The Archaeology of Live Response

Dempster-Shafer Theory of Evidence in Digital Visual Media Forensics

The Fallacy of Passwords

Malware Classification: A Taxonomic Approach

RAM Memory Forensic Analysis
TEAM

Editor-in-Chief
Joanna Kretowicz
joanna.kretowicz@eforensicsmag.com

Editors:
Marta Sienicka
sienicka.marta@hakin9.com

Marta Strzelec
marta.strzelec@eforensicsmag.com

Marta Ziemianowicz
marta.ziemianowicz@eforensicamag.com

Senior Consultant/Publisher:
Paweł Marciniak

CEO:
Joanna Kretowicz
joanna.kretowicz@eforensicsmag.com

Marketing Director:
Joanna Kretowicz
joanna.kretowicz@eforensicsmag.com

DTP
Marta Strzelec
marta.strzelec@eforensicsmag.com

Cover Design
Hiep Nguyen Duc

Publisher
Hakin9 Media Sp. z o.o.
02-676 Warszawa
ul. Postępu 17D
Phone: 1 917 338 3631

www.eforensicsmag.com
DEAR READERS,

Spring is finally here, and we’re ready with the new issue. In this one we’ll spend some time in the live analysis realm, and we think you’ll enjoy taking the journey with us.

The issue starts off with Tom Sela from illusive networks talking about live response. The whole article is available in the preview as well, so don’t miss out! Then we’ll take a little look at how transfer backdoor payloads with Damon Mohammadbagher, and consider malware naming schemes with Fernando Mejia.

After that we’re diving deep into forensic investigations: first, we’ll learn everything there is to know about examining Edge’s InPrivate mode, and Eliézer Pereira will show us how to perform RAM live analysis from start to finish, and finishing off the first half of the issue you’ll be able to enjoy a piece on how Dempster-Shafer’s Theory of Evidence can be applies to video forensics - a must read!

Rob Sommerville makes a guest appearance with his column in the issue as, before the second half of the magazine where you’ll be able to read about forensic case management, tools, techniques, and tricks, as well as analyze a sample malware attack with a social engineering twist, and ponder on passwords and what’s wrong with them.

We appreciate your feedback at all times, so if you have any comments or suggestions, do let us know! You can find us on social media, or write us an email - everyone’s welcome!

Enjoy your reading!
Marta Strzelec
and the eForensics Editorial Team
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>The Archaeology of Live Response: Examining the artifacts</td>
<td>5</td>
</tr>
<tr>
<td>by Tom Sela, Head of Security Research, illusive networks</td>
<td></td>
</tr>
<tr>
<td>Transferring Backdoor Payloads By ICMPv4 Traffic</td>
<td>13</td>
</tr>
<tr>
<td>by Damon Mohammadbagher</td>
<td></td>
</tr>
<tr>
<td>Malware Classification: a Taxonomic Approach</td>
<td>23</td>
</tr>
<tr>
<td>by Fernando Mejia</td>
<td></td>
</tr>
<tr>
<td>Forensic Analysis of Edge Browser InPrivate Mode</td>
<td>31</td>
</tr>
<tr>
<td>by Shumaila Alam, M.Ammar Aziz, Waseem Iqbal</td>
<td></td>
</tr>
<tr>
<td>RAM Memory Forensic Analysis</td>
<td>51</td>
</tr>
<tr>
<td>by Eliézer Pereira</td>
<td></td>
</tr>
<tr>
<td>On the Role of Dempster-Shafer Theory of Evidence in Digital Visual Media Forensics</td>
<td>61</td>
</tr>
<tr>
<td>by Raahat Devender Singh and Naveen Agganwal</td>
<td></td>
</tr>
<tr>
<td>Rob's Column</td>
<td>70</td>
</tr>
<tr>
<td>by Rob Sommerville</td>
<td></td>
</tr>
<tr>
<td>Deep Dive into digital forensic case management</td>
<td>73</td>
</tr>
<tr>
<td>by Washington Almeida</td>
<td></td>
</tr>
<tr>
<td>Deep Dive into Digital Forensics Case Management Tools, Techniques, Tricks, and Procedures</td>
<td>82</td>
</tr>
<tr>
<td>by David Capote</td>
<td></td>
</tr>
<tr>
<td>Social engineering and malware: basic attack scenario and analysis</td>
<td>87</td>
</tr>
<tr>
<td>by Paulo Henrique Pereira, Renato Basante Borbolla, Thiago Geronimo Ferreira, Rubens Louro Vieira</td>
<td></td>
</tr>
<tr>
<td>Fallacy of Passwords and Password Alternatives</td>
<td>94</td>
</tr>
<tr>
<td>by Dhiresh Salian</td>
<td></td>
</tr>
</tbody>
</table>
The Archaeology of Live Response: Examining the artifacts

