Chracer: Memory analysis of Chromium-based browsers

By:
Geunyeong Choi, Jewan Bang, Sangjin Lee, Jungheum Park

From the proceedings of
The Digital Forensic Research Conference
DFRWS APAC 2023
Oct 17-20, 2023

Forensic Science International: Digital Investigation 46 (2023) 301613

Contents lists available at ScienceDirect

Forensic Science International: Digital Investigation

journal homepage: www.elsevier.com/locate/fsidi

DFRWS 2023 APAC - Proceedings of the Third Annual DFRWS APAC

Chracer: Memory analysis of Chromium-based browsers

Geunyeong Choi [a], Jewan Bang [b], Sangjin Lee [a], Jungheum Park [a,*]

[a] School of Cybersecurity, Korea University, 145 Anam-Ro, Seongbuk-Gu, Seoul, South Korea
[b] Cyber Investigation Bureau, National Office of Investigation, Korean National Police Agency, Seoul, South Korea

ARTICLE INFO

Keywords:
Digital forensics
Volatile data
Memory forensics
Web browser
User activity
Counter anti-forensics

ABSTRACT

The web browsing activities of a user provide useful evidence for digital forensic investigations. However, existing analysis techniques that aim to analyze local artifacts (e.g., history and cache) cannot find useful data (e.g., visited URLs) if a user accesses the web using private or secret mode. Hence, string-searching and pattern-matching techniques have been proposed and used to examine user activities from a memory dump. These simple techniques are useful for identifying individual URLs visited in both normal and private modes. However, since a piece of individually detected data does not have context on how it is created, additional analysis efforts are required to properly interpret the meaning of the data. This paper proposes *Chracer*, a practical methodology for extracting forensically meaningful information from the virtual memory of a Chromium-based browser by systematically discovering objects of web browsing-related classes. Moreover, a proof-of-concept tool developed based on the proposed methodology demonstrates that users' web browsing-related artifacts can be extracted effectively from the virtual memory of any Chromium-based browser, such as Google Chrome, Microsoft Edge and Brave.

1. Introduction

The number of devices that use the Internet, including computers, smartphones, in-car infotainment systems, and Internet of Things (IoT) devices, is increasing rapidly. Today, individuals not only acquire new information through the web, the largest Internet service, but they also create new information themselves and post it online. Thus, analysts analyze the web browsing history recorded by web browser applications to examine users' web browsing activities. Because web browsing history contains various types of information related to visited Uniform Resource Locators (URLs), visit times, referrers, etc., history is useful in reconstructing a user's activity or investigating initial access to a cybersecurity incident.

Analysts have typically examined users' web browsing activities by analyzing local artifacts, such as history and cache. Unfortunately, existing local artifact-based techniques cannot find useful web browsing-related artifacts when users access the web using a private (or secret) mode, which provides a feature that leaves no web browsing-related data in a file system. In addition, many privacy-enhanced applications do not store sensitive data on nonvolatile storage devices but are only temporarily loaded and used in the memory. This makes memory forensic techniques increasingly necessary and important because data loaded on only the memory can be obtained from a memory dump. Volatility (Volatility Framework), the leading memory analysis framework, aims to analyze kernel objects managed by operating systems such as *EPROCESS* of Windows or *struct task_struct* of Linux. The virtual memory of processes can be analyzed using the Volatility framework; however, analysis of an application's virtual memory has mostly been performed using string-searching or pattern-matching techniques. These simple techniques are useful for identifying individual URLs visited in both normal and private modes from the virtual memory byte streams of a web browser application. However, search results do not provide context (Garcia, 2007) such as how they are created; therefore, additional analysis efforts are required to properly interpret the meaning of the data. Search results that do not have meaning can render them less useful.

This paper proposes *Chracer*, a methodology for extracting forensically meaningful information to analyze the virtual memory of Chromium-based web browser applications using an object layout. The object layout describes how the fields (member variables) of classes or structures are arranged in memory (e.g., '*int a*' and '*int b*' fields are located at offsets 0 and 4, respectively), and the object is the mapping of the field values in the memory based on the object layout. The proposed methodology is summarized as follows: (1) identifying the classes and

* Corresponding author.
E-mail addresses: geunyeong@korea.ac.kr (G. Choi), jwbang@police.go.kr (J. Bang), sangjin@korea.ac.kr (S. Lee), jungheumpark@korea.ac.kr (J. Park).

https://doi.org/10.1016/j.fsidi.2023.301613

Available online 13 October 2023

structures that represent a browser, tab, tab group, visited URL, etc. in the source code of the Chromium project; (2) generating the object layout of such classes and structures, and scanning a memory dump to find *Browser* objects as the starting points of memory analysis and consequently reveal Chromium-based user activities; (3) finally, by discovering a *Browser* object, it is possible to interpret meaningful fields and explore all retrievable subobjects using pointer variables to extract forensically meaningful information, such as opened tabs and visited URLs.

This study has the following contributions.

- Identify forensically meaningful classes and structures relevant to web browsing activities by inspecting the source code of the Chromium project in Windows.
- Propose a methodology to automatically generate version-specific object layouts of Chromium and to effectively detect web browsing-related objects from virtual memory.
- Overcome the limitations of string-searching and simple pattern-matching-based memory analysis techniques by extracting contextual information about web browsing activities.
- Demonstrate the usefulness of the proposed methodology through practical experiments with Google Chrome, Microsoft Edge and Brave.
- Provide a proof-of-concept tool implemented based on the proposed methodology.

This paper is organized as follows: Section 2 reviews prior studies about web browser forensics and application memory forensics, and Section 3 provides background knowledge on virtual memory and object layout. Sections 4 and 5 introduce Chromium's classes relating to web browsing activities and the proposed memory analysis methodology. Section 6 demonstrates the usefulness of the proposed methodology by testing it on the Chromium browser. Section 7 discusses the result to apply the methodology to Google Chrome, Microsoft Edge and Brave, the comparison with a 'Restore Window' feature, the possibility of carving objects from freed space in a physical memory dump, and the limitation of our work. Finally, Section 8 concludes this work and proposes directions for future work.

2. Related work

2.1. Memory analysis of web browser-related activities

A chromehistory (superponible), one of the Volatility framework plugins, extracts web browsing history from a virtual memory of Google Chrome by discovering record structures of the SQLite database. ChromeRagamuffin (cube0x8), a Volatility2 plugin, scans Google Chrome's virtual memory to obtain a list of visited URLs. However, it does not extract whether the browser is in private mode, tab group information, Secure Socket Layer (SSL) certificate information. Objects of *Browser* class need to be set as starting points, in order to associate visited URLs with additional useful information.

Dija et al. (2021) proposed a methodology to extract words from memory, that a user searched for on the Internet. For an experiment, they searched some words using web browsers and discovered those words from a memory dump. After discovering searched words, they proposed an algorithm to find the searched words by analyzing byte patterns nearby them. They utilized URL parameter patterns such as 'search_query=' or 'search?q=' to find search terms.

