upon which we can base additional analysis; sometimes without the benefit of accurate dates and times or, without any associated dates and times at all. Legitimate anchors used in the past have involved Microsoft's Windows operating system being installed, starting up, and shutting down.

Illegitimate anchors we have identified have involved the introduction of malware and anti-forensics tools (most often, data scrubbers) and/or remnants of their execution. "Relative time" refers to the time in which events happened in a certain order, but we cannot be certain the dates and times associated with those events (assuming that dates and times related to those events exist at all) are accurate. In fact, we are often certain that the dates and times related to these events are inaccurate.

The focus of this article is on anchors within your electronic evidence (i.e. internal anchors), but evidence does not exist in a vacuum; it exists in context with external anchors that might include court orders, video footage, historical events, etc. The designation of legitimate and illegitimate anchors should be guided and supported by what you know about the case and what you have learned about your evidence. Heavy doses of sanity checking are important here. For example, leveraging dates and times "from" external sources (e.g. web browsing *content*) within your electronic evidence. Also, in order to apply these anchors to relative time, we must be certain in what order they occurred and for that we can only rely on certain types of data which are not often exposed by digital forensics tools.

> "RELATIVE TIME REFERS TO THE TIME IN WHICH EVENTS HAPPENED IN A CERTAIN ORDER, BUT WE CANNOT BE CERTAIN THE DATES AND TIMES ASSOCIATED WITH THOSE EVENTS ARE ACCURATE."

15

M ARSENAL...

RELATIVE TIME

ARTICLES 1 & 2

MAGAZINE ISSUES 18 AND 27

APPLYING ANCHORS IN RELATIVE TIME

Our most powerful weapon reveals sophisticated evidence tampering which led to journalists' wrongful imprisonment, explains Mark Spencer...

/ INTERMEDIATE

Originally published in Issue 27, May 2016

Everyone said it was the malware. Everyone, including digital forensics experts at universities and consulting companies in the United States and abroad, was wrong. The failure of so many experts to identify an unprecedented series of electronic attacks against journalists, the likes of which we may never see again, almost resulted in the fascinating truth being buried forever.

"Anchors in Relative Time" ("ART") is an analysis technique described in my article "Beyond Timelines – Anchors in Relative Time" published in *Digital Forensics Magazine* Issue 18. As a quick summary, this technique involves identifying legitimate and illegitimate anchors within electronic evidence that can be placed in relative time (time in which events have happened in a certain order) regardless of whether dates and times associated with those anchors are accurate. My last article focused on three particular types of anchors found on Microsoft Windows ("Windows") systems to which I now add a couple more as highlighted in Table 1.

These types of anchors are particularly useful when determining the order in which events have occurred, regardless of any associated dates and times, as they normally increment in the order events have occurred. See the Definitions section at the end of this article for more details on each anchor type and source. Important events in our cases have included Windows starting up and shutting down, malware introduction and execution, and critical documents being created and deleted.

Odatv is a secular news organization founded in 2007 with a reputation for being critical of Turkey's government, controlled since 2002 by the Islamic Justice and Development Party (a.k.a. the AKP, in Turkish, Adalet ve Kalkınma Partisi or AK Parti). The Odatv website, odatv.com, is one of the most popular websites in Turkey.

In February and March 2011, the Turkish National Police began a series of raids and arrests involving Odatv. Critical electronic evidence seized during the raids appeared to connect Odatv employees and supporters to the Ergenekon terrorist organization. In November 2011, an indictment in essence charged Odatv with being the media wing of Ergenekon and singled out 14 Odatv employees and supporters. The indictment was based on electronic documents seized during the raids by the Turkish National Police, leading to the imprisonment of 11 of the 14 suspects.

Barıs Pehlivan, whose Odatv computer is the focus of this article, is a well-known investigative journalist, editor, producer and author, who worked at Odatv since its 2007 origin. He was among those arrested in February 2011, and was imprisoned for a year and a half (February 14, 2011 – September 14, 2012) based on documents recovered from both his Odatv and personal computers. As you will soon learn, those documents were not quite what they seemed.

Arsenal has extensive experience uncovering evidence spoliation and we are sceptical of any evidence related to Ergenekon and other high-profile Turkish trials, such as Sledgehammer.

The presence of malware on Odatv-related computers (Barıs Pehlivan's Odatv and personal computers, Müyesser Yıldız's personal computer) has been documented in a cursory way in many technical reports. While malware was in fact found on Mr. Pehlivan's Odatv Computer, it was readily apparent after applying ART that malware was not responsible for the creation and deletion of the incriminating documents.

/ ERGENEKON

Ergenekon is an alleged secularist "deep state" in Turkey with ties to the military, academia, NGOs, and the media. Ergenekon members were charged with plotting to overthrow the Turkish government in a series of 15 indictments between 2008 and 2011.

/ SLEDGEHAMMER

Sledgehammer involves the alleged planning of a Turkish military coup in response to the election of the AKP. Forged documents critical to the Sledgehammer trial include purported plans to bomb mosques, shoot down a fighter jet, and ultimately overthrow the Turkish government. ART analysis revealed the true nature of the forged documents.

Anchor Type	Anchor Source
Log Sequence Numbers ("LSNs")	NTFS $LogFile
Record Numbers (Sequence Number 1s)	NTFS $MFT
SecurityIds	NTFS $Secure
Update Sequence Numbers ("USNs")	NTFS $UsnJrnl
RecordNumbers or EventRecordIDs	Event Logging or Windows Event Log

Table 1. Anchor Summary

/ WINDOWS STARTUPS AND SHUTDOWNS PER EVENT LOG SERVICE

To become properly oriented with a piece of evidence using ART it is often useful to identify "legitimate" anchors involving Windows startups and shutdowns. Identifying these anchors on Windows boot volumes is relatively straightforward, but identifying them on auxiliary volumes can be quite challenging. Why then deal with the frustration of identifying these anchors not only on Windows boot volumes but also on auxiliary volumes? Generally speaking, operating in the initial fog of suspected evidence tampering demands anchors on every volume that can be relied upon, even if their associated dates and times cannot be trusted. More specifically, suspicious activity was found on both volumes of Mr. Pehlivan's Odatv computer and these anchors proved to be critical to understanding what actually happened to them.

In order to identify anchors reflecting Windows startups and shutdowns on Mr. Pehlivan's Odatv computer (on both the Windows boot and auxiliary volumes) we used a combination of Event Log service events and file system transactions.

Starting with the Event Log service, we identified startups and shutdowns of that particular service, which are normally consistent with Windows startups and shutdowns. We attained a high level of comfort with the values in Table 2 as legitimate anchors by looking for signs of tampering (e.g., inconsistencies between date/times and the normal progression of RecordNumbers) which would have affected the Event Logs (finding none), reviewing Event Log service startups and shutdowns over time, comparing these events to file system transactions discussed in this article, and considering what we know from external anchors such as the normal behaviour of Odatv employees.

While we were comfortable that the anchors from February 9, 2011 onward mentioned in Table 2 were legitimate (their dates and times were consistent with "real time", i.e., their dates and times could be relied upon) we found RecordNumbers 28231/28250 and 28323/28343 unusual based on our review of Event Log service startups and shutdowns over time as well as our understanding of the normal behaviour of Odatv employees.

RecordNumber	Event Number	Event Description	Date/Time (UTC)
28202	6005	Event Log Service Start	02/09/2011 07:44:03
28229	6006	Event Log Service Stop	02/09/2011 17:58:46
28231	6005	Event Log Service Start	02/09/2011 20:09:14
28250	6006	Event Log Service Stop	02/09/2011 20:10:13
28252	6005	Event Log Service Start	02/10/2011 08:05:42
28295	6006	Event Log Service Stop	02/10/2011 18:03:32
28297	6005	Event Log Service Start	02/11/2011 07:39:13
28321	6006	Event Log Service Stop	02/11/2011 17:18:31
28323	6005	Event Log Service Start	02/11/2011 20:54:13
28343	6006	Event Log Service Stop	02/11/2011 20:55:16

Table 2. Partition 1 – System Event Log – Windows Start/Stops

Light Blue = Event Log Service Start
Dark Blue = Event Log Service Stop

> "THE FAILURE OF SO MANY EXPERTS TO IDENTIFY AN UNPRECEDENTED SERIES OF ELECTRONIC ATTACKS AGAINST JOURNALISTS, THE LIKES OF WHICH WE MAY NEVER SEE AGAIN, ALMOST RESULTED IN THE FASCINATING TRUTH BEING BURIED FOREVER."

/ WINDOWS STARTUPS AND SHUTDOWNS PER FILE SYSTEM TRANSACTIONS ON FIRST PARTITION

Next, we identified file system transactions in the NTFS $UsnJrnl and $LogFile metafiles which uniquely identified Windows startups and shutdowns.

After modelling Windows startups and shutdowns on the Windows boot volume (the first partition) of Mr. Pehlivan's Odatv computer over time, we found that $UsnJrnl transactions "DATA_TRUNCATION" and "CLOSE+DATA_EXTEND+DATA_TRUNCATION+SECURITY_CHANGE" involving pagefile.sys uniquely and consistently identified Windows startups and shutdowns. See Table 3 for a list of those transactions from February 9, 2011 onward. →

FORENSIC ANALYSIS of the NetWire Stack

They said you couldn't determine that an attacker using the NetWire RAT delivered particular files to a victim's computer. They were wrong.

Our casework at Arsenal has involved the analysis of computers compromised by versions of the NetWire remote access trojan (RAT) up through 1.7 R11. Although the NetWire version we are focusing on in this article is 1.7 R11 (released in 2018), and the current version is 2.1, many of the concepts we describe in this article still apply to the current version. Please note that this article is focused not only on NetWire 1.7 R11 but also on Windows 7 32-bit as the compromised operating system.

NetWire was used in one of our highest-stakes cases to conduct long-term surveillance and surreptitiously deliver incriminating documents which were later used in criminal prosecution. While simple artifacts related to NetWire execution can be found in places such as the Windows Registry and prefetch files, we needed to know much more than when NetWire was running on compromised computers. We discovered that additional insight could be found by analyzing the various portions of memory used by NetWire that would sometimes end up stored on disk. Most importantly among these portions of memory is the stack used by the main NetWire thread. You will learn more about stacks and threads soon. NetWire stacks (particularly the stack used by the main thread which we focus on in this article), contain information that includes - amongst many other things - control codes sent by NetWire command and control servers (hereafter referred to simply as "c2"). We have found both complete and partially intact NetWire stacks in Windows swap (pagefile.sys), hibernation (hiberfil.sys), crash dumps (memory.dmp), and even in unallocated clusters. Due to the unique structure of some data contained within NetWire stacks, incredibly valuable information about a NetWire operator's activity conducted on a victim's machine can be recovered not only from complete stacks within disk images but from partially intact stacks as well. As an example of this incredibly important information, you may be able to identify (as we did) where a NetWire operator uploaded a particular file to on a victim's computer, when they uploaded the file, and even find content from the uploaded file still residing in the stack.

Before we discuss NetWire stacks in more detail, let's get some basics out of the way.

What is NetWire?

NetWire was a popular multi-platform RAT system until March 2023 when international law-enforcement cooperation resulted in the seizure of NetWire infrastructure and the arrest of its administrator. Previously, the NetWire system could be obtained by attackers a variety of ways, one of which was purchase from the official World Wired Labs website [1] that now displays a law enforcement seizure notice. NetWire was quite powerful and had been under ongoing development for many years - for example, news on the World Wired Labs website related to version updates went back to June 2013. In addition to remote control features which included uploading and downloading files, NetWire offered more insidious features such as proxy chaining (making the identification of attackers more difficult), "stealth" screenshots, key logging and password "recovery."