by Tom Sela, Head of Security Research, illusive networks

As a security researcher and a part time Incident Response (IR) analyst, appreciating the finer details becomes paramount. The role requires an understanding of an attacker’s actions on compromised machines via ongoing research. A typical research process requires examining hundreds or even thousands of artifacts to find the needle in the haystack.

But what if not all artifacts are created equal? What if specific artifacts enable the identification of attackers’ activities quicker and more accurately? This article reveals the retrieval of two artifacts identified as being significantly helpful when solving incidents: command prompt history and console output.

To contextualize the scenario, an automatic system has been developed that collects real-time forensic information from compromised machines. The collection and analysis of such information is called "Live Response", and the goal of this process is to identify and analyze the incident's trigger as quickly as possible. In Live Response, investigators capture volatile information that would not normally be
present in a postmortem investigation.

Live Response holds many challenges for investigators. The fact that artifacts are being discovered from a live running system, with an active adversary, presents the following issues:

- Ability to remotely access and run code on the compromised machine.
- To remain unnoticed by the attacker, the information-gathering process must be quick and stealthy.
- Operations must not "jeopardize the crime scene." (This is known as Locard’s Exchange Principle.)

A SANS Institute whitepaper lists examples of commonly collected artifacts in Live Response. These include running processes, running services, current login information, open files, open shares, open handles, and so on.

While building the forensic-collection tool, several artifacts were discovered that did not appear in the whitepaper’s list. Having investigated a wide range of attacks, it was apparent that attackers commonly use consoles (for example, Windows Command Prompt) to execute native commands as well as malicious code written in PowerShell, Python, and other scripting languages. This makes command-line history and console-output extremely valuable when examining compromised machines. These artifacts can be retrieved without dumping memory or performing code injections.

**Getting history of Windows Command Prompt (cmd.exe)**

**Existing tools**

The command history of an attacker’s console is extremely valuable in Live Response. Unlike the Linux shell (which stores command history in a file on disk, making the task of fetching history trivial), Windows Command Prompt history resides only in memory.

The only native Windows option to retrieve command history is Doskey.exe. From host memory, Doskey retrieves all commands executed on a cmd.exe console. Doskey’s major limitation is that it is called from a Command Prompt session and can only retrieve commands executed during that session. As in, calling Doskey from cmd.exe (PID 1337) retrieves only the history of cmd.exe (PID 1337).
As Doskey cannot retrieve the Command Prompt history of an external console, it can’t be used as a Live Response tool. It is worth mentioning that the volatility framework offers a useful plugin called cmdscan. The Cmdscan plugin searches process memory (csrss.exe or conhost.exe - depending on OS version) for commands executed via a console. Unfortunately, as with all volatility plugins, it requires a memory dump as input. As memory-dump creation requires a unique tool, is labor-intensive, and takes time, Cmdscan cannot be used as part of Live Response. Our goal is to retrieve an attacker’s executed commands while the attack is underway.

After having no luck finding a native solution, a decision was made to dive into the task hands-on.

**Reverse-engineering Doskey**

Doskey was revisited and reverse-engineered, finding two valuable kernel32 functions that are undocumented on MSDN:

- **GetConsoleCommandHistoryLengthW** - Retrieves the buffer size of the Command Prompt history, expects a console process's name as input.