Alfosail and Norris (2021) proposed a method for performing memory forensics on the Tor browser. They used the Volatility framework to identify Tor browser processes and Tor network connection and utilized pattern-matching techniques to analyze traces of access to onion sites by extracting URLs from the Tor browser's virtual memory. They also attempted to determine the actual access of the user by discovering HyperText Transfer Protocol (HTTP) requests and responses data.

Hariharan et al. (2022) conducted a memory forensic analysis of the portable web browsers' private mode that leaves no traces on the filesystem. They ran an executable file stored on an external storage device, such as a Universal Serial Bus (USB) flash drive, without installing it on the operating system. When using the private mode, no traces using a web browser are stored in history, cache, etc. They found traces left in memory after using web services such as Facebook and YouTube on Brave, Tor, Vivaldi, and Maxthon browsers. They showed that traces could be found using the yarascan plugin (Volatility plugin - yarascan) and it is possible to retrieve traces of a user's web browsing.

Iqbal et al. (2022) used Google Meet, an online meeting service using a web browser, in Google Chrome, Firefox, and Microsoft Edge, and identified online meeting-related information from a memory dump. They showed that it is possible to obtain the email addresses, meeting room addresses, and sent and received chat messages of users participating in Google Meet from the web browser's memory.

2.2. Object layout based memory forensics of user applications

Fernández-Álvarez and Rodríguez (2022) proposed a method to obtain memory artifacts for the Telegram application on Windows. They analyzed Telegram's source code to identify classes that are related to a chat message and modeled their reference relationships with each other. Since Telegram uses the Qt framework to provide a Graphical User Interface (GUI), in their study, objects such as *QString* were found in Telegram's virtual memory and reconstructed.

Manna et al. (2022) proposed a memory forensics technique for applications working on .NET Framework and .Net Core. Based on the publicly available source code of .Net Core, they developed a Volatility plugin that extracts the portable executable (PE) image, loaded modules, classes, class member variables, and methods of a .NET application by analyzing the layout of key structures.

2.3. Automatic object layout generation

Qi et al. (2022) proposed LogicMEM, which automatically generates profiles for Linux memory forensics. A profile contains a layout of the structures used by the operating system to manage important data such as a process. Windows provides symbol files in which Microsoft stores structure layout information such as *EPROCESS*, but Linux does not provide this information. Therefore, a profile must be obtained from a live system. LogicMEM automatically generates a profile for kernel module structures such as *struct task_struct* by determining the validity of values on a field-by-field basis. The profile generated by LogicMEM is used by the Volatility framework.

In our work, we propose an advanced memory forensic technique using object layout analysis to overcome the limitations of existing web browser-related memory forensics.

3. Background

3.1. Virtual memory

Virtual memory is one of the memory management techniques, in which the operating system allocates each process with a unique memory space only accessible by that particular process; this unique memory space is called a virtual address space. When a process accesses its own virtual address space, the operating system translates the virtual address into a physical address to access the data stored in the physical memory. The virtual memory technique prevents multiple processes from interfering with each other's memory space and allows each process to use a memory space larger than the size of the actual physical memory.

Virtual memory is separated into user mode and kernel mode. The user-mode area is divided into the text area where executable code is stored, the stack area where function local variables are stored, and the

heap area where dynamically allocated space is located during runtime.

Since the heap area contains data generated by a process as it runs, there is considerable useful information for analyzing the application memory, such as cipher keys that encrypt sensitive data and passwords entered by a user. In addition, the objects of class or struct used by a process to internally manage data for processing are located in the heap area. Hence, if it can identify the objects in the process memory, it can provide context for the extracted data.

3.2. Object layout

In C++, a class type includes *class*, *struct*, and *union*. This paper writes them as a class. A class consists of fields storing a value and methods being responsible for an action. When the source code is built, the compiler generates object layouts that determine the arrangement of fields in a class. The mapping of a source-level class into machine-level memory is called an "object layout algorithm" (Ramananandro et al., 2011). An object means that a class is mapped (instanced) into the memory based on the object layout. In Windows, the methods are located in a text area, such as the *.text* section in the PE file format. Fields, however, are located in the stack or heap area in the order in which they are declared.

For example, Chromium's *NavigationEntryImpl* is a class that stores a visited URL, the title of a document on such URL, etc. **Fig. 1** shows a *NavigationEntryImpl* object in a 64-bit Chromium browser process. **Fig. 1** (A) shows the object layout of the *NavigationEntryImpl* class, and **Fig. 1** (B) displays the raw byte stream of the object while being dynamically created and loaded into memory. **Fig. 1** (a)–(f) denotes the fields of *title_*, *favicon_*, *ssl_*, *transition_type_*, *user_typed_url_* and *restore_type_* and their corresponding values. In **Fig. 1** (B), there are 4-byte alignment paddings behind *transition_type_* and *restore_type_*. The *NavigationEntryImpl* object shown in **Fig. 1** (B) contains information about the default startup page URL, which is *chrome://newtab/*, when creating a new tab.

In C++ standard, there is an alignment requirement that describes the arrangement of the fields (Objects and alignment). Alignment implies that 2- and 4-byte must be located at memory addresses that are multiples of two and four, respectively. If a one-byte field is followed by a four-byte field, three bytes of alignment padding are inserted between the two fields. In the case of class inheritance, the fields of the parent class are placed first, followed by those of the child class. If a virtual function exists within a class, the object of that class has a pointer to the virtual function table. Nonvirtual member functions do not affect the layout of the object. In this work, we generate object layouts for the Chromium project source code and discover objects allocated to the virtual memory to extract meaningful information from a forensic perspective.

For example, in Chromium, the "64-bit object layout" of string, vector, and map, which are the representative C++ Standard Template Library (STL) container classes, is as follows.

3.2.1. *std::string and std::u16string*

A string type is a 24-byte data structure to store null-terminated strings. *std::string* has a string consisting of single-byte characters. If the length of the string is less than 23 characters, the character array is stored at offset 0; otherwise, the pointer in the character array is stored. *std::u16string* has a string consisting of 16-bit characters. If the length of the string is less than 11 characters, it stores a 16-bit character array; otherwise, it stores a pointer in the array.

3.2.2. *std::vector*

A vector is an ordered data structure. It is internally implemented as an array of elements. The start and end addresses of the array are stored at offsets 0 and 8, respectively.

3.2.3. *std::map*

A map is a data structure of key–value pair consisting of keys that cannot be duplicated and values that can be obtained through such keys. The map is internally implemented as a red-black tree ordered by the key data. Each node has three-pointers for the left, right, and parent nodes: Boolean, key data, and value data.

4. Chromium's classes relating to web browsing activities

In this section, we inspect the Chromium source code to identify classes relevant to a user's web browsing activity. The Chromium browser has a *Browser* class, which is responsible for a single window, and various classes for managing tab groups, tabs, and lists of visited URLs. **Figure** 2 shows the classes and relationships between them that are of interest from a digital forensics perspective; (a)-(d) represent classes related to the browser, tab group, tab, and information about the URL visited by a user, respectively. The colored boxes indicate the key classes. The key fields of each class are listed in **Table 1**.