GitHub

A GitHub project associated with this article contains additional resources related to NetWire stack analysis, including the open source tool NwStacks. NwStacks supports the analysis of Windows 7 (32/64-bit), Windows 8.1 (64-bit), and Windows 10 (64-bit) operating systems which have been compromised by NetWire. You can find this GitHub project at https://github.com/ArsenalRecon/NetWireStackForensics.

1. https://www.worldwiredlabs.com

The NetWire host (a/k/a agent) running on a victim's computer receives control codes (a/k/a commands) from a NetWire c2. The NetWire host executable can be compiled as a 32-bit module for Windows, GNU/Linux, Android, and Mac OS X. While the c2 supports the detection of a Solaris host, it does not appear that a Solaris host was available in version 1.7. Traffic between the NetWire host and its c2 is encrypted, except for control codes and payload sizes. A complete set of control codes that we have identified so far can be found at https://github.com/ArsenalRecon/NetWireStackForensics/blob/main/NetWire1.7-controls.txt.

Control code data sent from a NetWire c2 has the following format as seen in raw network packets:

Bytes 0 - 3: Size of payload in little-endian (4 bytes)
Byte 4: Control code (1 byte)
Bytes x - y: Encrypted payload of variable length, depending on the control code and its arguments

What is a process stack?

A process is essentially a program in its running state. When a program is launched and a process is started, memory is reserved for the storage of crucial data that makes it possible for the process to keep track of its own execution, arguments to function calls, local variables, and return addresses. This memory is a process stack (hereafter, stack). The initial stack structure is determined by the compiler when the program is compiled. With every function call a stack frame is created and stored on the stack. A stack frame contains arguments, a return address, and space for local variables. The size of a stack frame depends on the amount of memory needed for the execution of a function. When a function returns, the stack pointer is reset to its value prior to the function call. Figure 1 (from Wikipedia, see https://en.wikipedia.org/wiki/Call_stack) illustrates a sample stack.

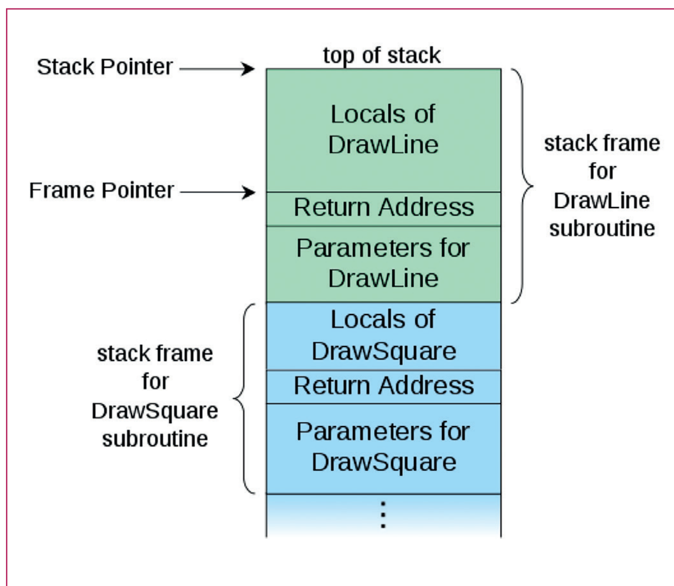


Figure 1. Sample stack illustration

The size of a stack frame depends on the amount of memory needed for the execution of a function. When a function returns, the stack pointer is reset to its value prior to the function call.

In other words, the stack depth depends on the number of nested function calls and the sizes of their associated frames. During the lifetime of a process, its stack may contain one or more stack frames from previously run functions left behind after the stack depth has changed. Stack frames from previously run functions may end up completely overwritten, partially overwritten, or not overwritten at all depending on the program's design and the specific code paths executed. ▷

Memory address referencing

When NetWire is loaded, its private virtual address is determined by a setting in the PE header called ImageBase. Internal address referencing for the program/code itself is done by using Relative Virtual Address (RVA). References to code as found in the stack will be a Virtual Address, hereafter referenced as "VA", which is calculated by ImageBase and RVA. There are settings in the PE header that may override ImageBase. In NetWire 1.7 the PE header's Optional Header, DllCharacteristics, is set to 0x100 (NX_COMPAT). Because 0x40 (DYNAMIC_BASE) is not set, the load address will always be that of the ImageBase setting (0x400000), and code VA's found on the stack will thus match the address seen in disassemblers such as IDA.

NetWire functionality involves the use of various threads which in turn rely on multiple stacks and usage of external heap memory. The "main" thread (and its stack) consists of high-level synchronization, socket handling, and input (from the c2) handling. Additional threads and their stacks are created based on the usage of particular NetWire functionality. The primary focus of the analysis described in this article is on the main thread and its stack.

NetWire Components

The core elements of a NetWire process include:

- Handling of sockets (the main thread and its stack)
- Processing of input controls from a c2 (the main thread and its stack)
- Sending data back to c2. In most cases this data is stored in an external heap where encryption is applied
- Keylogging (an additional thread that uses another stack)

An Overview of the NetWire Stack

As explained previously, stack frames are used in function calls and the presence of previous frames may depend on the code paths executed. As we will now see, the part of the program that handles higher-level synchronization and sockets does not reserve much space on the stack. In contrast, the part of the program responsible for managing commands from the c2 reserves far more space on the stack. The distinct code parts consistently overwrite the same areas, leaving the artifacts of the specific components in the exact same fixed locations.

File: Host.exe
Dos Header
Nt Headers
File Header
Optional Header
Data Directories [x]
Section Headers [x]
Export Directory
Import Directory
Relocation Directory
Address Converter
Dependency Walker
Hex Editor
Identifier
Import Adder
Quick Disassembler
Rebuilder
Resource Editor
UPX Utility

Host.exe

Member	Offset	Size	Value	Meaning
Magic	00000098	Word	010B	PE32
MajorLinkerVersion	0000009A	Byte	02	
MinorLinkerVersion	0000009B	Byte	19	
SizeOfCode	0000009C	Dword	0001C400	
SizeOfInitializedData	000000A0	Dword	00004800	
SizeOfUninitializedData	000000A4	Dword	00006800	
AddressOfEntryPoint	000000A8	Dword	00002BCB	.text
BaseOfCode	000000AC	Dword	00001000	
BaseOfData	000000B0	Dword	0001E000	
ImageBase	000000B4	Dword	00400000	
SectionAlignment	000000B8	Dword	00001000	
FileAlignment	000000BC	Dword	00000200	
MajorOperatingSystemVers...	000000C0	Word	0004	
MinorOperatingSystemVers...	000000C2	Word	0000	
MajorImageVersion	000000C4	Word	0001	
MinorImageVersion	000000C6	Word	0000	
MajorSubsystemVersion	000000C8	Word	0004	
MinorSubsystemVersion	000000CA	Word	0000	
Win32VersionValue	000000CC	Dword	00000000	
SizeOfImage	000000D0	Dword	0002C000	
SizeOfHeaders	000000D4	Dword	00000400	
CheckSum	000000D8	Dword	00030764	
Subsystem	000000DC	Word	0002	Windows GUI
DllCharacteristics	000000DE	Word	0100	Click here
SizeOfStackReserve	000000E0	Dword	00200000	

Figure 2. Optional Header of executable

Stack size and layout (Visually)

Let's take a look at how some of the data originating from NetWire's startup code is arranged on the stack. The table below shows how the stack size grows when certain events (code paths) occur.

Address	Size	VA	State
0x1FB000	0x35000	-	Only with registry use, password recovery (web or mail)
0x1FC000	0x34000	00402D11	After first call to 401092 when auth is sent to c2.
0x1FD000	0x33000	00402BDC	After second call (not yet connected to c2)
0x1FE000	0x32000	00402BD5	After first call
0x22D000	0x3000	00402BCB	Entry point
0x22FFFF	0x0	-	Start

Figure 3. A stack is read from the bottom up. In the case of NetWire, the initial stack size is predictable

The stack is usually found in a state with a size of 0x33000 bytes (process is running but not connected to c2) or 0x34000 bytes (connected to c2). Thus, we will use the 0x34000 size as a baseline in our analysis. We have prepared a set of suitable bitmaps from stack snapshots to represent the stack visually. Snapshots from the states when the size is below 0x34000 are normalized to 0x34000 with 00s prepended. The bytes are then inverted (xor'ed with 0xFF) to make 00's appear as white. Then a bitmap of 16bpp was created with a layout of 256x416, which then perfectly adds up with the stack bytes as; 256 x 416 x 2 = 212992 - > 0x34000.

Stack frames from previously run functions may end up completely overwritten, partially overwritten, or not overwritten at all depending on the program's design and the specific code paths executed.

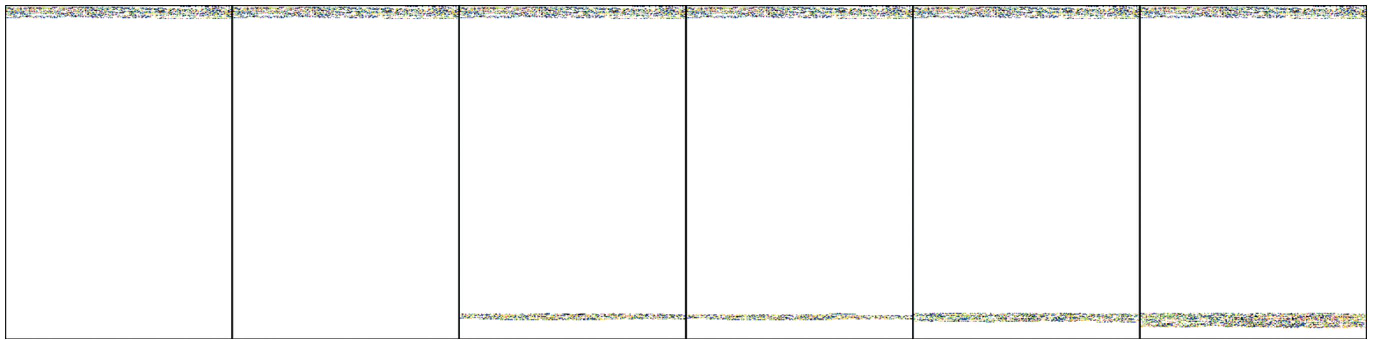


Figure 4. Stack snapshots of host with states 1 to 6 from left to right

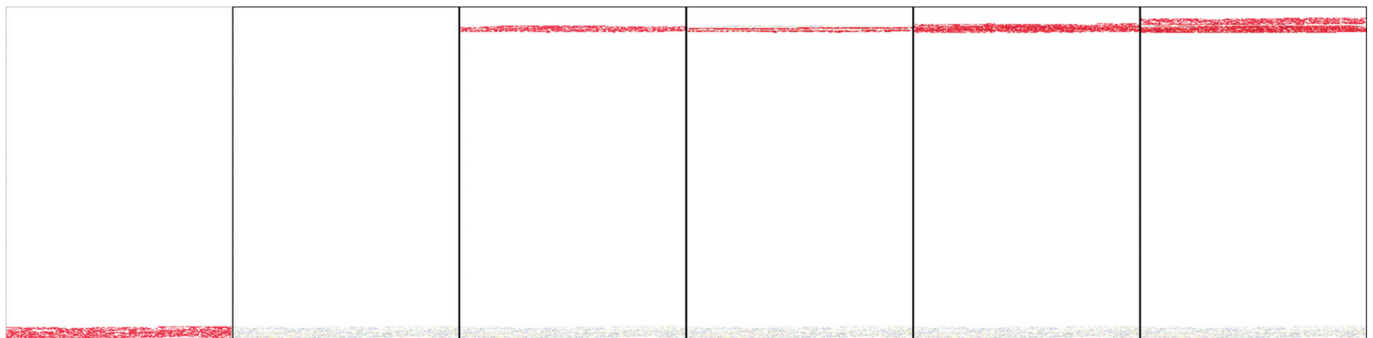


Figure 5. Stack snapshot differences highlighted in red

The six snapshots in Figure 4 above are from the following stack states:

1. Entry point - size 0x3000
2. First call - size 0x32000
3. Second call - size 0x33000
4. First five calls - size 0x33000
5. Running, not connected to c2 - size 0x33000
6. Connected to c2 - size 0x34000

In Figure 5, we have a set of bitmaps with differences, as produced by the compare functionality in Image Magick, taken from the same snapshots and order. A shadow is applied to the existing data, and the differences are highlighted in red.

