Hindawi
Security and Communication Networks
Volume 2017, Article ID 4621587, 15 pages
https://doi.org/10.1155/2017/4621587

Research Article

PMCAP: A Threat Model of Process Memory Data on the Windows Operating System

Jiaye Pan and Yi Zhuang

College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing 210016, China

Correspondence should be addressed to Yi Zhuang; zy16@nuaa.edu.cn

Received 7 March 2017; Accepted 19 April 2017; Published 22 May 2017

Academic Editor: Xiaojiang Du

Research on endpoint security involves both traditional PC platform and prevalent mobile platform, among which the analysis of software vulnerability and malware is one of the important contents. For researchers, it is necessary to carry out nonstop exploration of the insecure factors in order to better protect the endpoints. Driven by this motivation, we propose a new threat model named Process Memory Captor (PMCAP) on the Windows operating system which threatens the live process volatile memory data. Compared with other threats, PMCAP aims at dynamic data in the process memory and uses a noninvasive approach for data extraction. In this paper we describe and analyze the model and then give a detailed implementation taking four popular web browsers IE, Edge, Chrome, and Firefox as examples. Finally, the model is verified through real experiments and case studies. Compared with existing technologies, PMCAP can extract valuable data at a lower cost; some techniques in the model are also suitable for memory forensics and malware analysis.

1. Introduction

Although mobile platforms represented by Android and iOS have gained high market share and attracted many developers and researchers, PC terminals are still irreplaceable in people's work and life and Windows operation system (OS) plays an important role for its openness in activities such as writing and coding but is also accompanied by diversified malware nowadays, including ransomware, botnet, and banking Trojan [1, 2]. In the past few decades there are a lot of security research work on the Windows platform in order to enhance its security and protect sensitive data, including malware analysis and detection [3–5] and the state-of-the-art vulnerability exploits and mitigations [6–11].

From the perspective of Windows OS development, Data Execution Protection (DEP) and Address Space Layout Randomization (ASLR) are introduced on Windows XP SP2 to prevent the buffer overflow exploits. Then Windows 7 restricts program permissions by mechanisms such as User Account Control (UAC), protected mode, session 0 isolation, and BitLocker. Further Device Guard and AppContainer were integrated with Windows 10 which provide more fine-grained protection. Practice shows that the difficulty of compromising Windows OS is rising. Meanwhile Microsoft has launched the bounty program, which could motivate the researchers to discover new bypass methods [12]. This measure can help prevent real attacks from happening.

A large amount of runtime code and data information is stored in the volatile memory. Memory corruption bugs can lead to code execution vulnerability [9]. Private sensitive data of some programs are also stored in the volatile memory such as session information, and crypto key data. The HeartBleed vulnerability can cause the leakage of private key in the OpenSSL process memory [13]. The famous tool Mimikatz can extract plain user login passwords from the LSASS process on the running system [14]. Malware such as Qadars [1] and Lurk [2] can steal the banking information from the memory. In the field of memory forensics, much research has been done on memory data analysis and thus produced many practical results [15–18], such as the open source Volatility Framework [19]. On the other hand, some researches aim at process memory data protection [20, 21] but failed to carry

out large-scale deployment. Compared with memory data forensics or memory dump analysis, our research focuses on extracting and analyzing real time memory data which are more volatile and change more continuously. Therefore, apart from the reconstruction of data structure, time factor should also be considered.

Currently, Windows has been equipped with many security components and policies, but the applications running on the platform still face security challenges in practical use.

(a) Windows is an open ecosystem, which allows users to install and execute third-party program. It does not validate the program origin strictly and prevent the unknowns from being installed, despite giving some security warnings like UAC. Most security policies are not deployed by default and should always be configured by professionals. Attackers are also working with *PowerShell* to create their own threats even under the whitelist protection [22, 23]. All of these show that the possibility of malware intrusion still exists.

(b) Compared with the application sandbox on mobile platform, the access control mechanism on Windows is coarse-grained [20]. Some resources are not protected effectively; programs with basic permissions can generate potential threat capability. For example, processes at the same integrity level can access with each other; applications can read most directories and files on the disk. A privileged process can modify the system security configuration, add a trusted root certificate, or change the registry settings, and so on.

In view of the security situation described above, this paper proposes a new threat model PMCAP on the Windows platform. Our model targets the volatile process memory data in real time, especially network related data. Based on the attacks on the endpoint system, similar to a minimum fileless attack [24], the model first takes advantage of the imperfections of access control and the data protection mechanism, and then it can extract private data from the process memory. In general, our paper makes the following contributions.

First, it proposes a threat model of process memory data in real time on the Windows operating system. The model threatens web browsers and other programs built on diversified attack techniques. It also considers the influence of network factors on networking related memory data.

Second, it proposes a new memory data extraction method based on the thread stack space, and it can extract critical data effectively from the large address space.

Third, it analyzes the main data structures related to communications of popular web browsers, takes that as an example, and describes the implementation of the model.

Finally, it evaluates the implementation of the example and discusses its benefits and limitations.

The remainder of the paper proceeds as follows. In Section 2, we compare our work to some related works. In Section 3, a model overview is given. In Section 4, we present the implementation of the model in detail. In Section 5, experiments and analysis are given. Finally, a conclusion is given in Section 6.