4.1. Browser

Browser class manages a single window and can refer to pointer variables to discover tab groups, tabs, and other key subobjects. The key fields contain information, such as the window name, a tab strip to manage tabs, and a profile to store information about the user and session. Note that the *profile_* member has a *ProfileImpl* object when

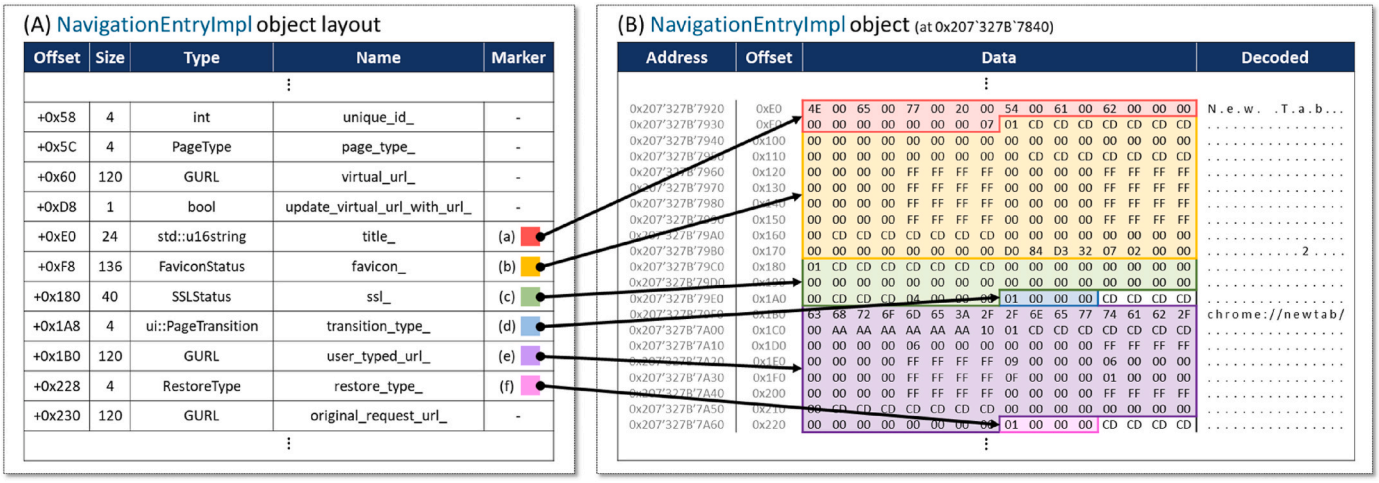


Fig. 1. 'Object layout' (A) and 'object' (B) of NavigationEntryImpl class.

Table 1
The list of key fields of each class.

Class Name	Field Name	Description
Browser	profile_	User profile (it can be used to distinguish whether that window is private or not)
	tab_strip_model_	Tabs and tab groups
	session_id_	ID of this *Browser* object
	bookmark_bar_state_	Bookmark bar state that represents whether it shows
	window_has_shown_	Boolean that represents whether
	user_title_	Window's name
ProfileImpl	path_	AppData path
TabStripModel	contents_data_	List of created tabs
	group_model_	Tab groups-related data
	selection_model_	Index of the last used tab
TabGroup	id_	ID of the tab group
	visual_data_	Visual data of the tab group
	tab_count_	Count of tabs in the tab group
TabGroupId	token_	16-byte array representing unique ID
TabGroupVisualData	title_	Name of the tab group
	color_	Theme color of the tab group
NavigationControllerImpl	entries_	List of visited URL
NavigationEntryImpl	frame_tree_	Data related to HTTP request
	unique_id_	ID of the entry
	title_	Document title of the visited web page
	favicon_	Favicon data (URL, etc.)
	ssl_	SSL data (Certificate, etc.)
	user_typed_url_	Visited URL (shown in address bar)
	timestamp_	Visited timestamp (Chrome timestamp)
	http_status_code_	HTTP response status code
FrameNavigationEntry	url_	Accessed URL (after redirect)
	referrer_	Referrer data
	redirect_chain_	List of redirected URLs
	method_	HTTP request method
FaviconStatus	url	Favicon URL
	image	Favicon image data
SSLStatus	certificates	SSL certificate data
X509Certificate	subject_	Owner of the certificate
	issuer_	Entity that issues the certificate
	valid_start_	Start of validity date
	valid_expiry_	Expiry of validity date
	serial_number_	Serial number of the certificate

normally used; however, it has an *OffTheRecordProfileImpl* object when using the private mode.

TabStripModel class manages tab groups and tabs. It stores a list of tabs that are currently active in a Chromium window using a single vector container and contains information about the tab group. It also stores the index number of the last used tab; therefore, it can be used to trace the user's last activity.

4.2. TabGroup

A tab group is a feature that combines multiple tabs into a single group. A *group_model_* member of a *TabGroupModel* type included in a *TabStripModel* class manages tab-group-related data using the *groups_* member of the *std::map* type with *TabGroupId* and *TabGroup* as the key and value, respectively. The *TabGroup* stores information regarding the current tab group, such as its name and color, in a *TabGroupVisualData*-type *visual_data_* member.

4.3. Tab

A tab is a component that renders the web page of the visited URL of the user. The *Tab* class manages whether a tab is pinned, the group to which the tab belongs, etc. It also stores a list of visited URLs, which can be obtained through the *contents_* member of the *WebContents* type. The *WebContents* class is implemented by the *WebContentsImpl* class. To obtain a list of visited URLs, they must be appropriately referred to subobjects through the class reference relationships shown in **Fig. 2**.

4.4. NavigationEntry

NavigationEntry class contains information about the individual URL the user visited. Each time the user visits a new URL in each tab, a new *NavigationEntry* object is created. They are stored and managed in the *entries_* member, which is a single vector container, in the *NavigationControllerImpl* class. In addition, the favicon and SSL certificate information are stored in this class. *NavigationEntry* class is implemented by the *NavigationEntryImpl* class.

The *frame_tree_* member of the *NavigationEntryImpl* class can refer to the *FrameNavigationEntry* class. The final URL of the entry can be obtained from the *FrameNavigationEntry* object. When a user visits a specific URL, the *user_typed_url_* member of the *NavigationEntryImpl* class has the first accessed URL, whereas the *url_* member of the *FrameNavigationEntry* class has the final URL after all the redirections. Hence, it contains a list of redirected URLs and referrer-related data.

5. Methodology

5.1. Overview

This section proposes an automated system for extracting the forensically meaningful information identified in Section 4 from the virtual memory areas of Chromium processes. First, the system builds the Chromium source codes (considering different commits and versions) to generate the corresponding symbol files. Then, to extract meaningful information from virtual memory, it obtains detailed object layout information by interpreting the symbol files and discovers relevant objects from a memory dump by validating the values that each field can store. The workflow of the proposed methodology is illustrated in Fig. 3.