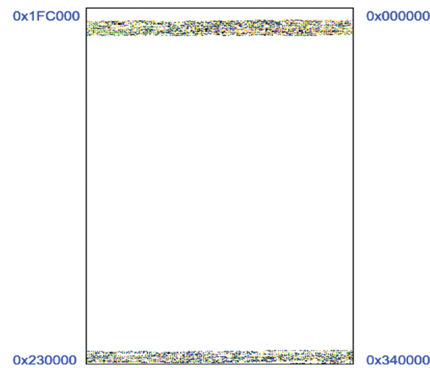


Figure 6. Stack bitmap with address and offset mapping

The bitmaps shown in Figure 6 establish a visual understanding of the stack layout, especially which areas are modified after various events occur. The normalized stack will thus have the following memory addresses (left side) and translated file offsets (right side):

At this stage, we can spot some distinct sections from the above bitmap:

- **Top**. Data section that may change when handling sockets and processing input.
- **Mid**. Large area reserved for the raw input received from c2.
- **End**. This small data section is static over the process's lifetime (caused by the startup code).

We now have a rough idea of what kind of data can be found in the various sections.

High-level code flow explanation

Let's take a look at the code. We will begin with the function prologue (start) of the most critical calls in the chain, starting with the entry point. For simplicity, we will split the explanation into 3 major levels (A through C):

Level A

```
.text:00402BCB    mov     eax, 3002Ch
.text:00402BD0    call    sub_41CE38
.text:00402BD5    sub     esp, eax
```

Level A handles higher level synchronization, socket handling, and basic control validation. Incoming data from the c2 in the form of a payload arrives to the host in smaller network packets and is stored on the stack in chunks of maximum 0x2ffff bytes. The next chunk is stored on the stack when the current chunk has been fully processed. This section of the code runs in a loop checking socket status. It is important to note that the size 0x2ffff fits within the reserved function workspace of 0x3002C. This area is represented by the large white midsection in the bitmap Figures 4, 5, and 6 above.

Process initialization (before execution arrives at the entry point) is found at the end, untouched, for the lifetime of the process. Thus, from the top of a typical stack (at the bottom of Figure 6 with the higher addresses ▷

towards 0x230000) we will find some unique data representing the absolute end. Or, more precisely, the starting point.

Figures 7 and 8 show what the end of the stack looks like initially and that the values at addresses 22ff88 and 22ff80 are written to the stack after the execution of the first few instructions going into the first function call. The data seen from 22ff8c and to the end at 22ffff is the initial data setup by the kernel. The only usage of this section of the stack is for validation. For example, we can spot several references to the entry point (see Figures 3 and 4). The entry point address in this section represents a regular process start of the standard host executable. In the case of other non-regular methods of starting the process, such as through process hollowing, the entry point address in this section may have a different value.

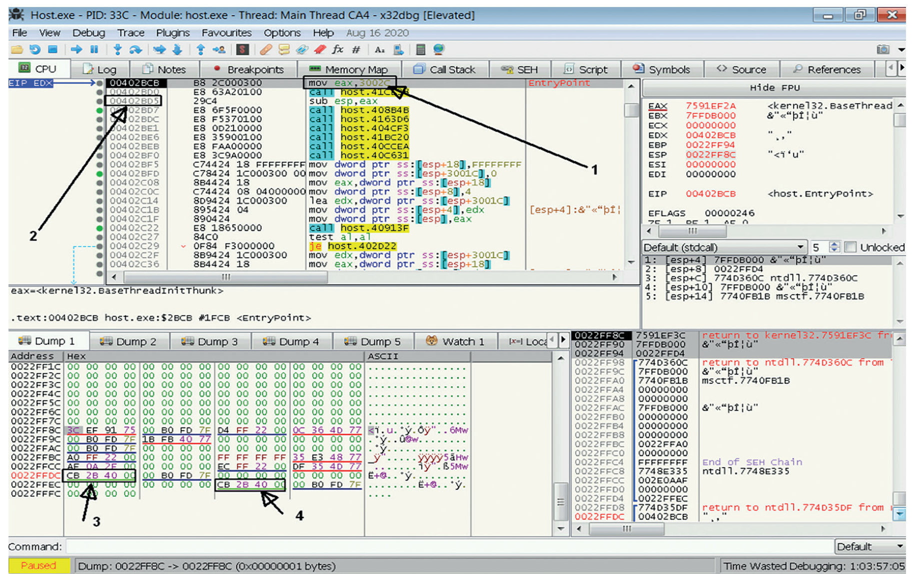


Figure 7. Debugger stopped at entry point

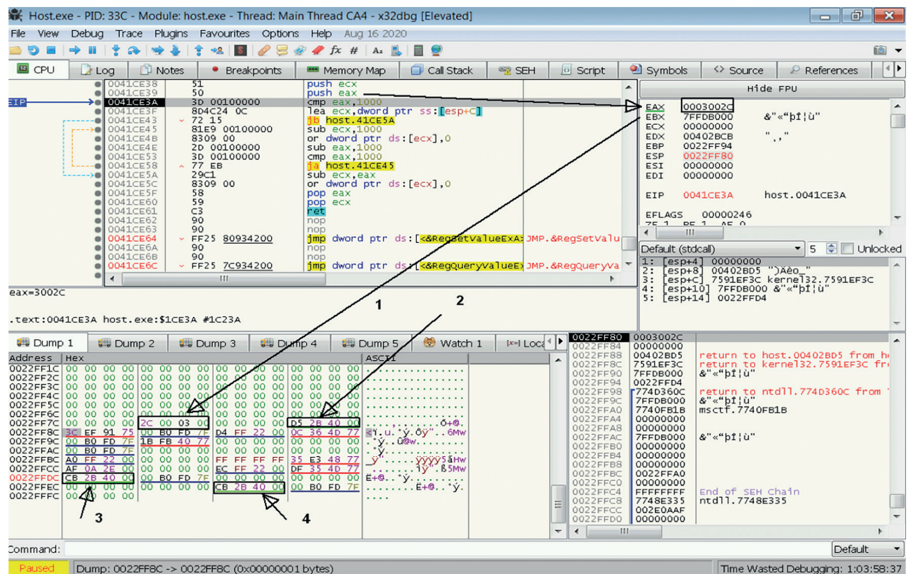


Figure 8. Debugger stopped in the start of the first call

Level B

```
.text:00401092    push    esi
.text:00401093    push    ebx
.text:00401094    mov     eax, 1434h
.text:00401099    call    sub_41CE38
.text:0040109E    sub     esp, eax
```

This is the main function for the handling of all input data (controls and associated payloads) as sent from the c2 and is where the most important data for forensic analysis begins. This function is called whenever new incoming data is detected in the sockets in level A. It is responsible for decrypting the payload and passing execution down to the next level depending on the control code.

Level C

Lower-level functions performing various tasks for processing control codes, called from level B. Let's go back to level B and take a closer look.

In Figure 9, on the next page, where the execution is halted at the main function's start, we will take a closer look at what the different interesting data observations mean. On the left side, where the c2 is visible, we can see TShark filtering and printing the NetWire TCP data sent from the c2. On the right side, we can see the debugger attached to the host process and with the stack visible in the lower "Dump 1" window. The important observations are:

- The socket descriptor (0x0184) used for this specific session. Address 0x1FFF60.
- The current control (0xA6) being processed. Address 0x1FFF64.
- The pointer to the payload (0x1FFF7D). Address 0x1FFF68.
- The length of the payload (0x06). Address 0x1FFF6C.
- The control (0xA6) in raw as copied verbatim from the socket. Address 0x1FFF7C.
- The encrypted payload (61 94 9E 29 BC 10) in raw as copied verbatim from the socket. Always null-terminated. Address 0x1FFF7D.

This is the main function for the handling of all input data (controls and associated payloads) as sent from the c2 and is where the most important data for forensic analysis begins.