- **GetConsoleCommandHistoryW** - Retrieves Command Prompt history as a null-delimited buffer of strings, expects the buffer size and process name as input.
The code below extracts the command history, similar to the operation of doskey.exe:

```
static const std::wstring COMMAND_LINE_PROCESS_NAME = L"cmd.exe";

// Retrieves the buffer size of the command line history
// Input - Process name (i.e "cmd.exe")
// Output - Buffer size in bytes (not wchar_t), zero in case of failure
DWORD history_buff_size = pGetConsoleCommandHistoryLengthW((LPWSTR)COMMAND_LINE_PROCESS_NAME.c_str());
if (0 == history_buff_size) {
    // No console history
    return false;
}

// history_buff_size is returned in bytes
// convert "byte count" to "wchar_t count"
size_t history_buff_len = history_buff_size / sizeof(wchar_t);
std::vector<wchar_t> output_history_buff(history_buff_len);

// Retrieves the command line history buffer
// Input - Buffer size in bytes, and process name
// Output - Null delimited buffer of history string commands
if (0 == pGetConsoleCommandHistoryW(output_history_buff.data(),
    history_buff_size,
    (LPWSTR)COMMAND_LINE_PROCESS_NAME.c_str())) {
    logMessage(L"Failed to get console history\n");
    return false;
}
```

After execution of GetConsoleCommandHistoryW, output_history_buff holds a null-delimited buffer with the Command Prompt's entire command history.
Attaching to the attacker's console

For Live Response, the command history from the attacker's console had to be obtained, requiring the application be ‘attached’ to it rather than any other console. When calling a process from an ‘unattached’ console, the two undocumented Doskey functions operate similarly to the following Windows API:

- **GetConsoleHistoryInfo** - "retrieves the history settings for the calling process's console".

The function's MSDN page states that if the calling process is not a console process, the function fails and sets the last error to **ERROR_ACCESS_DENIED**.

The following two documented Kernel32 APIs enable one to overcome this obstacle:

- **FreeConsole** - Detaches the calling process from its console
- **AttachConsole** - Attaches the calling process to the console of a specified process

These APIs should be executed prior to **GetConsoleCommandHistoryLengthW** and **GetConsoleCommandHistoryW**.

```c
FreeConsole();
bool res = AttachConsole(attacker_pid);
```

Lastly, the following code can be used to print the commands from `output_history_buff`.

```c
logMessage(L"Process Command History\n");

unsigned int cur_location = 0; // current location in the history buffer
unsigned int command_counter = 0; // history commands counter
while (cur_location < history_buff_len)
{
    // point to the current command string in output_history_buff
    std::wstring cur_history_command(
        std::wstring(output_history_buff.data()) + cur_location);
    logMessage(L"H%d: %s\n", command_counter,
        cur_history_command.c_str());
    // update location to the end of the current command string
    cur_location += cur_history_command.size() + 1;
    // increase command counter
    command_counter++;
}
```
Running the code above on the console in Figure 1 results in the following output:

<table>
<thead>
<tr>
<th>Process Command History</th>
</tr>
</thead>
<tbody>
<tr>
<td>H0: cd c:\</td>
</tr>
<tr>
<td>H1: dir</td>
</tr>
<tr>
<td>H2: whoami</td>
</tr>
<tr>
<td>H3: net use</td>
</tr>
<tr>
<td>H4: net group &quot;domain admins&quot; /domain</td>
</tr>
<tr>
<td>H5: cls</td>
</tr>
<tr>
<td>H6: doskey /history</td>
</tr>
</tbody>
</table>

**Getting console output**

The ability to retrieve the Command Prompt history of `cmd.exe` using [AttachConsole](#) fuelled greed; the console output also became sought after.