5.2. Automatic object layout generation

Chromium can be built in Windows using Ninja (Checking out and Building Chromium for Windows). Ninja (Ninja) compiles the source code using a Visual Studio compiler, which normally creates a Program Database (PDB) symbol file used to store additional information for debugging (e.g., object layouts, function names, and so on). To obtain object layouts from the created PDB, we utilized Ghidra's PDB parsing feature (Ghidra). Since the object layouts of each class mentioned in the previous section are stored in PDB files related to *chrome.dll* and *content.dll*, we attempted to convert each PDB file into a user-friendly readable format, such as Extensible Markup Language (XML), to easily utilize that information in subsequent automation processes.

5.3. Extraction of forensically meaningful information

The classes can have member variables of various types. These include fundamental types (e.g., *int* and *double*), class types (e.g., *class* and *struct*), enumeration types (e.g., *enum*), and pointer types with virtual memory address values. "Pointer" type members typically point to objects of other fundamental or class types on the heap area. All the algorithms are presented in the Appendix.

5.3.1. Validation of a pointer value

The **Algorithm 1** can be utilized to determine the validity of values stored in the pointer type members. Specifically, the algorithm takes a specific virtual address value and checks whether it is within the range of valid addresses in user mode and included in the Virtual Address Descriptor (VAD) list of the target process.

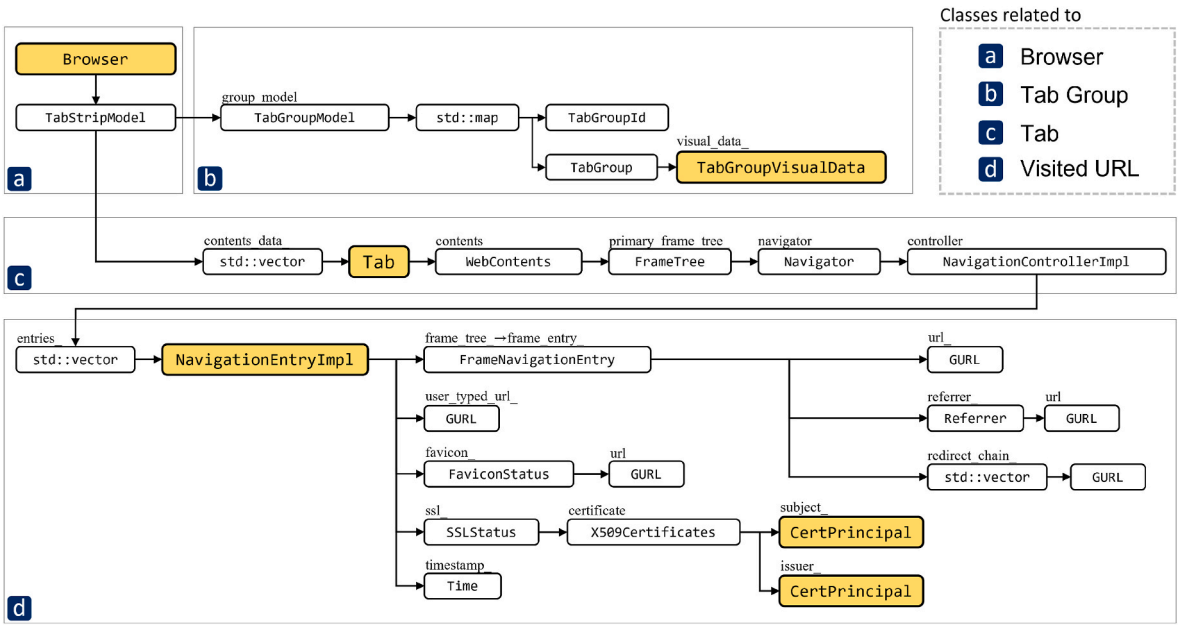


Fig. 2. Chromium's browsing-related classes and their relationships.

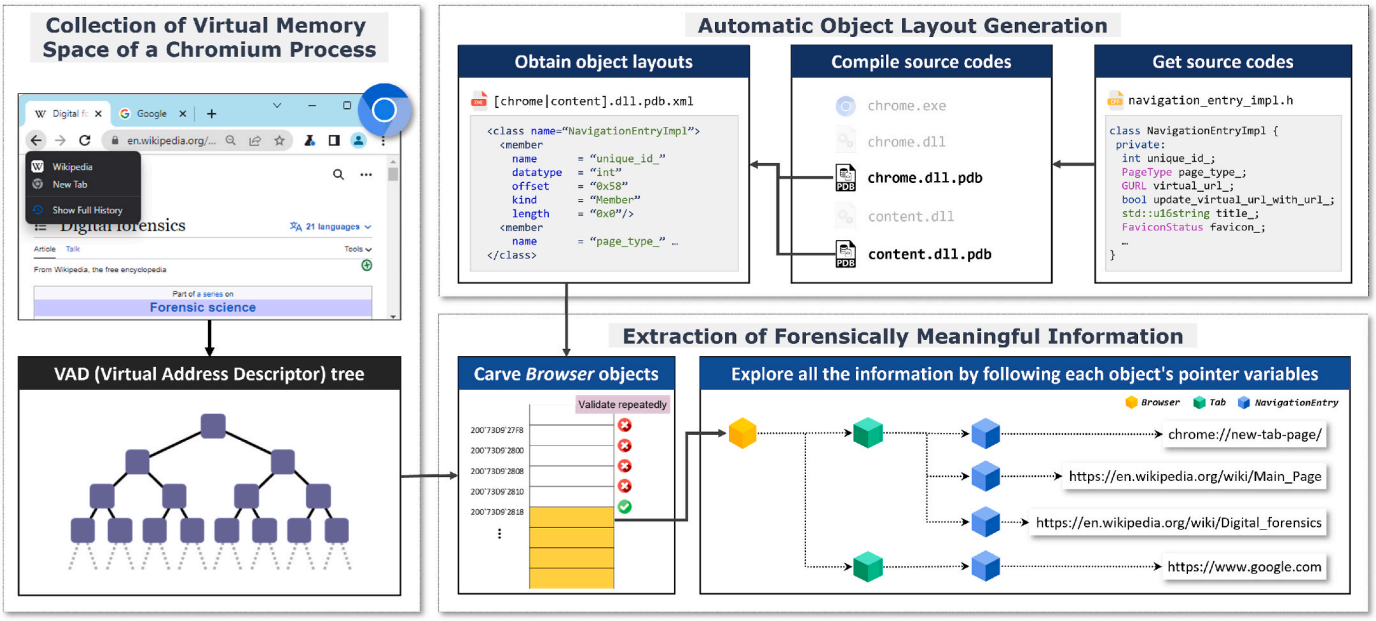


Fig. 3. Operational workflow for memory analysis of Chromium-based browsers.

To effectively discover the objects of a particular class in a memory dump, it is necessary to validate the values stored in several key fields of the class. For example, assume that a particular class has a field having a pointer value at offset "0x100". If an object of a particular class exists at address "0x200`3456`8760" (= base), a value stored at address "0x200`3456`8860" (= base + offset) should be a valid virtual address for referencing another object. The accuracy of detecting valid objects will increase with more member variables that should be checked for valid values.