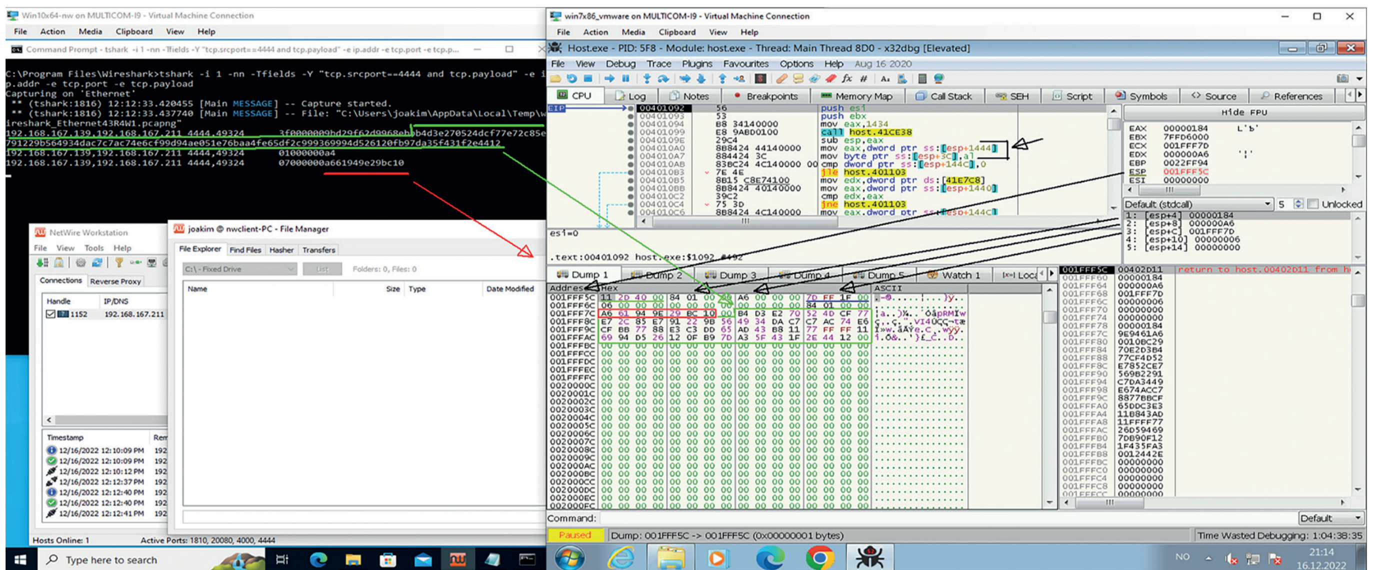


Figure 9. Debugger stopped at the function start before decryption

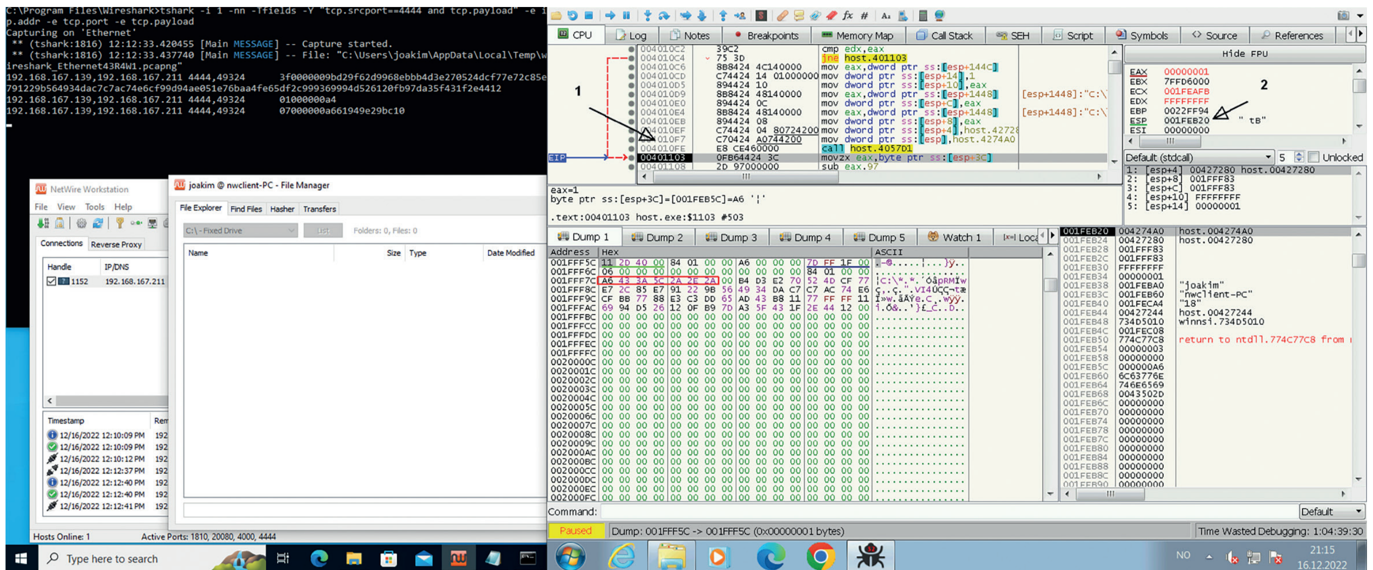


Figure 10. Debugger stopped right after decryption

Now we can verify that the encrypted data sent from the c2 is stored on the host's stack (red). In the TShark window we can see the previous control being an A4 (listing of drives), which does not have associated data. Moreover, we can also verify that the first auth packet sent (control 9B), which does not require decryption (the 0x10 byte session key is updated 0x1FFF9D - 0x1FFFAC), has portions of its payload data still visible in the slack area represented by the green.

In Figure 10, we see the debugger has stepped through the first few instructions of the main function and stopped at the exact location where the payload has been decrypted. The encrypted 6 bytes (61 94 9E 29 BC 10) are now decrypted (43 3A 5C 2A 2E 2A) in place at the same location and always with a null byte at the end. The current A6 control (File Explorer browse) was sent with the argument "C:*.*". Also, note the value for the stack pointer (0x1FEB20 which translates to offset 0x2BA20), which will be essential as an anchor in our parsing. All input coming from the c2 is stored and decrypted in the exact same way at the exact same location. This location is only overwritten when new input has arrived from the c2. Not all controls have associated payload and thus very often we will be able to find remnants of previous commands. This knowledge is advantageous in forensic analysis of the NetWire stack! The remainder of this function contains a huge jump table, essentially a switch statement for each of the control codes. Each jump destination will make one or more additional function calls (level C) which we will see leaves various artifacts on the stack. ▷

Where do we start?

Before delving into NetWire stack analysis, we need to establish an anchor that helps us identify a NetWire stack amongst other data. As digital forensics practitioners who often deal with disk images as opposed to live computers, we need to dig into places on the disk where memory may have ended up – for example, Windows swap (pagefile.sys). With a bit of trial and error we have found that the lower area of level B as mentioned above (towards address 0x1FEB20) is the ideal anchor. At a session start when connecting to the c2 a number of artifacts are stored in the stack in addresses that, to a varying degree, are re-used as buffers. Some of these buffers are never or rarely overwritten. We have defined four such addresses and are providing a complete table showing how and when each of them change. This table is available on GitHub at https://github.com/ArsenalRecon/NetWireStackForensics/blob/main/Artifacts-matrix.xlsx (sheet Buffers_matrix).

The addresses/offsets of the buffers we are interested in:

- buffer 0 0x1FEB60 / 0x2B60
- buffer 1 0x1FEBA0 / 0x2BA0
- buffer 2 0x1FECA4 / 0x2CA4
- buffer 3 0x1FEF00 / 0x2F00

During a session start (the initial auth to c2 when the 9B control is sent) the buffers are populated with the following data before being sent to the c2:

- buffer 0: hostname
- buffer 1: username
- buffer 2: a custom formatted string, representing a bitmask for the victim's OS. See sheet "Host_OS" at https://github.com/ArsenalRecon/NetWireStackForensics/blob/main/Artifacts-matrix.xlsx.
- buffer 3: a custom formatted string, representing various host details. See sheet "Host_details" at https://github.com/ArsenalRecon/NetWireStackForensics/blob/main/Artifacts-matrix.xlsx.

In Figure 11, we see how the anchor area and buffer 0 – 2 looks initially after the 9B control when a session is established.

Since buffer 3 will be part of what we decode, it is important to know exactly what it contains. The complete breakdown of the initial data in buffer 3 is shown in Figure 13.

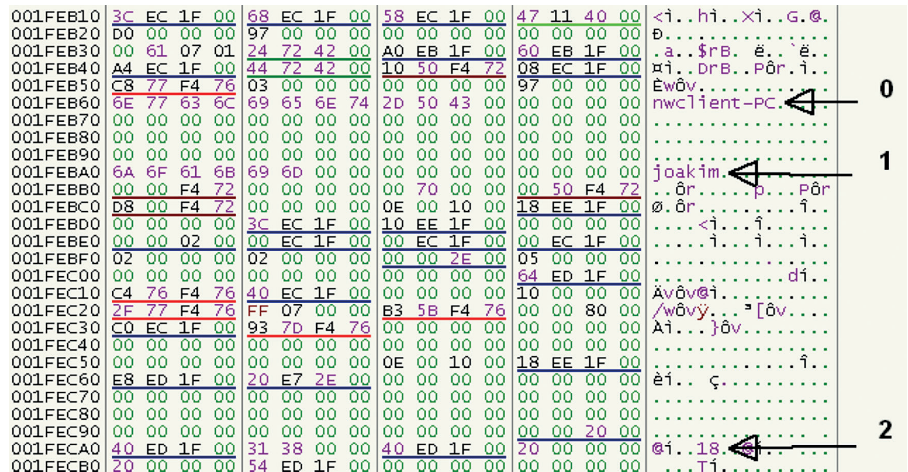


Figure 11. The anchor

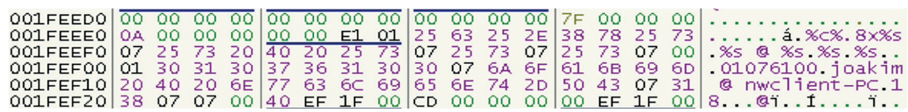


Figure 12. Buffer 3 right after a session is established

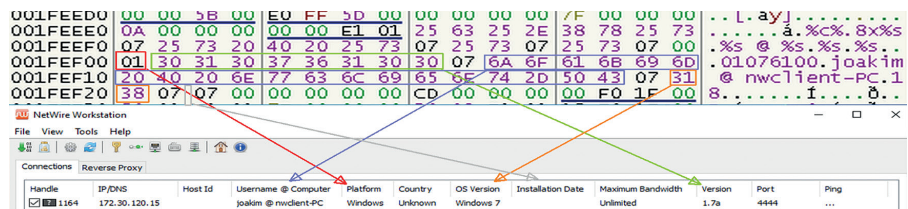


Figure 13. Buffer 3 containing host details

You can cross check this breakdown with the table in sheet "Buffers_matrix" in the spreadsheet at https://github.com/ArsenalRecon/NetWireStackForensics/blob/main/Artifacts-matrix.xlsx.

Referencing back to Figure 9, we can see the code at 004010A0 and 004010A7 where the control code is copied to 0x1FEB5C / 0x2B5C.

Since the hostname stays untouched for the lifetime of a session, we have a perfect spot to build a signature from, at the heart of the most compelling data. We can now build this byte level regex:

[\x97-\xE8] + .{3} + <hostname> + \x00\x00\x00

Which means:

- 1 byte for a valid control code
- 3 bytes of anything
- Variable byte length for the actual hostname formatted in hex
- 3 bytes of 00's

This is a rather simple regex, so any code involved in processing the hits will need additional validation. For this validation we have provided a proof of concept (POC) tool named NwStacks on GitHub at https://github.com/ArsenalRecon/NetWireStackForensics/tree/main/NwStacks. The memory address pointers of a NetWire process may change depending on how it was executed. However, the actual address, as seen in memory on the main thread's stack, is not crucial to our analysis as long as we can identify the data on disk and work our way from a translated file offset. The data points we will use always have a fixed relative distance between them, thus making identification and validation possible. We will call these "crucial data points" in the rest of this article. The fixed distance is caused by the stack frame for the given function call (except for calls going to external DLLs, which may differ between Windows versions) always having the same size regardless of the control codes and payloads being processed.

For most situations when a connection to a c2 has been established, the stack size will be a minimum of 0x34000 bytes. The stack size will expand to 0x35000 bytes in a few well-defined situations. Our research has shown that when the size has grown to 0x35000 bytes, the data in the first 0x1000 bytes contains nothing

useful. For the analysis described in this article, we have settled on using the size 0x34000 as a baseline. That means for any search hit on the byte regex described above, we will need to assume that the stack begins 0x2b5c bytes before the signature hit and then treat the following 0x34000 bytes as the full stack.

Validation

Our next step will be to validate certain crucial data points within the stack to determine their validity. In the remainder of this article, we will not refer to memory addresses unless specifically needed and instead refer to the file offsets in hex notation with dec in brackets. A small subset of the validation checks could look like this:

- The VA at 0x2B1C (11036) is valid
- The socket descriptors at 0x3F60 (16224) and 0x3F78 (16248) match (they might differ in the case of a reconnect)
- Pointers at 0x2B38 (11064), 0x2B3C (11068) and 0x2B40 (11072) are actually pointing to buffer 0, 1 and 2
- Valid controls at 0x2B5C (11100) and 0x3F7C (16252) + they match
- The length of payload at 0x3F6C (16236) is valid, considering the current control
- The string formatting definition prior to buffer 3 is valid, at 0x2EE8 (12008) (missing if the connection is reset and socket descriptor is FFFFFFFF)
- The VA at 0x3F50 (16208) is valid
- The VA at 0x3F5C (16220) is valid
- The value of 0003002C is present at 0x22FF80 (2293632)
- The VA 00402BD5 is present at 0x22FF88 (2293640)

What data in the stack is not helpful for our analysis?

- Addresses to heap memory
- Addresses to functions in DLLs loaded (kernel32.dll, msvcrt.dll etc)
- Socket descriptors (can be used as validation within a given stack, but may change for a different process or session)

Significant artifacts we can use in our analysis!

By now, we have a good understanding of the layout of the stack and how to find and validate that data - including data that needs to be treated carefully and what data we can ignore. The most significant artifacts which will be useful in our analysis are:

0)

The last return address pushed to the stack in level B (function 00401092) is stored at 0x2B1C (11036). See a full listing in our spreadsheet at https://github.com/ArsenalRecon/NetWireStackForensics/blob/main/Artifacts-matrix.xlsx. Most often (in the case of a ping) this value is 00401147 which originates from this code section:

```
.text:0040111F    movzx   edx, [esp+8+arg_30] ; jumptable 0040111D case 151
.text:00401124    mov     eax, [esp+8+s]
.text:0040112B    mov     [esp+8+lpValueName], 0 ; int
.text:00401133    mov     dword ptr [esp+8], 0 ; char *
.text:0040113B    mov     [esp+8+var_4], edx ; int
.text:0040113F    mov     [esp+8+Size], eax ; s
.text:00401142    call    sub_408B8F ; Send the default reply packet (command 97).
.text:00401147    jmp     loc_402B45 ; Nothing more to do, jump to end.
```

We can use this value for further validation and for the understanding of #6 described below.