2. Related Work

Many researchers focus on vulnerability analysis of application security; also, the detection and analysis of malware receive continuous attention. Some related research works are shown as follows.

2.1. Vulnerability Attack Techniques. Many state-of-the-art vulnerability exploit techniques such as code reuse attacks are proposed to bypass the latest mitigations; Schuster et al. proposed a novel attack technique denoted as counterfeit Object-Oriented Programming (COOP) [10], which constructs the malicious execution by invoking only chains of existing C++ virtual functions in the program and can bypass most Control Flow Integrity (CFI) solutions. A technique called data-oriented programming (DOP) was presented by Hu et al. [11], which does not depend on controlling data to hijack the control flow and only uses noncontrol data to build the Turing-complete attacks. Jia et al. analyzed the isolation of Chrome and proposed a method to bypass the same origin policy from a render process attack; then the attacker can access local systems with the help of cloud services [26].

Making use of an exploit to execute attack Payload on the target system is the first step of our model; the existing available attack techniques can be applied. Windows is equipped with many security mechanisms but there are still some bypass approaches, including some strong practical techniques [27]. Although these attack techniques have dependencies when actualized, but it shows that there always is potential threat of unknown attacks.

2.2. Memory Forensics. Researches on memory forensics mainly contain memory acquisition, memory analysis, and data structure recovery. More specifically it first uses hardware card, virtualization, and applications at different levels including user level and kernel level. To acquire memory [15], Hargreaves and Chivers proposed a method for recovering the decryption keys from the memory using linear scan [28]. Sylve et al. proposed a novel technique for locating kernel object allocations with quick pool tag scanning [17], which has a good performance in the large memory space. Taubmann et al. presented TLSkex which can extract the master key of a TLS connection at runtime from the virtual machine's memory. It works in a nonintrusive way and uses a brute force approach to find TLS master secrets by decrypting and verifying TLS records in a loop. For memory data analysis, Fu et al. presented an automatic memory analysis methodology based on data correlation through analyzing all kinds of OS data structures [16]. MemPick proposed by Haller et al. can detect and classify high-level data structures used in stripped binaries [29]. DSCRETE is an excellent automatic reverse engineering technique for recovering a variety of application data which reuses application logic from a subject binary program [30]. Neasbittt et al. proposed a lightweight forensic engine for web browsers called WebCapsule which can record and replay web sessions [31]. It is implemented with instrumentation code.

Compared with these research works, our model aims at obtaining live volatile memory data and tries to acquire and

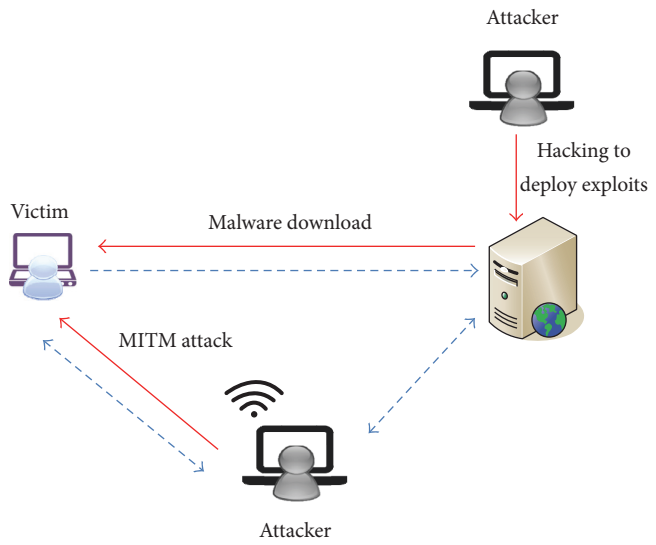

Figure 1: Common attack scenario.

analyze data continuously in real time. It is a challenge to solve the problem of locating and extracting the ephemeral data. The attack Payload of the model will run in the user mode on the target system. It has some limitations in terms of nonintrusive matters, such as thread schedule. Some techniques mentioned above, such as the automated data structure recognition, can provide assistance to the model. On the other hand, we think that some of the techniques referred in the model are helpful for memory extraction based on virtualization or hardware.

2.3. Process Data Protection. There are several research works for process data protection including encryption and isolation. Xie et al. proposed a binary code obfuscation method against stack trace analysis [32] which encodes the return address on the stack. Luțaș et al. proposed a protection tool (U-HIPE) [21], which prevents the hook injection and injected code execution in the process. The solution is based on hardware virtualization. Sze and Sekar presented a new approach SPIF [20] which ensures that the permissions for any process are influenced by code and data from untrusted sources are restricted. Chen et al. proposed Shreds [33], which implements a set of OS-backed programming primitives and helps developers protect sensitive memory in the process. Shreds has also been demonstrated on Linux. The most practical technique is the Intel Software Guard Extensions (SGX) [34], which has been supported by Windows 10 [35]. It provides a sandbox mechanism which can isolate the malwares on the platform including rootkits. It can stop the attacks implemented by Mimikatz [34] but is only supported by the Skylake architecture. Although these protections can prevent memory data from being accessed by malware, they sometimes depend on hardware and are difficult to be applied to common users of the Internet for large-scale protection in the near future.