Surprisingly, this was simpler than retrieving command history. Once the application is ‘attached’ to the attacker’s console, the Windows [GetStdHandle](#) API obtains a handle to the specified standard input, output, or error. In this case, the `const STD_OUTPUT_HANDLE` was used to obtain a handle to the standard output.

```c
HANDLE hConsoleOutput = GetStdHandle(STD_OUTPUT_HANDLE);
if (INVALID_HANDLE_VALUE == hConsoleOutput)
{
    logMessage(L"GetStdHandle failed with %d\r\n", GetLastError());
    return false;
}
```

The next phase is to retrieve information about the process's console screen buffer using the Windows [GetConsoleScreenBufferInfo](#) API. The API returns the screen buffer's attributes in the `CONSOLE_SCREEN_BUFFER_INFO` structure. This information will be used in order to read data of the right size from the console buffer.

```c
CONSOLE_SCREEN_BUFFER_INFO lpConsoleScreenBufferInfo;

// Get the current screen buffer size and window position.
if (0 == GetConsoleScreenBufferInfo(hConsoleOutput,  
    &lpConsoleScreenBufferInfo))
{
    logMessage(L"GetConsoleScreenBufferInfo Failed Error - (%d)\r\n",  
        GetLastError());
    return false;
}
```
The final stage is to read the console’s output with the Windows `ReadConsoleOutput` API. The function retrieves a 2-dimensional array of `CHAR_INFO` values that represents the screen buffer.

```c
// The read coordinates of the upper-left cell of the screen buffer.
COORD dwBufferCoord = { 0, 0 }; 
// The size of the output screen buffer (lpBuffer)
COORD dwBufferSize = { lpConsoleScreenBufferInfo.dwSize.X, lpConsoleScreenBufferInfo.dwSize.Y }; 
// The current screen buffer rectangle
SMALL_RECT lpReadRegion{ 0, 0, lpConsoleScreenBufferInfo.dwSize.X, lpConsoleScreenBufferInfo.dwSize.Y }; 
// The output screen buffer
std::vector<CHAR_INFO> lpBuffer(dwBufferSize.Y * dwBufferSize.X);

BOOL fSuccess = ReadConsoleOutput(hConsoleOutput, // An handle to the console screen buffer
lpBuffer.data(), // A pointer to a destination buffer
dwBufferSize, // The size of the lpBuffer parameter
lpBufferCoord, // The read coordinates of the upper-left cell
lpReadRegion); // The current screen buffer rectangle

if (FALSE == fSuccess) {
    logMessage(L"ReadConsoleOutput failed - (%d)\r\n", GetLastError());
    return false;
}
```

The following code was used to print the output and skip empty lines:

```c
logMessage(L"Console Output Buffer\r\n");
// The current line number in the console buffer rectangle
CHAR_INFO *cur_line_location = NULL;
for (int line_number = 0; line_number < dwBufferSize.Y; line_number++)
{
    // Current line's position at lpBuffer
    cur_line_location = lpBuffer.data() + (line_number * dwBufferSize.X);
    // The current line string of the console buffer rectangle
    std::wstring curr_line;
    for (int j = 0; j < dwBufferSize.X; j++)
    {
        // Add current char to the current line
        curr_line += (wchar_t)cur_line_location[j].Char.UnicodeChar;
    }
    // Skip lines that are composed only from space chars
    std::wstring trimmed_line = trimStr(curr_line);
    if (!trimmed_line.empty()) {
        logMessage(L"L%d: %s\r\n", line_number, curr_line.c_str());
    }
}
```

Running the above code on the following console’s pid, generated the output below. Console:

![Figure 3 Command Prompt Example](image-url)
The same code is able to retrieve the output on any kind of console, including PowerShell and Python.

**Wrapping up**

Live Response can be used to solve incidents quickly and efficiently. Among the artifacts collected, some can provide more value to IR analysts than others. This exercise concluded that the Command Prompt's history and console output are highly valuable, recommending more analysts start to collect them as part of forensic investigations. There may be additional artifacts that help Live Response, and the community should be encouraged to share any other relevant information.

*The full source code is available for download [here](#). Find out more about [illusive networks](#).*

---

**About the Author: Tom Sela**

Tom Sela is Head of Security Research at [illusive networks](#), specializing in Reverse Engineering, Malware Research, and OS internals. Prior to joining illusive, Tom lead the Malware Research team at Trusteer (acquired by IBM). Tom majored in Computer Science at Ben-Gurion University and studied at the Israeli Naval Academy, University of Haifa.