1)

Buffer 1 - 3

(Based on the table within the sheet "Buffers_matrix" at https://github.com/ArsenalRecon/NetWireStackForensics/blob/main/Artifacts-matrix.xlsx) We can use this information in multiple ways. The presence of data (or the lack of data, represented by the amount of zeroed data), and even in combination. For example, some actions always leave specific data behind, whereas other actions never interact with these buffers. Some actions wipe the buffer with a specific amount of 00 before use, whereas others never wipe and simply overwrite and leave "slack" data behind. For some actions that leave data behind, the data format is unique enough to tell which control command it originated from.

As digital forensics practitioners who often deal with disk images as opposed to live computers, we need to dig into places on the disk where memory may have ended up.

2)

Timestamp

At 0x3C4C (15436), there is a 64-bit timestamp (in local time) written on every socket event. We can use this timestamp to determine from which point in time the stack is from. When a session is active, a function at 00408F43 contains a loop where the socket is checked with a 15-second timeout. The loop goes from 3 to 0; upon reaching 0, a 98 control is sent to the c2. On the c2 there are settings for answering a host ping and also to actively send ping requests to hosts. If both settings are deactivated on the c2, the timestamp is updated every 15 seconds. However, the timestamp will be updated even more frequently if other controls are actively sent from the c2. If both settings are deactivated on the c2, and a 98 control is triggered from the host, data in area 0x3AB4 - 0x3BEF (15028 - 15343) will be updated with irrelevant content.

3)

c2 hostname

At 0x3068 (12392), there may be a remnant from the initial connection to the c2, which is the c2 domain name. If this area is zeroed (the area covered by buffer 3), one of the actions listed within the sheet "Buffers_matrix" at https://github.com/ArsenalRecon/NetWireStackForensics/blob/main/Artifacts-matrix.xlsx must have been executed.

4)

Control

Part of the validation is to check 0x2B5C (11100) and 0x3F7C (16252). Because of the frequent ping requests, the current control is very often 97. ▷

5)
Payload
The payload bytes start at 0x3F7D (16253). This area is never wiped. Any data here passed as an argument to control will overwrite existing data with the size of the current payload and with a null termination. For some of the controls that do not have an argument (such as 97, the ping), the size is 0, and the first byte here is 00. In many cases, we will find the remnants of previous payloads here. The complete payload area stretches from 0x3F7D (16253) to 0x22FF7B (2293627), giving it a maximum size per block of 0x2FFFF (196607). The block size is relevant when a payload from the c2 has a size beyond 0x2FFFF (196607), in which case it is split up into 0x2FFFF (196607) blocks. This happens only with file uploads when the file size is larger than 0x2FFFF (196607). In the case of a file upload, the current block size may be found as a remnant in the lower 24 bits of the uint32 at 0x1FEAF8 / 0x2AF8, which we will take a closer look at later in this article. In the cases where no upload of significant size has arrived, there will often be remnants from the initial 9B control during the establishment of the connection to the c2. The payload in this situation is always of size 0x3E (62) and looks like random data caused by encryption (last byte at 0x3FBA (16314)). Note that the area from 0x22E150 (2285904) may contain remnants from the initial startup of the process (the "end" section described for the bitmaps at the top). For a full listing of all c2 actions and sample payloads decrypted, see https://github.com/ArsenalRecon/NetWireStackForensics/blob/main/Decrypted-payloads.txt.

Level C artifacts are caused by the lower-level functions when processing controls. This is the topmost area in the "Top" section described in the bitmaps. Data resolved in this location is, to a large degree, caused by calls to the winapi, and thus some addresses where data ultimately gets stored may be OS-dependent. Depending on how and where the stack was found, some of this area may be missing or overwritten. This is especially likely the deeper in the stack the data was stored. Treat the following items with care. In our tool NwStacks, this is dynamically resolved. The most significant File Explorer / Key Logs usage remnants are as follows.

Note
Ping requests overwrite portions of a 0x340 (832) byte area at 0x27F0 - 0x2B30 (10224 - 11056). Take this into consideration.

6)
File Explorer init (drive listing) at 0x1B0C (6924). This is caused by a call to kernel32!GetLogicalDriveStringA. The location is constant for all operating systems. It is one of the few situations where the full output data sent to the c2 is stored directly on the stack used by the main thread (due to its small size). Each entry representing a volume is 4 bytes long. The full data set size is 0x1AFC (6908). At 0x1AEC, there is a return address (VA) written of 004095E4. See our NwStacks tool for validation and detection.

7)
The 004095ED function, which is used for both regular file explorer browse (A6 control) and key logger reader/browse (CC control), leaves somewhat unique artifacts often visible in this larger area. The processing of these controls will return details from the file system (type, size, name, timestamp), and it uses a series of winapi calls to get the raw data before formatting it and sending it back to the c2. We will take a closer look at how it works, starting with the function epilogue (start):

```
.text:004095ED
.text:004095ED    push    edi
.text:004095EE    push    esi
.text:004095EF    push    ebx
.text:004095F0    sub     esp, 410h
```

Here we immediately see the workspace of this function being reserved as 410h. Effectively the esp has been reduced from 0x1FEB20 to 0x1FE700 after going through the epilogue.

Some actions always leave specific data behind, whereas other actions never interact with these buffers. Some actions wipe the buffer with a specific amount of 00 before use, whereas others never wipe and simply overwrite and leave "slack" data behind.

It turns out this function call chain will go quite deep and use the area stretching back to 0x1FDEE4. That leaves 0x1FEB20 - 0x1FDEE4 = 0xC3C bytes of stack data relating to this function's processing of file system data. As we will see, this information will help us further understand why certain data is found in this region of the stack. The actions performed in this function are:

A. Retrieve directory information based on buffer 2 - > kernel32!FindFirstFileA - > (ntdll!NtOpenFile + ntdll!NtQueryDirectoryFile.
B. Iterate through all items, and retrieve file system details - > kernel32!FindNextFileA
C. Extract the details such as attribute type, name, size, and the last write timestamp
D. Convert timestamp to UTC - > kernel32!FileTimeToSystemTime
E. Format the data in a special way as is expected on the c2 end - > msvcrt!_vsnprintf
F. Differentiate on files vs folders in how the formatting is done (folders don't have size.
G. Each item is formatted and prepared for output and is then copied to a separate heap memory location
H. When all items are done, send the result back to the c2

Now let's go through the most useful data that we can find here, starting with the topmost data (the deepest point on the stack that this function went to):

7a)
0x1F1C (7964)
As part of A) there will ultimately be a call to ntdll!_RtlGetFullPathName_Ustr which is needed for preparing the string before calling ntdll!NtOpenFile.

Deeper in the call chain within kernel32!FindFirstFileA there will ultimately be a call to ntdll!_RtlGetFullPathName_Ustr which is needed for preparing the string before calling.

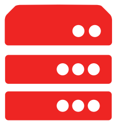
7b)
0x21F4 (8692)
As part of A) the call to ntdll!NtQueryDirectoryFile will leave a FILE_BOTH_DIR_INFORMATION struct of the target directory.

kernel32!FindFirstFileA - > ntdll!NtQueryDirectoryFile

It is important to note that in the case of a refresh/browse in File Explorer after an upload, the 3 timestamps except CreateTime are updated. ▷

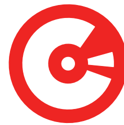
DIGITAL FORENSICS TOOLS
BY DIGITAL FORENSICS EXPERTS

Making Maximum Exploitation of Electronic Evidence More Accessible



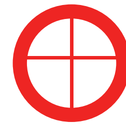
Arsenal Image Mounter

Mount the contents of disk images as "real" disks on Windows® with powerful and unique digital forensics functionality



Hibernation Recon

Reconstruct active memory and extract multiple types (and levels) of slack from Windows hibernation files



Registry Recon

Unlock the potential of huge volumes of Windows Registry data and see how Registries changed over time

"After many unsuccessful attempts to launch forensic images into virtual machines with a popular digital forensics tool, I decided to give Arsenal Image Mounter a try. I'm very glad I did, because I was able to virtualize forensic images from multiple suspects. AIM also bypassed Microsoft cloud account passwords within the virtual machines, so I was able to take valuable screenshots for the US Attorney. In addition, I have found AIM's multiple methods of Volume Shadow Copy exporting to be useful."

-- ICE/Homeland Security Investigation

"Hibernation Recon has become DoD's must-have tool for extracting digital artifacts from Windows hibernation files. Not only does Hibernation Recon properly reconstruct active memory for all versions of Windows when other tools fail, it is the only tool that extracts various types of "slack space", which has yielded critical forensic artifacts for DoD's foreign intelligence mission that could not have been obtained any other way."

-- United States Department of Defense

ArsenalRecon.com sales@ArsenalRecon.com @ArsenalRecon

7c)
0x24A4 (9380)
During B-C a temporary copy of the WIN32_FIND_DATAA structure for the next item is stored here. This copy may be partially overwritten with data from the ftLastWriteTime member.
(This only applies to a Windows 7 victim)

7d)
0x26FC (9980)
Last ret address pushed on the stack. When the function is done it will be 409A89 or 409A2F. 0x2704 (9988) / Control A6 or CC.

7e)
0x2730 (10032)
The format statements used when feeding the various data pieces through msvcrt!_vsnprintf.

7f)
0x2780 (10112)
During B) another copy of the WIN32_FIND_DATAA structure for the next item is also stored here.

7g)
0x28E4 (10468)
During E) and F) the ftLastWriteTime timestamp is taken from 0x1FE794 and outputs the string at 0x1FE8E4.

7h)
0x2904 (10500)
During G) – H) format full row for item.

8)
File Upload and Download.

Upload

At 0x2AEC (10988) the return address 41BCEB is pushed to the stack only in the case of a file upload. This is caused by the incoming AD control closing the file handle. The address originates from the function 0041BCA8, which is responsible for closing certain file handles. The only other case resulting in this code path being triggered is for downloads, but because the outgoing AD control is performed in a separate thread (not the main thread as with incoming controls), the address is not pushed to the stack used by the main thread. In the upload case, the memory address 0x1FEAF4 = 1 (0x2AF4 / 10996), and the lower 24 bits of uint32 at memory address 0x1FEAF8 (0x2AF8 / 11000) equals the size of the current/last chunk. The chunk size is written to the stack at 0041BCA8 during the processing of an incoming AC control at this code location:

```
.text:0041C8FB          mov     [esp+2Ch+Count],    eax ;     Count
```

Download

At 0x1FEA8 the return address 41C150 is pushed to the stack. This is caused by the AB control, which is effectively caused by the function located at 00401B66. Let's take a closer look at it:

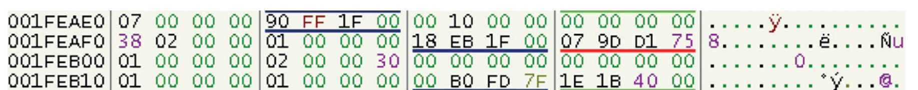
```
.text:00401B3B                  call    sub_40AC70
.text:00401B40                  mov     ecx, [esp+8+arg_1410]
.text:00401B47                  mov     [ecx+218h], eax
.text:00401B4D                  mov     [ecx+21Ch], edx
.text:00401B53                  nop
.text:00401B54
.text:00401B54  loc_401B54:             ; CODE XREF: sub_401092+934,Üëj
.text:00401B54                           ; sub_401092+A14,Üëj
.text:00401B54                  mov     eax, [esp+8+arg_1410]
.text:00401B5B                  mov     [esp+8+var_4], eax
.text:00401B5F                  mov     [esp+8+Size], offset sub_41C150
.text:00401B66                  call    sub_40B2AB
.text:00401B95                  jmp     loc_402B45
```

It turns out that the 41C150 address in the stack is written for uploads and downloads in the same location. Still, because a new thread is started for sending data back to the c2 with the download, the address stays here in the download scenario. For the upload scenario, the closing of a handle with an incoming AD control overwrites and leaves its unique fingerprint.
For the download scenario, memory address 0x1FEAF0 = 0 (0x2AF0 / 10992), and the 8 bytes at memory address 0x1FEAF8 (0x2AF8 / 11000) represents the 64-bit LastAccessTime timestamp of the downloaded file. The timestamp is caused by a call to GetFileattributesExA which generates a WIN32_FILE_ATTRIBUTE_DATA struct at memory address 0x1FEAEC (0x2AEC / 10988). This struct is almost completely overwritten soon after, with the exception of LastAccessTime member. In Figures 14, 15 and 16, let's just take a closer look at how it happens with snapshots when execution has been halted in debugger at 3 different locations from the above code section.