Security vulnerability plays a key role on the penetration. Most existing research works for attack technology focus on code execution, for it is the first step of the entire attack process. Research works on memory forensics mostly solve the problem of memory snapshot analysis. Protections for memory data are not widely deployed on the endpoint of common users. On the other hand, the real malware always threatens data security through invasive behaviors such as hooking [1, 2]. The model in this paper is built on vulnerability attacks and emphasizes on live memory data extraction, including the effect of network factors. Our research proves it to be practical.

3. Model Overview

In most cases, the user encounters a malware attack because of the software installation bound with malicious code or a visit of deceptive URL link. The URL may be a normal website address, but it contains malicious code which has been deployed earlier by the attacker. On the other hand, the URL link may redirect to the attacker-controlled site when the communications are hijacked. Most commonly, the user's computer connects to the Wi-Fi hotspot faked by the attacker and gets attack by the Man in The Middle (MITM) technology. Even though the TLS protocols have been widely used, there still is a lot of web traffic being exposed to the risk. Figure 1 shows the typical attack scenario well known by professionals.

3.1. Hypothesis. To obtain a clear description of the model's concerns, we give some assumptions as follows. In that, assumptions (i) and (ii) are requisite, and assumption (iii) is potential.

(i) Attackers hold some exploit kits which contain remote code execution or privilege escalation vulnerabilities; they act on web browsers or other programs. In fact, there is much proof of concept code for web browsers and their plugins on the Internet, such

as Metasploit Framework [36] and Exploit kits [2]. It is a challenge to obtain unknown or zero-day vulnerabilities.

(ii) Code integrity protection exists in the target system. In other words, the attacker is unable to modify existing code on the operating system using technologies such as hooking.

(iii) The attacker can fake Wi-Fi hotspots and then control the communication of victim who connects to the hotspot. Free Wi-Fi hotspots are pervasive in the city, so attackers can deceive the users easily based on the flaw of the Wi-Fi protocol.

3.2. Model Description. PMCAP makes efforts to alleviate the influence on the target system and takes aim at the network related object structures in the live process memory. It has two advantages. (i) When searching, we have a clear goal instead of string matching in the entire memory address space, so as to improve efficiency. (ii) Object structures, with the shape of the expanding root of a tree, can indicate the data content in a wide range. Starting from data structures, we can obtain more useful data at a lower overhead. First, we give the definitions of PMCAP model as follows.

Definition 1 (victim). A victim is denoted as the four tuples as shown in the following formula:

$$\mathrm{VR} = (\mathrm{Ver}, \mathrm{GP}, \mathrm{HP}, \mathrm{TP}), \tag{1}$$

where Ver is a set of version information, including system modules and programs on the victim's computer. GP is a process set which can trigger the exploits and then cause a remote code execution. HP is the host process set; attack code can inject itself into it for runtime persistence. TP is the target process set, from which attack code will extract data.

Definition 2 (attacker). An attacker is denoted as a tuple as shown in the following formula:

$$\mathrm{AR} = (\mathrm{Exp}, \mathrm{Payload}, \mathrm{Loc}), \tag{2}$$

where Exp is the exploit collection owned by attackers; Payload is the set of attack code, which perform the task of data extraction in different ways. Loc denotes the location of the attacker and decides whether the attacker can control the victim's network or not.

Definition 3 (PMCAP). PMCAP model is denoted as a tuple as shown in the following formula:

$$\mathrm{PMCAP} = (\mathrm{VR}, \mathrm{AR}, \mathrm{ENV}), \tag{3}$$

where VR is the victim including the pertaining resources; see in Definition 1. Similarly AR represents the attacker; see Definition 2. ENV is the network environment for the attacker and victim; it decides if there are attacking paths between them.

Then from all the elements enumerated above, we can analyze the threat of the model by the Model Threat Value (MTV), which is defined as follows. From its definition and calculation method we will realize the key components in the model.

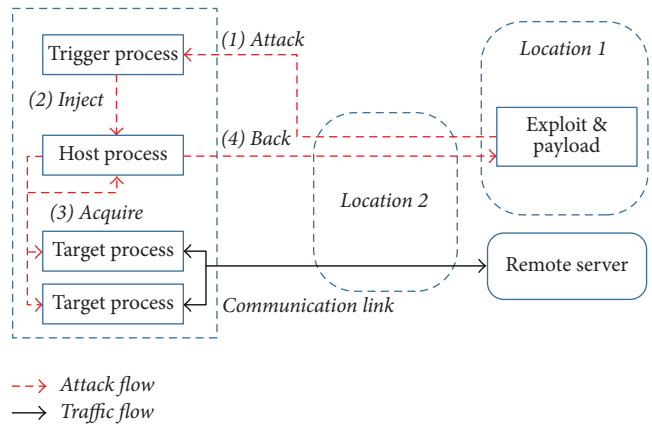

Figure 2: Intuitive view of PMCAP model.

Definition 4 (MTV). MTV can measure the threat engendered by the concrete attack; the calculation method is shown in the following formula:

$$\mathrm{MTV} = f(\mu, \mathrm{PC}, \alpha), \tag{4}$$

where f is the calculation function; we do not give the specific definition for it can be custom-defined according to requirements. μ is the network factor, which depends on the ENV and the Loc. PC denotes the capability of the attack code, which depends on the concrete Payload in Payload. α is the stage flag; the flag denotes the current stage of the entire attack process. For example, if GP is empty, the victim does not satisfy the attack conditions; if TP is empty, the attack code can not extract the data even if it has already been executed on the target system.