5.3.2. *Validation of fields*

Algorithm 2 represents detailed steps for validating the values of an object loaded at a virtual address. Individual pointer fields can be validated using Algorithm 1, and a certain field can be validated by determining whether its value is *nullable*. The value of an enumeration type can be checked to determine whether it is included within the range corresponding to its enumeration definition, and a boolean field value must have only zero or one as its value. In addition, if a field contains a pointer value to reference an object of another class, Algorithm 2 is called recursively, as indicated in line 29 of the algorithm.

5.3.3. *Scanning Browser objects*

Algorithm 3 describes how to detect *Browser* objects in a virtual memory space of the target Chromium process, which can be obtained by traversing nodes stored in the process's VAD tree. The algorithm uses only memory areas with PAGE_READWRITE protection, excluding read-only pages, to increase the efficiency of the object-carving operation. In addition, since the *Browser* class has a *virtual* function, it has a 64-bit

pointer at offset 0 to reference a relevant virtual function table. Moreover, it must be located at a virtual memory address that is a multiple of eight owing to the alignment requirements of C++. Therefore, it performs a validation check on each candidate *Browser* object in 8-byte increments.

As mentioned above, when a candidate virtual memory address is detected where a *Browser* object exists, several key fields of the object can be accessed and validated by referring to its object layout. Furthermore, the relevant classes mentioned in the previous section are retrieved based on their referential relationships to extract forensically meaningful information.

6. Experiment and result analysis

6.1. Experimental setup

The operating system used for the experiment was Windows 11 22H2 (OS Build 22621.1265), and the Chromium version was 113.0.5650.0 (Chromium source code commit fed2d65). The compiler used for the build was Visual C++ Compiler v19.33.31630 in Visual Studio 2022 v17.3.6.

The goal of the experiment is to carve the *Browser* object from the virtual memory of the target Chromium based on the object layout analyzed in Section 4 and further obtain information about the user's activity, such as the list of created tabs, visited URLs, and SSL certificates.

6.2. Implementation and dataset

Chracer, which we developed as a proof-of-concept tool, takes two inputs: a user-space memory dump (in this case a Windows minidump), and parsed symbol file. The minidump file was extracted from the Chromium browser process using Process Hacker v2.39.124 (Process Hacker), and the symbol file was parsed by Ghidra by converting the PDB of *chrome.dll* and *content.dll* into XML files. *Chracer* uses these XML files to obtain object layouts of user activity-related classes. The source code and dataset are publicly available on Github (Chracer). The detailed user's browsing activities performed for each experiment were documented in *Chracer*'s GitHub repository.

6.3. Results

6.3.1. Carving browser objects

To discover the Browser object that is responsible for a single Chromium window in the virtual memory of the Chromium process, the methodology proposed in the previous section was applied. In this experiment, we created four Chromium windows with a default tab for each window, visiting particular websites in the following order: Google, GitHub, YouTube, and Chromium. Fig. 4 shows that *Chracer* discovers four *Browser* objects in the virtual memory dump and extracts the session ID, tab number, document title on a page, and visited URL. Each Chromium window can be distinguished by its session ID, resulting in four *Browser* objects with different session IDs residing in the memory.

6.3.2. Extracting tab groups

A *std::map* is a data structure comprising a pair of keys and values and has a binary tree structure internally. In this experiment, *Chracer* extracts tab-group information from the virtual memory of the Chromium process by interpreting the *std::map* structure, with *TabGroupId* and *TabGroup* as the key and value, respectively. We added eight tabs to one Chromium window and set each of the two tabs into one group to create a total of four tab groups. In addition, we set the name of each tab group to "TabGroup{*Number*}" manually. Fig. 5 shows that *Chracer* identifies the *TabGroup* and *TabGroupVisualData* objects and outputs the tab group name, tab group color, and the index of the tabs included in that tab group.

SessionID	Tab	Title	URL
450578886	0	New Tab	chrome://newtab/
450578886	0	Google	https://www.google.com/
450579008	0	New Tab	chrome://newtab/
450579008	0	GitHub: Let's build from here · GitHub	http://github.com/
450579010	0	New Tab	chrome://newtab/
450579010	0	(4) YouTube	https://www.youtube.com/
450579012	0	New Tab	chrome://newtab/
450579012	0	Home	https://www.chromium.org/chromium-projects

Fig. 4. Output of *Chracer*: Browser objects.

6.3.3. Extracting tabs

Chromium uses *std::vector* to manage multiple tabs and tab-related information (e.g., visited URLs). An *std::vector* is an ordered data structure internally containing an array. In this experiment, *Chracer* interprets the vector structure to extract the tabs included in a Chromium window and visited URLs included in each tab. We created two tabs in a single Chromium window and visited five URLs in each tab. Table 2 shows the list of visited URLs.

As shown in Fig. 6, *Chracer* identifies two tabs and five visited URLs in each tab.

6.3.4. Extracting SSL certificates

The SSL certificate information of the web server visited by the user can be obtained from memory. Fig. 7 shows obtained certificate information (**.wikipedia.org* and **.chromium.org*) from a memory dump of the previous experiment (Section 6.3.3). A web server certificate is an important component that ensures server authenticity. However, in a system infected by a malicious attacker, a malicious certificate can be inserted into the list of trusted certificates. In this case, a certificate error may not occur when accessing a phishing site created by the attacker. Therefore, it is necessary to extract the certificate information loaded in the memory and utilize it when analyzing an infected system.

6.3.5. Identifying private mode

Chromium provides an incognito (private) mode. Prior research (Hariharan et al., 2022; Said et al., 2011) has shown that using the private mode leaves no data on the file system. In this experiment, *Chracer* identifies all Chromium windows from the memory and determines whether each window is in private mode. It also extracts the URLs visited in the private mode. By determining the class type of an object stored in the *profile_* of the *Browser* object using **Algorithm 2**, we can determine whether the window is in private mode. If a Chromium window is in private mode, the object of the *OffTheRecordProfile* class is stored in the *profile_* field. Otherwise, the *profile_* normally contains an object of the *ProfileImpl* class. Therefore, if the result of "validate (**profile_*, *OffTheRecordProfile*, *OL*, *VAD*)" is true, its Chromium window is in private mode.

In this experiment, we created four windows: two private windows and two normal windows. In addition, we visited one URL in each window, accessing Chromium and Wikipedia pages in private windows and Google and YouTube pages in normal windows.

As shown in Fig. 8, *Chracer* identifies all Chromium windows and effectively determines whether they are in private mode. Furthermore, the visited URLs can be extracted from a private mode window. This is useful for investigating user activities that attempt to hide suspicious

SessionID	TabGroup	TabGroupColor	Tab	URL
450577899	TabGroup1	kGrey	0	chrome://new-tab-page/
450577899	TabGroup1	kGrey	1	chrome://new-tab-page/
450577899	TabGroup2	kBlue	2	chrome://new-tab-page/
450577899	TabGroup2	kBlue	3	chrome://new-tab-page/
450577899	TabGroup3	kRed	4	chrome://new-tab-page/
450577899	TabGroup3	kRed	5	chrome://new-tab-page/
450577899	TabGroup4	kYellow	6	chrome://new-tab-page/
450577899	TabGroup4	kYellow	7	chrome://new-tab-page/

Fig. 5. Output of *Chracer*: Tab groups.