Fortunately, the artifact in the 0x20 byte area 0x1FEAE0 (0x2AE0 / 10976) – 0x1FEAFF (0x2AFF / 11007) which is unique to uploads and downloads is left untouched when a ping overwrites the surrounding area. Even a file explorer refresh/browse will not overwrite the bytes used in this area specific to the identification of upload/download. However, other controls/actions may overwrite and cause identification of upload/download specific to this location to be impossible. See our NwStacks tool for detection and validation.

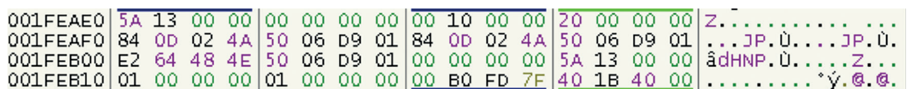
Note

If the socket descriptor equals 0xFFFFFFFF, it means the host is not connected to the c2. If the host still needs to connect to the c2, the buffer 0 – 4 will contain no useful information, and the control + payload area at 0x3F7C+ will be blank. If a previously established connection gets dropped, the area at 0x3F7C+ will contain the data last stored there.



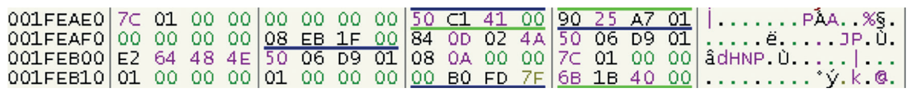
```
001FEAE0 07 00 00 00 90 FF 1F 00 00 10 00 00 00 00 00 00 .....ÿ..........
001FEAF0 38 02 00 00 01 00 00 00 18 EB 1F 00 07 9D D1 75 8.........ë....Ñu
001FEB00 01 00 00 00 02 00 00 30 00 00 00 00 00 00 00 00 ........0.......
001FEB10 01 00 00 00 01 00 00 00 00 B0 FD 7F 1E 1B 40 00 ..........°ý...@.
```

Figure 14. Stack snapshot when execution is halted at VA 00401B3B



```
001FEAE0 5A 13 00 00 00 00 00 00 00 10 00 00 20 00 00 00 Z...............
001FEAF0 84 0D 02 4A 50 06 D9 01 84 0D 02 4A 50 06 D9 01 ...JP.Ù....JP.Ù.
001FEB00 E2 64 48 4E 50 06 D9 01 00 00 00 00 5A 13 00 00 âdHNP.Ù.....Z...
001FEB10 01 00 00 00 01 00 00 00 00 B0 FD 7F 40 1B 40 00 ..........°ý.@.@.
```

Figure 15. Stack snapshot when execution is halted at VA 00401B40



```
001FEAE0 7C 01 00 00 00 00 00 00 50 C1 41 00 90 25 A7 01 |.......PÁA..%§.
001FEAF0 00 00 00 00 08 EB 1F 00 84 0D 02 4A 50 06 D9 01 .....ë.....JP.Ù.
001FEB00 E2 64 48 4E 50 06 D9 01 08 0A 00 00 7C 01 00 00 âdHNP.Ù.....|...
001FEB10 01 00 00 00 01 00 00 00 00 B0 FD 7F 6B 1B 40 00 ..........°ý.k.@.
```

Figure 16. Stack snapshot when execution is halted at VA 00401B95

Wrap-up with a demonstration

We have recreated a realistic user scenario with multiple file uploads and file deletion. A series of stack snapshots have been taken, one after each action from the c2, which we have made available at GitHub https://github.com/ArsenalRecon/NetWireStackForensics/tree/main/SampleStackSnapshots/win7-32 (article). All the files we used for uploads are available on our GitHub project at https://github.com/ArsenalRecon/NetWireStackForensics/tree/main/SampleFilesUploaded. Here is a summary of our operations:

1) File Explorer open c:\share\sample_uploads

2) Upload of the file "My savings plan for 2023 – v1.4 (12.30.2022).pdf" to c:\share\sample_uploads. File size is 54821 bytes

3) Execute a refresh in File Explorer to verify that the file is uploaded

4) Upload a second file "file_w_33_0x40.txt" to c:\share\sample_uploads
File size is 64 bytes.

5) Execute a refresh in File Explorer to verify that the file is uploaded

6) Ping sent from the c2

The c2 was configured not to answer or send pings to create some snapshots without ping overwriting anything. Instead, ping requests were actively sent from the c2 using the NetWire UI, so that snapshots could be taken to show what a ping overwrite looks like.

Now, we will not analyze each of the six stack snapshots here, as they are made available for anyone wanting to verify the results. But we will take snapshot three and verify our defined points of interest.

Snapshot 3

Step 1: Validation

```
00002B10 | 00 E0 FD 7F 00 00 00 00  00 00 00 00 ED 18 40 00 |  àý         í @
00002B20 | A4 EC 1F 00 7D FF 1F 00  04 02 00 00 98 FF 1F 00 | ¤ì  }ÿ       ÿ
00002B30 | FF FF FF FF 01 00 00 00  A0 EB 1F 00 60 EB 1F 00 | ÿÿÿÿ     ë  `ë
00002B40 | A4 EC 1F 00 44 72 42 00  10 50 25 73 08 EC 1F 00 | ¤ì  DrB  P%s ì
00002B50 | C8 77 10 77 03 00 00 00  00 00 00 00 A6 00 00 00 | Èw w        ¦
00002B60 | 6E 77 63 6C 69 65 6E 74  2D 50 43 00 00 00 00 00 | nwclient-PC
00002B70 | 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 |
00002B80 | 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 |
00002B90 | 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 |
00002BA0 | 6A 6F 61 6B 69 6D 00 00  00 00 00 00 00 00 00 00 | joakim
00002BB0 | 00 00 25 73 00 00 00 00  00 70 00 00 00 50 25 73 |   %s     p   P%s
00002BC0 | D8 00 25 73 00 00 00 00  0E 00 10 00 18 EE 1F 00 | Ø %s         î
```

Figure 17. Validation 1

```
00003F40 | 05 00 00 00 04 00 00 00  00 00 00 00 00 00 00 00 |
00003F50 | 9E 10 40 00 00 E0 FD 7F  00 00 00 00 27 2C 40 00 | ž @  àý     ',@
00003F60 | 80 01 00 00 7C FF 22 00  04 00 00 00 1B 00 00 00 | €   |ÿ"
00003F70 | 00 00 00 00 00 00 00 00  80 01 00 00 A6 43 3A 5C |         €   ¦C:\
```

Figure 18. Validation 2

```
00033F70 | 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 |
00033F80 | 2C 00 03 00 00 00 00 00  D5 2B 40 00 3C EF F9 76 | ,       Õ+@ <ïùv
00033F90 | 00 E0 FD 7F D4 FF 22 00  0C 36 11 77 00 E0 FD 7F |  àý Ôÿ"  6 w àý
00033FA0 | BC 53 3F 77 00 00 00 00  00 00 00 00 00 E0 FD 7F | ¼S?w          àý
00033FB0 | 00 00 00 00 00 00 00 00  00 00 00 00 A0 FF 22 00 |              ÿ"
00033FC0 | 00 00 00 00 FF FF FF FF  35 E3 0C 77 08 A2 0D 00 |     ÿÿÿÿ5ã w ¢
00033FD0 | 00 00 00 00 EC FF 22 00  DF 35 11 77 CB 2B 40 00 |     ìÿ" ß5 wË+@
00033FE0 | 00 E0 FD 7F 00 00 00 00  00 00 00 00 00 00 00 00 |  àý
00033FF0 | 00 00 00 00 CB 2B 40 00  00 E0 FD 7F 00 00 00 00 |     Ë+@  àý
```

Figure 19. Validation 3

- Signature was found at 0x2B5C, where we see an A6 control, then three nulls and the hostname.
- The VA at 0x2B1C is valid: 004018ED.
- The socket descriptors at 0x3F60 and 0x3F78 match: 0x0180.
- The pointers at 0x2B38 (0x1EFEBA0), 0x2B3C (0x1FEB60) and 0x2B40 (0x1FECA4) are as expected. Substitute 0x1FC000 from each and you'll see that they match what we defined as buffer 0, 1 and 2.
- The control at 0x2B5C and 0x3F7C match: 0xA6.
- The size of the payload at 0x3F6C looks valid: 0x1B.
- The formatting at 0x2EE8 is present: %c%.8x%s etc.
- The VA at 0x3F50 is valid: 0040109E.
- The VA at 0x3F5C is valid: 00402C27.
- At 0x33F80 we find the expected value: 0003002C.
- The VA at 0x33F88 is valid: 00402BD5

Step 2: Artifacts

a) Check payload. ▷

```
00003F50 | 9E 10 40 00 00 E0 FD 7F  00 00 00 00 27 2C 40 00 | ž @  àý     ',@
00003F60 | 80 01 00 00 7C FF 22 00  04 00 00 00 1B 00 00 00 | €   |ÿ"
00003F70 | 00 00 00 00 00 00 00 00  80 01 00 00 A6 43 3A 5C |         €   ¦C:\
00003F80 | 73 68 61 72 65 5C 73 61  6D 70 6C 65 5F 75 70 6C | share\sample_upl
00003F90 | 6F 61 64 73 5C 2A 2E 2A  00 3C 2F 54 79 70 65 2F | oads\*.* </Type/
00003FA0 | 43 61 74 61 6C 6F 67 2F  50 61 67 65 73 20 32 20 | Catalog/Pages 2
00003FB0 | 30 20 52 2F 4C 61 6E 67  28 65 6E 2D 55 53 29 20 | 0 R/Lang(en-US)
00003FC0 | 2F 53 74 72 75 63 74 54  72 65 65 52 6F 6F 74 20 | /StructTreeRoot
00003FD0 | 33 33 20 30 20 52 2F 4D  61 72 6B 49 6E 66 6F 3C | 33 0 R/MarkInfo<
00003FE0 | 3C 2F 4D 61 72 6B 65 64  20 74 72 75 65 3E 3E 2F | </Marked true>>/
00003FF0 | 4D 65 74 61 64 61 74 61  20 38 38 20 30 20 52 2F | Metadata 88 0 R/
00004000 | 56 69 65 77 65 72 50 72  65 66 65 72 65 6E 63 65 | ViewerPreference
00004010 | 73 20 38 39 20 30 20 52  3E 3E 0D 0A 65 6E 64 6F | s 89 0 R>>  endo
00004020 | 62 6A 0D 0A 32 20 30 20  6F 62 6A 0D 0A 3C 3C 2F | bj  2 0 obj  <</
00004030 | 54 79 70 65 2F 50 61 67  65 73 2F 43 6F 75 6E 74 | Type/Pages/Count
00004040 | 20 31 33 2F 4B 69 64 73  5B 20 33 20 30 20 52 20 |  13/Kids[ 3 0 R
00004050 | 39 20 30 20 52 20 31 31  20 30 20 52 20 31 33 20 | 9 0 R 11 0 R 13
00004060 | 30 20 52 20 31 35 20 30  20 52 20 31 37 20 30 20 | 0 R 15 0 R 17 0
00004070 | 52 20 31 39 20 30 20 52  20 32 31 20 30 20 52 20 | R 19 0 R 21 0 R
```

Figure 20. Payload section

In the section for raw input from the c2, starting at 0x3F7C we can see:

- The control is A6, which is a File Explorer browse command.
- Taking the defined size of 0x1B bytes from offset 0x3F7D gives us: C:\share\sample_uploads*.*.
- There is an expected null byte at 0x3F98 (0x3F7D + 0x1B).
- We can spot more data after the current payload, starting at 0x3F99; this strongly indicates some other action before the current 0xA6.

b) Check buffers 1, 2, and 3 according to our table in https://github.com/ArsenalRecon/NetWireStackForensics/blob/main/Artifacts-matrix.xlsx.