A more intuitive description of the model is shown in Figure 2. The left side shows the victims system which contains a series of active processes with network activities. The right side includes the normal web server and the malicious server which is controlled by the attacker. The two locations denote the possible positions of the attacker. The attack flow is implied by the dashed arrow in this figure, and the solid arrow denotes the normal communication flow of target processes.

The dashed arrow and the solid arrow show the highlighted parts of PMCAP. We divide the attack process into four steps. First, the attacker tries to compromise the target website and deploy the exploit in Exp subsequently. The process in GP running on the victim's system triggers the vulnerability exploit; mostly the process would be a web browser. Second, the attack code in Payload executes and considering that the trigger process may crash injects new code into the memory space of the host process in HP sought by the attack code at the same time. Third, the attack code hunts for target processes in TP and acquires the live memory data. Finally, the attack code transfers data back to the server

controlled by the attacker. There are three types of process on the victim's system. The trigger process GP is often web browser process or other network related processes. The host process HP can be a similar process or other processes with a long lifetime. The target process TP is the process which contains sensitive data, such as a web browser or instant message software. Specific description is as follows.

(1) Attack. The attacker leverages the vulnerable software and constructs the exploit and then makes the target computer trigger the attack code. The attack has been proved by many incidents in real world [1, 2]. As shown in Figure 2, if the attacker situates in *Location 1* that is on side of the communication channel, a third party website is needed to accomplish remote attack. If the attacker situates in *Location 2*, the MITM attack will play its role by reason of network control. The attacker aims at the HTTP session when the victim is connected and then is able to choose web browser or other types of software with remote execution vulnerability as the attack target. Besides, arbitrary memory read vulnerabilities may be not be suitable for the model. One of the reasons is that it can not read memory data across processes. Nearly all web browsers support multiprocess. The other reason is that the exploit code will disturb the running status of the process and even lead to a crash.

(2) Code Execution. The application on the victim's system triggers the exploit, after that the attack code will be executed. The attack code faces two challenges. One of them is the code permission problem. Mandatory Integrity Control (MIC) provides a mechanism for controlling access to objects based on integrity levels upon Windows 7 OS. The six integrity levels defined by MIC, *Installer*, *System*, *High*, *Medium*, *Low*, and *Untrusted*, are stored in an access control entry (ACE) of the object together with object mandatory policy. Object mandatory policies are *No-Write-Up*, *No-Read-Up*, and *No-Execute-Up* [25]. Most applications on the system are assigned Medium integrity label which inherited from the *explorer* process. From Figure 3 we see that the *No-Read-Up* and *No-Write-Up* policies can not stop the Medium integrity process from reading and writing the process at the same integrity level or lower. This means that if the attack code obtains Medium integrity level, it can access resources of most processes. For example, when Internet Explorer is running in protected mode, its process is assigned Low integrity, so any other process can read its memory data. MIC has not provided perfect data protection for process memory. The other problem is the attack code form. When some protected process is forbidden to create a child process (e.g., Child Process Policy [12]), code injection method can also be used. Such method is widely used in both security software and malware [1].

(3) Data Extraction. Acquiring and extracting live memory data are the core function of the attack code. The model pays more attention to the memory data related to network; these data are more volatile and variable. Other data (e.g., cache data, user interface data) will maintain for a longer period in memory and are much easier to extract. Traditional malware often uses runtime hooking method to obtain target

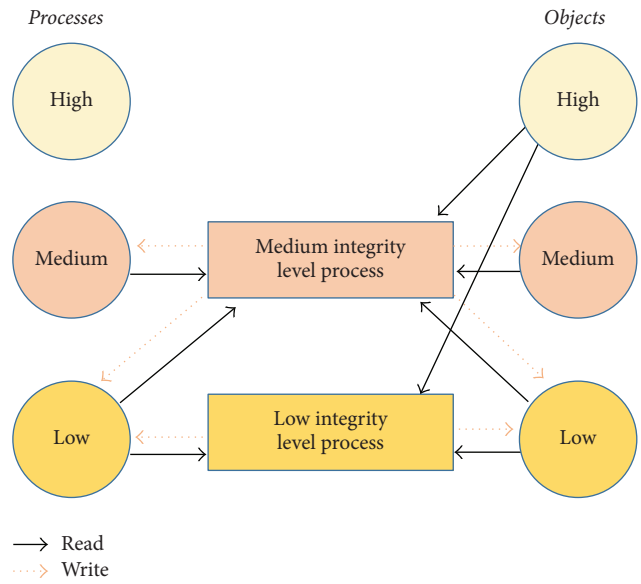

Figure 3: Access to processes versus objects for Medium and Low integrity level processes [25].

information [1, 2], but our model does not take this method for the following two reasons. (i) Function hooking needs the memory write permission and will violate the code integrity policy and disturb the origin work flow. (ii) With hooking, lots of data will be intercepted and processed, which will lead to higher costs. In an extreme case, our model only needs process memory read permission. Several data acquisition methods are supported in PMCAP. We will introduce and discuss them using web browsers as examples in Section 4.