Table 2
The list of visited URLs on each tab.

Tab	Order	Web Pages
First Tab	1	https://www.wikipedia.or
	2	https://en.wikipedia.org/wiki/Digital_forensics
	3	https://en.wikipedia.org/wiki/Computer_forensics
	4	https://en.wikipedia.org/wiki/Digital_evidence
	5	https://en.wikipedia.org/wiki/Best_evidence_rule
Second Tab	1	https://www.chromium.org/chromium-projects/
	2	https://www.chromium.org/Home/
	3	https://www.chromium.org/developers/
	4	https://www.chromium.org/developers/getting-around-the-chrome-source-code/
	5	https://www.chromium.org/developers/design-documents/multi-process-architecture/

intentions.

7. Discussion

7.1. Chromium-based browsers

We applied the methodology proposed in Section 5 to Chromium-based Google Chrome (v111.0.5563.65), Microsoft Edge (v111.0.1661.44), and Brave (v1.52.129). We found that some object layouts, including the *Browser* class, were different from those in the official Chromium source code. For example, the *title_* field of the *NavigationEntryImp1* class is located at 0xE0 offset in Chromium's source code but is located at 0xC8 offset in Google Chrome's virtual memory. However, we needed to adjust the offsets of some fields by manually comparing a raw byte stream of key objects, such as *Browser*, to interpret and correlate all relevant objects.

We dumped minidump files after visiting some websites. Table 3 shows the list of visited URLs using Google Chrome, Microsoft Edge, and Brave.

Fig. 9 shows the results of extracting user activity-related information of Google Chrome, Microsoft Edge, and Brave, respectively.

7.2. Non-volatile session restore information

Chromium browser provides a session restore feature, called "Restore Window", to restore the last used session when the Chromium process is abnormally terminated or a user wants to reuse the last session. A session contains information about a user's web browsing activities, including windows, tabs, and a list of visited URLs, that has accumulated since the browser was launched. Chromium stores the last session information in the *Last Session* and *Last Tabs* files. There are publicly available tools to interpret these artifacts (CCL Solutions Group; lemnos); however, they can only parse information stored in nonvolatile files. Since SSL certificate information in normal/private mode and the list of visited URLs in private mode are not stored in the file system, utilizing the proposed methodology can provide additional useful information from a digital forensics perspective than using only nonvolatile data.

7.3. Free pages and browsing-related objects

When a process terminates or frees unused pages, the operating system does not immediately initialize (overwrite NULL to) the freed page. The freed pages are set to the free state until they are allocated to another process, and the data used by the process remain on the page while they are in the free state. This suggests the possibility of finding forensically meaningful information on freed pages in the physical memory. If analysts know the object layout of the class that they want to discover, the object-carving algorithm (**Algorithm 2**) used in this methodology can be utilized to find forensically meaningful information in the freed pages.

7.4. Limitations of the proposed methodology

The proposed methodology first carves the *Browser* object and then

SessionID	Incognito	Tab	Title	URL
450578311	False	0	New Tab	chrome://newtab/
450578311	False	0	Google	https://google.com/
450578448	False	0	New Tab	chrome://newtab/
450578448	False	0	YouTube	https://www.youtube.com/
450578450	True	0	New Incognito Tab	chrome://newtab/
450578450	True	0	Home	https://www.chromium.org/chromium-projects
450578452	True	0	New Incognito Tab	chrome://newtab/
450578452	True	0	Wikipedia	https://www.wikipedia.org/

Fig. 8. Output of *Chracer*: URLs visited in *private* mode.

Table 3
The list of visited URLs on each browser.

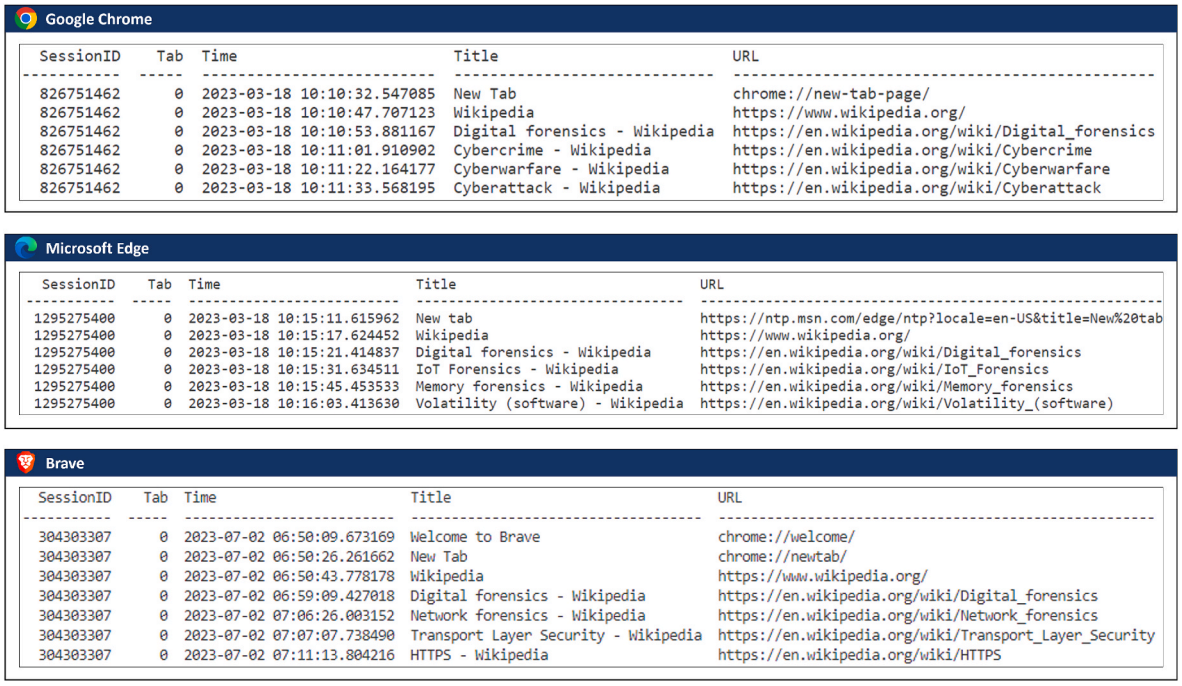
Browser	Order	Web Pages
Google Chrome	1	https://www.wikipedia.org
	2	https://en.wikipedia.org/wiki/Digital_forensics
	3	https://en.wikipedia.org/wiki/Cybercrime
	4	https://en.wikipedia.org/wiki/Cyberwarfare
	5	https://en.wikipedia.org/wiki/Cyberattack
Microsoft Edge	1	https://www.wikipedia.org
	2	https://en.wikipedia.org/wiki/Digital_forensics
	3	https://en.wikipedia.org/wiki/IoT_Forensics
	4	https://en.wikipedia.org/wiki/Memory_forensics
	5	https://en.wikipedia.org/wiki/Volatility_(software)
Brave	1	https://www.wikipedia.org
	2	https://en.wikipedia.org/wiki/Digital_forensics
	3	https://en.wikipedia.org/wiki/Network_forensics
	4	https://en.wikipedia.org/wiki/Transport_Layer_Security
	5	https://en.wikipedia.org/wiki/HTTPS