Buffer 1

In Figure 17, we see in buffer one at 0x2BA0 that there is a username visible. Also, other data remnants are visible after the username, indicating that a control that wipes the buffer has not been executed in this session.

Buffer 2

At offset 0x2CA4 we can see data in buffer 2. (Figure 21) Here we see a null-terminated string that perfectly matches the browse command seen in the previous step:

C:\share\sample_uploads*.*

But we can also see remnants of more data in the buffer:

avings plan for 2023 - v1.4 (12.30.2022).pdf

It seems clear that the argument here has been a file name, and according to the "Buffers_matrix" table at https://github.com/ArsenalRecon/NetWireStackForensics/blob/main/Artifacts-matrix.xlsx, there are only two possible options. A file rename or a file upload. Moreover, since 0x200 bytes was wiped earlier, this strongly suggests a file upload occurred.

We can clearly see the data from the initial connection has been wiped, leaving only a 0x30 byte.

```
00002CA0  40 ED 1F 00 43 3A 5C 73  68 61 72 65 5C 73 61 6D  @í  C:\share\sam
00002CB0  70 6C 65 5F 75 70 6C 6F  61 64 73 5C 2A 2E 2A 00  ple_uploads\*.*
00002CC0  61 76 69 6E 67 73 20 70  6C 61 6E 20 66 6F 72 20  avings plan for
00002CD0  32 30 32 33 20 2D 20 76  31 2E 34 20 28 31 32 2E  2023 - v1.4 (12.
00002CE0  33 30 2E 32 30 32 32 29  2E 70 64 66 00 00 00 00  30.2022).pdf
```

Figure 21. Buffer 2

Buffer 3

```
00002EE0  0A 00 00 00 00 00 36 01  25 63 25 2E 38 78 25 73        6 %c%.8x%s
00002EF0  07 25 73 20 40 20 25 73  07 25 73 07 25 73 07 00   %s @ %s %s %s
00002F00  30 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  0
00002F10  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
00002F20  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
00002F30  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
```

Figure 22. Buffer 3

We can clearly see the data from the initial connection has been wiped, leaving only a 0x30 byte. This artifact thus matches our suspected file upload from the previous step, according to the "Buffers_matrix" table at https://github.com/ArsenalRecon/NetWireStackForensics/blob/main/Artifacts-matrix.xlsx.

c) Check the socket timestamp.

```
00003C40  00 00 00 00 08 FD 1F 00  F8 FC 1F 00 40 02 30 0D      ý  øü  @ 0
00003C50  E6 1D D9 01 FF FF FF FF  FF FF FF 7F 00 00 00 00  æ Ù ÿÿÿÿÿÿÿ
```

Figure 23. Socket timestamp

At offset 0x3C4C, we find the 8 bytes 4002300DE61DD901 decoded as a 64-bit timestamp from little endian format, equals: 2023-01-01 13:36:28.5073984.

d) Verifying the upload artifact. We already have a strong indication that a file upload has occurred.

At offset 0x2AF8 (ref 8), we find the value 0xD625, which according to surrounding data at 0x2AEC and 0x2AF4, indicates the size of the last block for a file upload.

```
00002AE0  A0 74 42 00 FB EA 1F 00  FB EA 1F 00 EB BC 41 00   tB ûê  ûê  ë¼A
00002AF0  60 29 3B 75 01 00 00 00  25 D6 00 57 B0 25 86 5E  `);u    %Ö W°%†^
00002B00  E3 3E 28 35 19 7D B2 54  E8 C7 63 00 25 00 00 00  ã>(5 }²TèCc %
```

Figure 24. Size of uploaded file

As the file bytes start at the second byte in the payload from an AC control, we will need to copy 0xD625 bytes from offset 0x3F7E.

```
00011550  42 38 30 45 30 32 44 35  34 45 34 45 38 33 31 32  B80E02D54E4E8312
00011560  34 30 33 37 42 34 38 36  46 34 45 32 3E 5D 20 2F  4037B486F4E2>] /
00011570  50 72 65 76 20 35 32 36  36 35 2F 58 52 65 66 53  Prev 52665/XRefS
00011580  74 6D 20 35 32 32 31 31  3E 3E 0D 0A 73 74 61 72  tm 52211>>  star
00011590  74 78 72 65 66 0D 0A 35  34 36 34 32 0D 0A 25 25  txref  54642  %%
000115A0  45 4F 46 00 00 00 00 00  00 00 00 00 00 00 00 00  EOF
000115B0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
000115C0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
```

Figure 25. Payload end

We can see that the bytes ending at 0x115A2 indicate the end.

Here is a side-by-side comparison of the copied and original file bytes showing how the argument overwrote the first few bytes to the A6 control.

```
0000  3a 5c 73 68 61 72 65 5c 73 61 6d 70 6c 65 5f 75  :\share\sample_u   0000  25 50 44 46 2d 31 2e 37 0d 0a 25 b5 b5 b5 b5 0d  %PDF-1.7..%µµµµ.
0010  70 6c 6f 61 64 73 5c 2a 2e 2a 00 3c 2f 54 79 70  ploads\*.*.</Typ   0010  0a 31 20 30 20 6f 62 6a 0d 0a 3c 3c 2f 54 79 70  .1 0 obj..<</Typ
0020  65 2f 43 61 74 61 6c 6f 67 2f 50 61 67 65 73 20  e/Catalog/Pages    0020  65 2f 43 61 74 61 6c 6f 67 2f 50 61 67 65 73 20  e/Catalog/Pages
0030  32 20 30 20 52 2f 4c 61 6e 67 28 65 6e 2d 55 53  2 0 R/Lang(en-US   0030  32 20 30 20 52 2f 4c 61 6e 67 28 65 6e 2d 55 53  2 0 R/Lang(en-US
0040  29 20 2f 53 74 72 75 63 74 54 72 65 65 52 6f 6f  ) /StructTreeRoo   0040  29 20 2f 53 74 72 75 63 74 54 72 65 65 52 6f 6f  ) /StructTreeRoo
0050  74 20 33 33 20 30 20 52 2f 4d 61 72 6b 49 6e 66  t 33 0 R/MarkInf   0050  74 20 33 33 20 30 20 52 2f 4d 61 72 6b 49 6e 66  t 33 0 R/MarkInf
0060  6f 3c 3c 2f 4d 61 72 6b 65 64 20 74 72 75 65 3e  o<</Marked true>   0060  6f 3c 3c 2f 4d 61 72 6b 65 64 20 74 72 75 65 3e  o<</Marked true>
0070  3e 2f 4d 65 74 61 64 61 74 61 20 38 38 20 30 20  >/Metadata 88 0    0070  3e 2f 4d 65 74 61 64 61 74 61 20 38 38 20 30 20  >/Metadata 88 0
0080  52 2f 56 69 65 77 65 72 50 72 65 66 65 72 65 6e  R/ViewerPreferen   0080  52 2f 56 69 65 77 65 72 50 72 65 66 65 72 65 6e  R/ViewerPreferen
0090  63 65 73 20 38 39 20 30 20 52 3e 3e 0d 0a 65 6e  ces 89 0 R>>..en   0090  63 65 73 20 38 39 20 30 20 52 3e 3e 0d 0a 65 6e  ces 89 0 R>>..en
00a0  64 6f 62 6a 0d 0a 32 20 30 20 6f 62 6a 0d 0a 3c  dobj..2 0 obj..<   00a0  64 6f 62 6a 0d 0a 32 20 30 20 6f 62 6a 0d 0a 3c  dobj..2 0 obj..<
00b0  3c 2f 54 79 70 65 2f 50 61 67 65 73 2f 43 6f 75  </Type/Pages/Cou   00b0  3c 2f 54 79 70 65 2f 50 61 67 65 73 2f 43 6f 75  </Type/Pages/Cou
```

Figure 26. File upload comparison

e) Check lower-level artifacts from File Explorer and browse command.

(ref 6)

At 0x1AEC, we find the VA 004095E4, which is the last return address pushed on stack within the 00409508 function which is only called when processing the A4 control. At 0x1AFC we find the total size of the data which is 0xC. The array of detected volumes are found from 0x1B0C – 0x1B17.

```
00001AD0  11 00 00 00 00 00 00 00  11 00 00 00 11 00 00 00
00001AE0  00 00 00 00 00 00 00 00  94 FF 22 00 E4 95 40 00          ”ÿ" ä•@
00001AF0  80 01 00 00 A4 00 00 00  0C DB 1F 00 0C 00 00 00  €   ¤    Û
00001B00  00 00 00 00 00 00 00 00  00 00 00 00 41 3A 02 07              A:
00001B10  43 3A 03 07 44 3A 05 07  00 00 00 00 00 00 00 00  C:  D:
```

Figure 27. File Explorer detecting available drives

The decoded volume array;

A: – > Floppy Drive

C: – > Fixed Drive

D: – > CD-ROM Drive

(ref 7a)

At offset 0x1F1C we see the expected wchar equivalent of the browse argument (as prepared to NtOpenFile).

```
00001F10  48 84 10 77 00 00 00 00  80 73 02 00 43 00 3A 00  H„ w    €s  C :
00001F20  5C 00 73 00 68 00 61 00  72 00 65 00 5C 00 73 00  \ s h a r e \ s
00001F30  61 00 6D 00 70 00 6C 00  65 00 5F 00 75 00 70 00  a m p l e _ u p
00001F40  6C 00 6F 00 61 00 64 00  73 00 5C 00 2A 00 2E 00  l o a d s \ * .
00001F50  2A 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  *
```

Figure 28. Low level analysis of offset 0x1F1C

(ref 7b)

At offset 0x21FC we also find the FILE_BOTH_DIR_INFORMATION struct which includes the 4 $STANDARD_INFORMATION timestamps from the directory being browsed.

```
000021F0  FF 07 00 00 00 00 00 00  00 00 00 00 68 7A 02 CE  ÿ           hz Î
00002200  1C 1D D9 01 F3 CA 81 01  E6 1D D9 01 F3 CA 81 01    Ù óÊ   æ Ù óÊ
00002210  E6 1D D9 01 F3 CA 81 01  E6 1D D9 01 00 00 00 00  æ Ù óÊ   æ Ù
00002220  00 00 00 00 00 00 00 00  00 00 00 00 10 00 00 00
00002230  02 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
00002240  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
00002250  00 00 2E 00 00 00 61 00  48 EE 10 77 E8 FF 63 00    .    a Hî wèÿc
```

Figure 29. Low level analysis of offset 0x21FC

Full translation;