(4) Network Control. As shown in Figure 2, if the attacker situates in *Location 2*, the MITM attack will help to improve the threat level. It should be common since there are many Wi-Fi hotspots without authentication or with a known password in the public place (e.g., cafe, hotel, and airport). In this case, we can reduce network flow rate to increase the lifetime of network related data in memory, even if the target process encrypts communications using the SSL/TLS protocol, since we do not need to know the packet formats and only need to adjust the speed of packet forwarding. Another benefit is that PMCAP can use normal communication sessions to transfer extracted data.

Furthermore, there are a few challenges in real implementation.

(1) Process Privilege Problem. Most applications running on Windows have a Medium integrity level label; in order to read another process memory the attack code should be at Medium integrity level at least. Actually it is not very difficult to achieve the goal, as a lot of third party software including security software may have code execution vulnerability [37]. We can also see that Low integrity process can access the Internet Explorer process running in protected mode. The browser runs at Low integrity level and makes it much easier for it to be accessed by others.

(2) Search in Large Address Space. The size of a 32-bit process virtual address space is 4 GB and 2^{64} bytes for 64-bit process. It is difficult to traverse all of this. In fact, since it is only necessary to search in the allocated memory space of target process and in user mode, the target searching range is still rather limited. Therefore, this paper proposes a stack based data extraction method to avoid this problem. We will describe it in the next chapter.

(3) Attack Persistence. The main problem is how to discover the target process. One direct method is to enumerate the process identify (PID) on the system, because the value of PID has a limited range. Another way is to use the system API function such as *CreateToolhelp32Snapshot* which can be called from any integrity level. Enumerating the windows related to threads is also a good method.

Compared with other methods, PMCAP has the following characteristics. (i) It is an extension based on the attack of vulnerability exploits and malware and takes full advantage of the imperfection of access control mechanism on the Windows platform. (ii) The attack Payload for data extraction is live on the system. It does not corrupt the code integrity of the target program and only needs memory read permission in the extreme case. (iii) It pays more attention to the network related data in the process memory which can be impacted by network control.

4. Implementation

Some techniques involved in the model, such as vulnerability exploit and attack method, can be frequently found being researched and analyzed in many other papers. The implementation will concentrate on data extraction in the target process memory. We take IE/Edge, Chrome, and Firefox as the target programs and choose three class object structures as the target data for extraction. They are *HTTP_REQUEST_HANDLE_OBJECT* in IE/Edge, *URLRequestHttpJob* in Chrome, and *nsHttpTransaction* in Firefox which contains abundant HTTP session information. The reasons are stated as follows. (i) Web browsers, as typical applications, are commonly used by people and are also most frequently attacked by vulnerability exploits. (ii) Source code and debug symbols of these programs can help us to analyze object structures rapidly. In fact, PMCAP is generic on the Windows operating system, which threatens the process memory data of not only web browsers but also other applications.

4.1. Process Memory Acquisition. Since the process may crash after triggering the exploit on the victim's system, the attack code should try to create a new process as a host with current access token and then inject attack Payload into the new process memory space for runtime persistence. It will complicate matters if the trigger process is not allowed to create a child process. We have to search for another process which has the same integrity level as the trigger process; sometimes the case will never happen.

Modern browsers use a multiprocess architecture which means that each browser tab will run in a separate process.

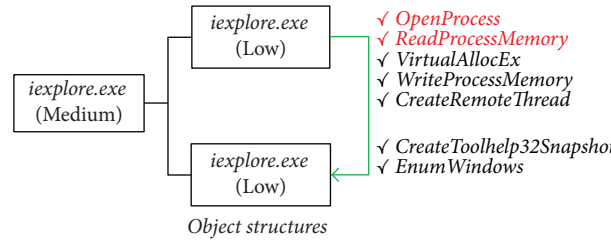

Figure 4: Capabilities of IE process at Low integrity level.

This mechanism can improve the browser's reliability since the crash of tab process will not impact the entire browser. To avoid vulnerability exploits, tab process always runs with a lower privilege. If the attack code can obtain the *Medium* integrity level permission, it will be easier for it to access the browser processes. Firefox chosen in this paper uses single-process mode by default, and its latest version supports multiprocess mode. IE/Edge and Chrome use multiprocess by default.

IE/Edge runs in protected mode, and each browser window has its separate process. The network related data which we concerned lay in the tab process. The difference between IE and Edge is that the tab process of IE runs at *Low* integrity level and the other runs in the *AppContainer* sandbox. *AppContainer* is a new process isolation environment introduced on Windows 10, which provides fine-grained access control to objects through adding additional access control entry (ACE) in the system access control list (SACL) of objects. However, different processes with the same *AppContainer* security identify (SID) can also access one another, as shown in Figures 4 and 5, while for IE/Edge the attack code only runs with lower privilege, it can also read network related data in memory and does not need to escape from the sandbox.

Unlike Microsoft's products, the main process of Chrome manages the tabs and processes the network communication at the same time. It runs at *Medium* integrity level, the GPU process runs at the same integrity level, and the render processes run at *Untrusted* integrity level. If the attack code runs at the latter two integrity levels, it can not read the main process due to the MIC mechanism. As shown in Figure 6, to acquire the network related data in memory, the attack code must obtain at least the *Medium* level permission. On the other hand, the render processes can not access one another. Compared with IE, the architecture restricts the capability of vulnerability exploit for the render process.