SessionID	Tab	Time	URL	Referrer
450578107	0	2023-03-14 07:45:04.539029	chrome://new-tab-page/	
450578107	0	2023-03-14 07:45:53.380520	https://www.wikipedia.org/	
450578107	0	2023-03-14 07:46:02.324514	https://en.wikipedia.org/wiki/Digital_forensics	https://www.wikipedia.org/
450578107	0	2023-03-14 07:46:44.296012	https://en.wikipedia.org/wiki/Computer_forensics	https://en.wikipedia.org/wiki/Digital_forensics
450578107	0	2023-03-14 07:46:57.161281	https://en.wikipedia.org/wiki/Digital_evidence	https://en.wikipedia.org/wiki/Computer_forensics
450578107	0	2023-03-14 07:47:11.672663	https://en.wikipedia.org/wiki/Best_evidence_rule	https://en.wikipedia.org/wiki/Digital_evidence
450578107	1	2023-03-14 07:45:28.334586	chrome://new-tab-page/	
450578107	1	2023-03-14 07:56:37.220422	https://www.chromium.org/chromium-projects/	https://www.chromium.org/
450578107	1	2023-03-14 07:56:41.482306	https://www.chromium.org/Home/	https://www.chromium.org/chromium-projects/
450578107	1	2023-03-14 07:59:36.147159	https://www.chromium.org/developers/	https://www.chromium.org/Home/
450578107	1	2023-03-14 07:59:37.065162	https://www.chromium.org/developers/how-tos/getting-around-the-chrome-source-code/	https://www.chromium.org/developers/
450578107	1	2023-03-14 08:00:04.460213	https://www.chromium.org/developers/design-documents/multi-process-architecture/	https://www.chromium.org/developers/how-tos/getting-around-the-chrome-source-code/

Fig. 6. Output of *Chracer*: Tabs and visited URLs on each tab.

SerialNumber	CommonName	Issuer	ValidStart	ValidExpiry
04:0F:88:67:7F:F0:B0:C3:D9:42:86:02:62:16:EC:E9	*.wikipedia.org	DigiCert TLS Hybrid ECC SHA384 2020 CA1	2022-10-27 00:00:00	2023-11-17 23:59:59
6F:8A:31:8E:CB:DC:E4:E8:12:E7:1E:0F:9F:81:6C:7D	www.chromium.org	GTS CA 1D4	2023-01-26 17:02:05	2023-04-26 18:01:27

Fig. 7. Output of *Chracer*: SSL certificates relating to visited websites.



Google Chrome

SessionID	Tab	Time	Title	URL
826751462	0	2023-03-18 10:10:32.547085	New Tab	chrome://new-tab-page/
826751462	0	2023-03-18 10:10:47.707123	Wikipedia	https://www.wikipedia.org/
826751462	0	2023-03-18 10:10:53.881167	Digital forensics - Wikipedia	https://en.wikipedia.org/wiki/Digital_forensics
826751462	0	2023-03-18 10:11:01.910902	Cybercrime - Wikipedia	https://en.wikipedia.org/wiki/Cybercrime
826751462	0	2023-03-18 10:11:22.164177	Cyberwarfare - Wikipedia	https://en.wikipedia.org/wiki/Cyberwarfare
826751462	0	2023-03-18 10:11:33.568195	Cyberattack - Wikipedia	https://en.wikipedia.org/wiki/Cyberattack

Microsoft Edge

SessionID	Tab	Time	Title	URL
1295275400	0	2023-03-18 10:15:11.615962	New tab	https://ntp.msn.com/edge/ntp?locale=en-US&title=New%20tab
1295275400	0	2023-03-18 10:15:17.624452	Wikipedia	https://www.wikipedia.org/
1295275400	0	2023-03-18 10:15:21.414837	Digital forensics - Wikipedia	https://en.wikipedia.org/wiki/Digital_forensics
1295275400	0	2023-03-18 10:15:31.634511	IoT Forensics - Wikipedia	https://en.wikipedia.org/wiki/IoT_Forensics
1295275400	0	2023-03-18 10:15:45.453533	Memory forensics - Wikipedia	https://en.wikipedia.org/wiki/Memory_forensics
1295275400	0	2023-03-18 10:16:03.413630	Volatility (software) - Wikipedia	https://en.wikipedia.org/wiki/Volatility_(software)

Brave

SessionID	Tab	Time	Title	URL
304303307	0	2023-07-02 06:50:09.673169	Welcome to Brave	chrome://welcome/
304303307	0	2023-07-02 06:50:26.261662	New Tab	chrome://newtab/
304303307	0	2023-07-02 06:50:43.778178	Wikipedia	https://www.wikipedia.org/
304303307	0	2023-07-02 06:59:09.427018	Digital forensics - Wikipedia	https://en.wikipedia.org/wiki/Digital_forensics
304303307	0	2023-07-02 07:06:26.003152	Network forensics - Wikipedia	https://en.wikipedia.org/wiki/Network_forensics
304303307	0	2023-07-02 07:07:07.738490	Transport Layer Security - Wikipedia	https://en.wikipedia.org/wiki/Transport_Layer_Security
304303307	0	2023-07-02 07:11:13.804216	HTTPS - Wikipedia	https://en.wikipedia.org/wiki/HTTPS

Fig. 9. Output of *Chracer*: URLs visited in Google Chrome, Microsoft Edge, and Brave.

extracts forensically meaningful information by referencing its fields and pointers. Thus, if it fails to find the *Browser* object, it is difficult to proceed with the step that refers to fields or pointers to extract useful information, such as tab groups and the list of visited URLs.

Since the size of the symbol files (*chrome.dll* and *content.dll*) is significantly large, *Chracer* takes a long execution time; thus, it must be optimized. Our future work will focus on enhancing *Chracer* to improve this aspect.

As mentioned in Section 7.1, although Google Chrome, Microsoft Edge and Brave are based on Chromium, the offsets of certain fields may differ from those of the official Chromium source code. Automatic adjustments of these offsets were not performed.

8. Conclusion and future directions

Existing memory forensics for user applications has been mostly performed with string searching and simple pattern matching. However, since there is no or insufficient context to interpret a piece of data detected by those traditional approaches, additional analysis efforts are required to reveal user activity-related context. This work dissected Chromium's source code to identify classes relating to a user's web browsing activities such as creating a browser window, adding tabs, and visiting specific URLs. Based on object layouts automatically generated from the source codes, it is possible to detect and correlate relevant objects (e.g., *Browser*, *Tab*, *NavigationEntryImpl*, etc.) from virtual memory of Chromium processes, in order to extract forensically meaningful browsing-related traces. Also, we were able to extract information, such as SSL certificates, that previous works had not extracted. Furthermore, this paper proposed a systematic methodology to automate the entire process for Chromium browser memory forensics.