CreationTime: 2022-12-31 13:35:53.6698984

LastAccessTime: 2023-01-01 13:36:08.9107187

LastWriteTime: 2023-01-01 13:36:08.9107187

ChangeTime: 2023-01-01 13:36:08.9107187

EndOfFile: 0

AllocationSize: 0

FileAttributes: 16 (directory)

FileNameLength: 2

EaSize: 0 ▷

(ref 7d)

At 0x26fc we find the VA 409A89 and at 0x2704 we see the value A6, suggesting the low-level browse related artifacts belong to the A6 control.

```
000026F0  DD 55 3F 3D FE FF FF FF  27 DD E6 74 89 9A 40 00
00002700  CC E8 1F 00 A6 00 00 00  20 8C C8 01 A6 00 00 00
```

Figure 30. Low level analysis of offset 0x26FC

(ref 7f)

At 0x2780, we find a WIN32_FIND_DATAA struct for the last item in alphabetic order found in the directory browsed.

```
00002780  20 00 00 00 9C 6E D5 42  57 F4 D8 01 9C 6E D5 42      œnÕBWôØ œnÕB
00002790  57 F4 D8 01 50 33 DA 42  57 F4 D8 01 00 00 00 00  WôØ P3ÚBWôØ
000027A0  E0 5E 00 00 00 00 00 00  00 00 00 00 7A 7A 7A 2E  à^          zzz.
000027B0  74 78 74 00 67 73 20 70  6C 61 6E 20 66 6F 72 20  txt gs plan for
000027C0  32 30 32 33 20 2D 20 76  31 2E 34 20 28 31 32 2E  2023 - v1.4 (12.
000027D0  33 30 2E 32 30 32 32 29  2E 70 64 66 00 E8 1F 00  30.2022).pdf è
```

Figure 31. Low level analysis of offset 0x2780

Full translation;
FileAttributes: 32 (archive)
CreationTime: 2022-11-09 16:21:02.8045468
LastAccessTime: 2022-11-09 16:21:02.8045468
LastWriteTime: 2022-11-09 16:21:02.8357968
FileSizeHigh: 0
FileSizeLow: 24288
FileName: zzz.txt

(notice the slack data from the file name of a previous file passed into the struct)

(ref 7g)

At 0x28e4 we find the formatted string of LastWriteTime from the above struct:

09/11/2022 16:21:02.

```
000028E0  00 00 00 00 30 39 2F 31  31 2F 32 30 32 32 20 31      09/11/2022 1
000028F0  36 3A 32 31 3A 30 32 00  00 00 3E 00 38 0F C9 01  6:21:02    > 8 É
```

Figure 32. Low level analysis of offset 0x28E4

(ref 7h)

At 0x2904, we find the details of the current/last item formatted as returned to the c2;

32 zzz.txt 24288 09/11/2022 16:21:02

```
00002900  00 00 00 00 33 32 07 7A  7A 7A 2E 74 78 74 07 32      32 zzz.txt 2
00002910  34 32 38 38 07 30 39 2F  31 31 2F 32 30 32 32 20  4288 09/11/2022
00002920  31 36 3A 32 31 3A 30 32  07 00 2E 33 30 2E 32 30  16:21:02  .30.20
00002930  32 32 29 2E 70 64 66 07  35 34 38 32 31 07 30 31  22).pdf 54821 01
00002940  2F 30 31 2F 32 30 32 33  20 31 33 3A 33 36 3A 30  /01/2023 13:36:0
00002950  38 07 00 01 00 00 00 00  08 00 00 00 00 00 3E 00  8             >
```

Figure 33. Low level analysis of offset 0x2904

This field is of variable size, and just as we saw in the WIN32_FIND_DATAA struct above, slack data may be visible. Notice the 07 field separator that is heavily used in NetWire.

Summary snapshot 3

This stack snapshot is from around 2023-01-01 13:36:28.5073984.

When File Explorer was initialized, three volumes were detected: A:, C:, and D:.

The current A6 control is a File Explorer browse with the argument C:\share\sample_uploads*.* verified at three different levels.

A file ending with "avings plan for 2023 - v1.4 (12.30.2022).pdf" appears to have been uploaded to C:\share\sample_uploads\ with almost the entire content recoverable within the stack.

From low-level timestamps, the exact point in time appears to be 2023-01-01 13:36:08.9107187.

The last executed A6 control was executed sometime between 2023-01-01 13:36:08.9107187 and 2023-01-01 13:36:28.5073984.

A filesystem analysis has not been included here, but it would be yet another set of artifacts to compare the stack findings with. •

Snapshots

Our findings have been verified on Windows 7 32-bit and 64-bit, Windows 8.1 64-bit, and Windows 10 64-bit. A collection of stack snapshots for each OS is provided at https://github.com/ArsenalRecon/NetWireStackForensics/tree/main/SampleStackSnapshots. We have done some preliminary testing of the latest NetWire (version 2.1), and the methodology outlined here is still applicable to a large extent. There are changes, though - for example, the stack has shrunk to roughly 1/4 of the size in 1.7, and in the pe header the DYNAMIC_BASE setting is present in DllCharacteristics. Control codes look to be the same. We may address version 2.1 more specifically in the future.

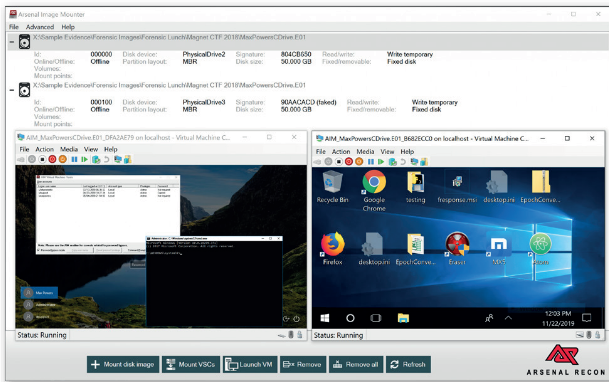
AUTHOR

***Joakim Schicht** is a Cloud Engineer at Vizrt, Digital Forensic Analyst at Joakim Schicht Consulting, and assists Arsenal with digital forensics software development and casework. Joakim enjoys writing code (https://github.com/jschicht) and deciphering the unknown.*

TECHNICAL EDITORS

***Mark Spencer, Arsenal** (https://ArsenalExperts.com) and **Brandon Levene, Lightforge Ventures** (https://www.lightforgeventures.com)*

ARSENAL RECON

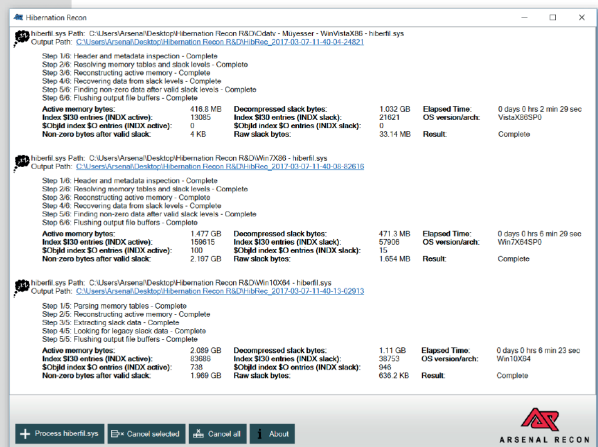


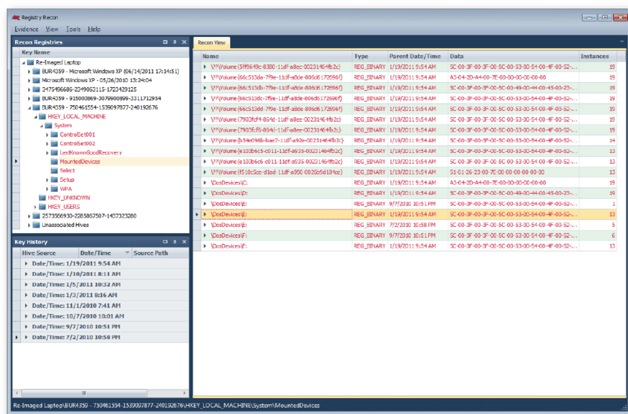
Arsenal Image Mounter

Arsenal Image Mounter mounts the contents of disk images as complete disks in Windows®. As far as Windows is concerned, the contents of disk images mounted by Arsenal Image Mounter are real SCSI disks, allowing users to benefit from disk-specific features like integration with Disk Manager, launching virtual machines (and then bypassing Windows authentication and DPAPI), managing BitLocker-protected volumes, mounting Volume Shadow Copies, and more.

Hibernation Recon

Hibernation Recon not only supports active memory reconstruction from Windows XP, Vista, 7, 8/8.1, 10, and 11 hibernation files, but also extracts massive volumes of information from the multiple types (and levels) of slack space that may exist within them. Additional features of Hibernation Recon include the automatic recovery of valuable NTFS metadata and parallel processing of multiple hibernation files.





Registry Recon

Registry Recon is not just another Registry parser. We developed powerful new methods to parse Registry data so that Registries which have existed on a Windows system over time can be rebuilt, providing unique insight into how Registry data has changed over time. Registry Recon provides access to an enormous volume of Registry data which has been effectively deleted, whether that deletion occurred due to benign system activity, malfeasance by a user, or even re-imaging by IT personnel.

Arm Yourself

Get the entire collection of Arsenal tools with an affordable subscription which:

- Enables the full functionality of all our tools with a single license
- Locks in a low price with discounts based on subscription length
- Provides easy access to new versions and support (no more SMS hassle)

Select the plan that's right for you at ArsenalRecon.com!

ArsenalRecon.com sales@ArsenalRecon.com @ArsenalRecon

Arsenal Consulting

— ARM YOURSELF —

DIGITAL FORENSICS & INFORMATION SECURITY

INTELLECTUAL PROPERTY THEFT, EVIDENCE SPOLIATION, INTERNET INVESTIGATIONS, FINANCIAL FRAUD, COMPUTER INTRUSION, EXTORTION

THE ARSENAL DIFFERENCE

Do you know where electronic evidence exists?

Without digital forensics, you don't. Whether you are involved in an internal investigation or ongoing litigation, traditional electronic discovery only scratches the surface when it comes to locating and understanding crucial electronic data. Arsenal clients have repeatedly found that utilizing digital forensics provides them with improved insight into internal matters and a significant advantage when it comes to ongoing disputes.

Team

The Arsenal team is led by President Mark Spencer, who has over twenty years of law-enforcement and private-sector digital forensics experience. We are forensic practitioners at our core and not your typical "computer guys." When faced with adversity, our personnel don't give up - they fight harder.

Approach

Arsenal specializes in applying the most powerful digital forensics tools and techniques to provide consulting services in high-profile and high-stakes cases. Our services, using methods acceptable in courts of law, result in clear and concise answers for our clients.

Experience

We have extensive experience with both criminal and civil litigation, having served as expert consultants and witnesses in state, federal, and international courts. Our hard-fought experience allows us to better understand clients' challenges and tailor the best solutions for them. In addition to providing consulting services, we develop digital forensics tools and train our peers.

As a direct result of Arsenal's work, we were able to obtain a quick and favorable resolution of a non-compete matter. You want the digital forensics experts at Arsenal on your side.

— Philip Y. Brown, Principal
Brown Counsel

I am a journalist who spent 19 months in jail due to a conspiracy within the Turkish Republic. Arsenal assisted in my acquittal thanks to their detailed forensic analyses and extraordinary efforts, and I thank them for once more proving that science will defeat lies.

— Barış Pehlivan, Investigative Journalist
Odatv

Our clients were facing the possible loss of their professional licenses. Arsenal helped us identify what data needed to be preserved then devised and ran searches that we used to prove our clients' innocence.

— Carole Cooke, Attorney
Todd & Weld LLP

Can you afford to trust anyone else?

Web: ArsenalExperts.com | Twitter: @ArsenalArmed | Email: info@ArsenalExperts.com