4.2. Data Extraction. This paper presents three methods for memory data extraction, which we describe as follows.

(1) Search Based on the Vtable Pointer. The modern software development generally uses Object-Oriented Programming (OOP) method. Many classes define virtual function because of the heavy use of inheritance and polymorphism. The compilers will add a hidden member pointer at the beginning of the object. The pointer points to an array which contains all virtual function pointers; this is called virtual method table (vtable). Most compilers store the vtable and vtable

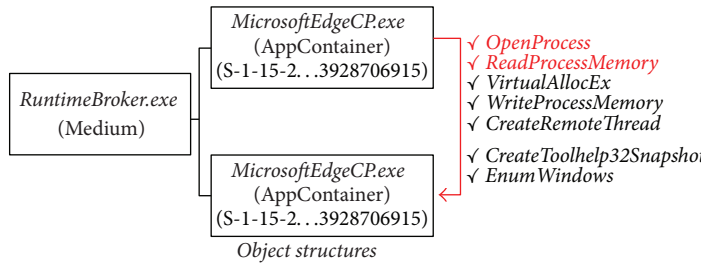

Figure 5: Capabilities of Edge process at AppContainer integrity level.

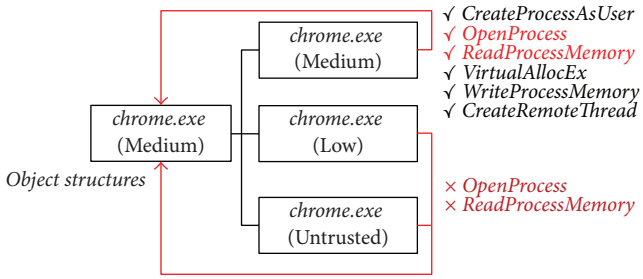

Figure 6: Capabilities of Chrome processes.

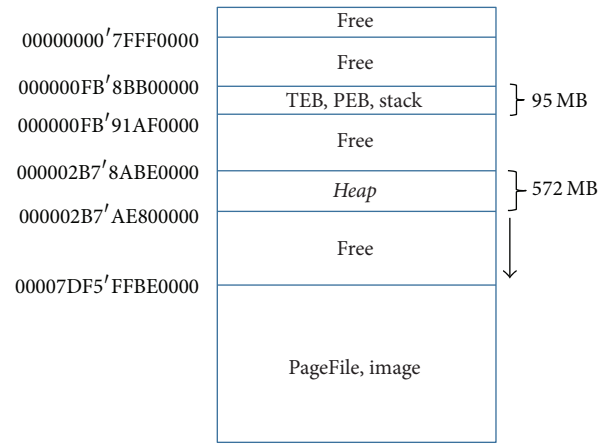

Figure 7: Example of ×64 process memory layout.

pointer in the binary file as a global variable. In the exploit of vulnerabilities the vtable is always used to leak the module address to bypass the ASLR protection [38]. On the contrary, if we know the version and the image base address of the target module, the address of vtable pointer can be calculated at runtime. Therefore it is possible to mark the object by vtable pointer and then search for object structures in heap memory based on vtable pointer. Using only vtable pointer as a characteristic may cause a high false alarm rate. The size of the pointer is 4 bytes or 8 bytes, which is too small. We can choose an additional characteristic in the object structure; the characteristic should be fixed in the specific version of the target module, such as magic number and callback function pointer. In this case, we may achieve a low false alarm rate.

In fact, the method of brute force memory scanning has been used in memory forensics [28]. It has to overcome the difficulties of large address space and unclear goals. Although the virtual address space is large especially in 64-bit processes, the size of physical memory is 4 GB or 8 GB in general. The size of process mapped memory depends on its real time usage. An example of ×64 process memory layout is shown in Figure 7. We can see that a large number of virtual addresses are free. The attack code of PMCAP runs in user mode on the target OS, so it can not get the Virtual Address Descriptor (VAD) of target process though *_EPROCESS* structure [15]. We can still use *VirtualQueryEx* to get the usage of virtual addresses of the process, which needs the *PROCESS_QUERY_INFORMATION* access right and is not prevented by MIC. In this case, the search range is limited, so we can get a better result when searching for a clear goal.

The attack code is side to the target process; ASLR defense has no impact on the data acquisition. However we face a challenge that the number of system and program module versions is rather large [39]. We need to get the precise offsets by additional enumeration and automation techniques.

(2) Search Based on the Global Variable. Applications usually apply linked list, array, and hash table to manage objects, which makes it possible to search object structures based on these data structures that may be pointed by global variables. Similar to characteristic value, search technique based on global constants is used in memory forensics. For example, *PsActiveProcessHead* is a constant which points to the beginning of the process linked list, and hash table *_TCBTable* points to the linked list of connection information [15]. Some vulnerability exploit also makes use of global constants to obtain the variable address in memory [27]. Unlike them, the model in this paper aims at the live memory data; the technique will encounter a dynamic continuous memory environment.