Further, this paper demonstrated that the proposed methodology can correctly extract forensically meaningful information, such as a list of visited URLs, even when a Chromium browser is running in private mode. Although some classes' fields and types needed to be manually adjusted due to differences with the official Chromium, our experiments showed that the proposal can be applied to any Chromium-based browsers such as Google Chrome, Microsoft Edge and Brave. Our findings overcome the limitations of traditional string search and pattern matching methods, and moreover can be used to identify users' web browsing activities in detail.

This work only focuses on Windows environments along with object layouts obtained by building the Chromium source code on Windows. In future work, we will propose a generalized methodology that can be applied to other operating systems such as Linux, macOS, and Android. In addition, we plan to conduct research on popular user applications other than web browsers, in order to effectively detect memory objects and analyze their correlations for assisting digital investigation.

The current version of *Chracer* processes a minidump of target process. Therefore, We are working on developing a Volatility3 plugin based on our proposal for processing a physical memory dump. Unfortunately, we haven't found an effective way to reference higher-level classes from lower-level classes. We will find the method that backtracks traversal of pointer from child object to parent object.

Appendix

Algorithm 1. Validating a virtual memory address

Algorithm 1 Validating a virtual memory address

```
1:  VAD ← VAD tree of the target process
2:  p ← Virtual memory address to be validated
3:
4:  procedure validate_pointer(p, VAD)
5:      if p > 0x7FFF `FFFF `FFFF then
6:          return False
7:      end if
8:      if not p in VAD then
9:          return False
10:     end if
11:     return True
12: end procedure
```

Algorithm 2. Validating fields of a candidate object

Algorithm 2 Validating fields of a candidate object

```
VAD ← VAD tree of the target process
OL ← List of supported object layouts
base ← Virtual memory address to be validated
t ← Target class to be validated

procedure validate(base, t, OL, VAD)
    fields ← get_object_layout(t, OL)
    for all field ∈ fields do
        va ← base + field.offset
        if type(field) == pointer then
            p ← convert_to_pointer(va, length=64)
            if not validate_pointer(p, VAD) then
                return False
            end if
            if not validate(p, type(*p)) then
                return False
            end if
        else if type(field) == enum then
            e ← convert_to_integer(va, length=32)
            if not e ∈ enum(field) then
                return False
            end if
        else if type(field) == bool then
            b ← convert_to_integer(va, length=8)
            if not b ∈ {0, 1} then
                return False
            end if
        else if type(field) is class then
            if not validate(va, type(field), OL, VAD) then
                return False
            end if
        end if
    end for
    return True
end procedure
```

Algorithm 3. Scanning Browser object(s)

Algorithm 3 Scanning `Browser` object(s)

```
VAD ← VAD tree of the target process
OL ← List of supported object layouts

procedure carve_browser_objects(OL, VAD)
    browser_objects ← []
    for all vad ∈ VAD do
        if vad.protection == PAGE_READWRITE then
            base ← vad.start
            while base+sizeof(Browser) < vad.end do
                if validate(base, Browser, OL, VAD) then
                    browser_objects.append(base)
                end if
                base ← base+8
            end while
        end if
    end for
    return browser_objects
end procedure
```


Acknowledgements

This work was supported by Institute for Information & communications Technology Promotion(IITP) grant funded by the Korea government(MSIT) (No.2022-0-00281, Development of digital evidence analysis technique using artificial intelligence technology).

References

Alfosail, M., Norris, P., 2021. Tor forensics: Proposed workflow for client memory artefacts. Comput. Secur. 106, 102311.

Chromium Docs. Checking out and Building Chromium for Windows. Available: https://chromium.googlesource.com/chromium/src/+/main/docs/windows_build_instructions.md. (Accessed 16 May 2023).

cppreference. Objects and alignment. Available: https://en.cppreference.com/w/c/language/object. (Accessed 16 May 2023).

CCL Solutions Group. ccl_chrome_indexeddb. Available: https://github.com/cclgroupltd/ccl_chrome_indexeddb. (Accessed 16 May 2023).

cube0x8. ChromeRagamuffin. https://github.com/cube0x8/ChromeRagamuffin. (Accessed 16 May 2023).

Dija, S., Ajana, J., Indu, V., Sabarinath, M., 2021. Web Browser Forensics for Retrieving Searched Keywords on the Internet. In: 2021 3rd International Conference on Advances in Computing, Communication Control and Networking. ICAC3N, pp. 1664–1668.

Fernández-Álvarez, P., Rodríguez, R.J., 2022. Extraction and analysis of retrievable memory artifacts from Windows Telegram Desktop application. Forensic Sci. Int.: Digit. Invest. 40, 301342 (selected Papers of the Ninth Annual DFRWS Europe Conference).

Volatility Foundation. Volatility Framework. https://www.volatilityfoundation.org. (Accessed 16 May 2023).

Volatility Foundation. Volatility plugin - yarascan. Available: https://github.com/volatilityfoundation/volatility3/blob/develop/volatility3/framework/plugins/yarascan.py. (Accessed 2 July 2023).

Garcia, G.L., 2007. Forensic physical memory analysis: an overview of tools and techniques. In: TKK T-110.5290 Seminar on Network Security, pp. 305–320.

Chracer. Available: https://github.com/geun-yeong/Chracer. (Accessed 17 July 2023).

Hariharan, M., Thakar, A., Sharma, P., 2022. Forensic Analysis of Private Mode Browsing Artifacts in Portable Web Browsers Using Memory Forensics. In: 2022 International Conference on Computing, Communication, Security and Intelligent Systems. IC3SIS, pp. 1–5.

Iqbal, F., Khalid, Z., Marrington, A., Shah, B., Hung, P.C., 2022. Forensic investigation of Google Meet for memory and browser artifacts. Forensic Sci. Int.: Digit. Invest. 43, 301448.

NSA. Ghidra. Available: https://ghidra-sre.org. (Accessed 16 May 2023).

Ninja. Available: https://ninja-build.org. (Accessed 16 May 2023).

lemnos. chrome-session-dump. Available: https://github.com/lemnos/chrome-session-dump. (Accessed 16 May 2023).

Process hacker. Available: https://processhacker.sourceforge.io. (Accessed 16 May 2023).

Manna, M., Case, A., Ali-Gombe, A., Richard, G.G., 2022. Memory analysis of .NET and .Net Core applications. Forensic Sci. Int.: Digit. Invest. 42, 301404 (proceedings of the Twenty-Second Annual DFRWS USA).

Qi, Z., Qu, Y., Yin, H., 2022. LogicMEM: Automatic Profile Generation for Binary-Only Memory Forensics via Logic Inference. In: Proceedings of 2022 Network and Distributed System Security Symposium.

Ramananandro, T., Dos Reis, G., Leroy, X., 2011. Formal verification of object layout for c++ multiple inheritance. Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL'11, pp. 67–80.

Said, H., Al Mutawa, N., Al Awadhi, I., Guimaraes, M., 2011. Forensic analysis of private browsing artifacts. In: 2011 International Conference on Innovations in Information Technology, pp. 197–202.

superponible, 2023. Volatility Plugins. Available: https://github.com/superponible/volatility-plugins. (Accessed 16 May 2023).