In software development singleton pattern and static variable will be converted to global variable by the compiler. Towards the open sources software, we can locate the variable through keywords like *GetInstance*, *static*, and so on. For binary programs we have to do additional analysis in the data segment. The global variable has the fixed offset from the beginning of binary module the same as vtable pointer.

The search method based on global variable can avoid searching in the whole address space but also has limitation. Fortunately, the web browsers we chose satisfy the condition,

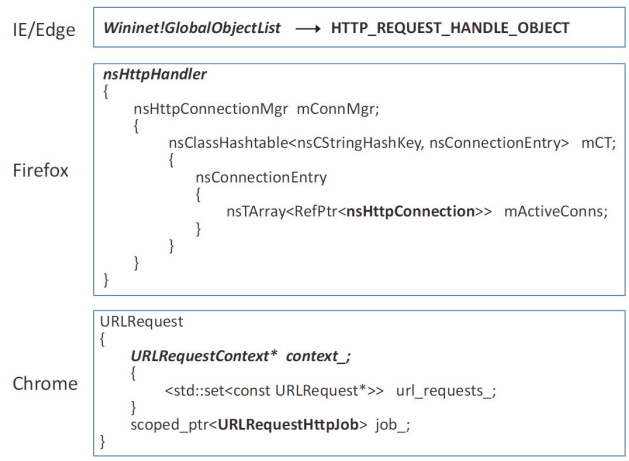

Figure 8: Relations between the global variable and target objects.

with each program having a little difference. As shown in Figure 8, *HTTP_REQUEST_HANDLE_OBJECT* objects are linked by a double linked list which is referenced by a global constant *Wininet!GlobalObjectList*; we can read the value directly through an address offset corresponding to the module version.

There is no global constant pointing to *nsHttpTransaction* objects; *nsHttpTransaction* appears as a member variable of class *nsHttpConnection* which is referenced by class *nsHttpConnectionMgr*. Class *nsHttpConnectionMgr* is also a member variable of *nsHttpHandler*; *nsHttpHandler* is defined as a singleton pattern which has only one instance at runtime. We can also get *nsHttpTransaction* objects through the *nsHttpHandler* instance after some jumps.

URLRequestHttpJob objects are also not organized by a variable directly; the instance of *URLRequestHttpJob* is a member variable of class *URLRequest*. Meanwhile *URLRequest* has a member variable which points to the instance of *URLRequestContext*. Class *URLRequestContext* manages all *URLRequest* objects, but its instance is created at runtime, and its value can be found using the aforementioned method in this paper.

We can see that three cases are sketched in Figure 8, including three type data structures: double linked list, hash table, and red black tree (*std::set*). These structures dynamically change at runtime, which will bring a challenge to the search procedure. In particular, *std::set* uses red back tree as an internal implementation. Its insertions and deletions will cause a subversive structure change, but our experiments show that it has limited effects.

(3) Search Based on the Stack Space. This approach tries to discover the address of the target object in the thread stack space and then extract the detailed object information in the entire address space. The motivation is to avoid searching in the large address space. The premise of the method is that there are lots of leaks of code address and argument address on the thread stack at runtime; commonly a return address is saved on the stack. The range of stack address is limited,

```
.text:0000000180E4D164 sub_180E4D164   proc near           ; CODE XREF: sub_180E50460+BA↓j
.text:0000000180E4D164                                     ; sub_180E50C10+2F3↓p
.text:0000000180E4D164                                     ; DATA XREF: ...
.text:0000000180E4D164
.text:0000000180E4D164 var_F8                  = dword ptr -0F8h
.text:0000000180E4D164 var_F0                  = byte ptr -0F0h
.text:0000000180E4D164 var_E8                  = qword ptr -0E8h
.text:0000000180E4D164 var_E0                  = byte ptr -0E0h
.text:0000000180E4D164 var_DC                  = dword ptr -0DCh
.text:0000000180E4D164 var_C8                  = byte ptr -0C8h
.text:0000000180E4D164 var_B8                  = byte ptr -0B8h
.text:0000000180E4D164 var_8                   = byte ptr -8
.text:0000000180E4D164
.text:0000000180E4D164                         mov     rax, rsp
.text:0000000180E4D167                         mov     [rax+10h], rbx
.text:0000000180E4D16B                         mov     [rax+18h], rsi
.text:0000000180E4D16F                         mov     [rax+20h], rdi
.text:0000000180E4D173                         push    rbp
.text:0000000180E4D174                         lea     rbp, [rax-18h]
.text:0000000180E4D178                         sub     rsp, 110h
.text:0000000180E4D17F                         mov     rax, cs:__security_cookie
.text:0000000180E4D186                         xor     rax, rsp
.text:0000000180E4D189                         mov     [rbp+0], rax
```

Figure 9: A code snippet from chrome.dll.

so the search range is greatly reduced. The major challenge we face is that data only stay on the stack for a short time. The address information on the stack mainly comes from function return addresses, function arguments, and local variable, also including the history value of register at runtime brought by the compiler. A case is shown in Figure 9, where the compiler does not care about the value propagation, so it is possible that register values are pushed into the stack.

We need to focus on the phenomenon that the arguments passing between 32-bit and 64-bit program are different. Mostly the 32-bit program uses the stack to pass arguments. Even when using *fastcall*, it only passes the first two arguments by *ecx* and *edx*. Therefore there will be much address information on the stack. The 64-bit program uses *rcx*, *rdx*, *r8*, and *r9* registers to pass the first four arguments [40]; this will reduce the number of address appearing on the stack.

We propose the Thread Stack Search Algorithm (TSSA) for PMCAP. First we give two definitions for describing the algorithms:

Definition 5 (anchor point). It is a position where the address of code appears on the stack. The address value remains stable on the lifetime of process and is only impacted by ASLR and program version, for example, function pointer address or return address.

Definition 6 (target point). It is a position where an address appears on the stack; through the address we can directly or indirectly get the target object information. The distance is defined as the offset between the Anchor Point and Target Point on the stack.

Then we can obtain the target object structure information based on the *Anchor Point*. It is a simple scheme to choose a return address as the *Anchor Point*, and then there can be multiple *Anchor Points* at the same time. When the value associated with *Target Point* is fixed we can extract the *Target Point* prior from the binary program and calculate the distance between them. An example is shown in Figure 10. We design two methods.

First, we can adopt simple taint trace method on the program execution path related to the target object. There are some excellent helpful tools, such as the IDA Pro and its plugins. For object structures, we can choose the value of register *ecx* or *rcx* as the taint source and simply trace the

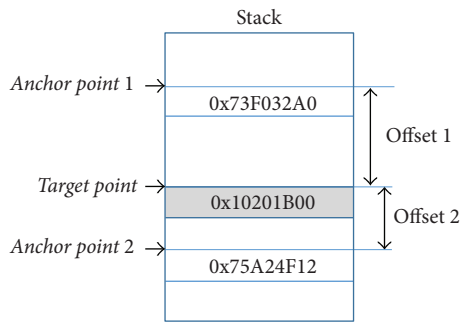

Figure 10: Example for the *Anchor Point* and *Target Point* on the stack.

value propagation on the instructions: *mov*, *lea*, *pop/push*, and so on. We do not consider the impact by multiplication, division, shift instructions, and the truncation of registers for now. This method can not assess the time that the value stays on the stack, because some value may be ephemeral.

Another method is based on the binary dynamic analysis. We can debug and instrument the target binary program using debuggers and trace tools and insert extra code before and after each *call* instruction. As shown in Figure 11, before the *call* instruction the code checks if there is an object address value on the current stack frame. When the *call* instruction returns, the code counts the instructions of the following code and estimates the overhead of the execution. Then we can approximately obtain the *Anchor Point* which makes the object address stay on the stack for a relatively long time. It is approximate, because there is the impact by thread switches and analysis code self. In this case the value associated with the *Anchor Point* is a return address, so before the value is popped out the stack we can gain a stable stack frame during the following code execution of target process. The search code can obtain the target object address value in a rather short time period. One ideal case is that there is some code receiving the socket data in the following code, so a transfer delay may help to expand the time period.

When the *Target Point*, the *Anchor Point*, and the distance between them are confirmed, we use Algorithm 1 to obtain the object address values in the process memory at runtime and then can extract the data in an expansive region. First, the algorithm will get the thread stack address range which is contained in thread TEB structure. It is easy to find the PEB and all TEB structures in the process memory in side channel and then pick the target TEB by a thread stack feature. The thread stack feature is related to the target objects; often thread function address can be used. This means that the thread deals with the target object data. Then the code continuously searches on the stack in a loop; if some value matches the value on the *Anchor Point*, the target object address can be acquired in the position as a relative offset to the *Anchor Point*.

Since our model acquires the process memory inside the operating system, it can also use system call to get the addresses of TEB structures. We can also choose multiple *Anchor Points* to reduce the false alarm rate. By analyzing the practice of the three web browsers, we know the target objects are network related. IE/Edge processes the network data with multiple threads and the others take a single thread. It may produce a high overhead when tracing the stacks of multiple threads, because the threads are created and destroyed dynamically. However interestingly, we observe that the multiple threads of IE/Edge are linked together and pointed by a global constant, so we can rapidly capture the changes of the threads.

5. Evaluation

We take real experiments to measure the effect of the model. The experiments focus on the data extraction of different methods, as there are many vulnerability exploits in practice. Firstly we assess the practicability and then give two case studies. The target systems run on the VMware. They are 64-bit Windows 7 SP1 with an Intel(R) Core(TM) i7-6700 CPU @ 3.40 GHz and 4 GB of RAM and 64-bit Windows 10 6700 with an Intel(R) Core(TM) i7-6700 CPU @ 3.40 GHz and 8 GB of RAM; the network bandwidth is 20 Mbps. Actually, for compatibility reasons, the 32-bit web browsers are more widely used. The target web browsers are 32-bit IE11 (11.0.9600.18524), Firefox (48.0.2), Chrome (51.0.2704.103), 64-bit Chrome (51.0.2704.103), and Edge (38.14393.0.0).

5.1. Lifetime of the Target Objects. We measure the lifetime of the three types of objects in the process memory and analyze the network delay influence on it. For getting the time of corresponding object constructed and destructed, we inject extra code into the process memory of the browsers. We record the time data by using a Lock-Free queue [41] to reduce the influence on thread switches of origin processes. We take experiments of the five browsers under the 20 Mbps and 2 Mbps network bandwidth, respectively. We generate nearly 50 thousand objects by visiting lots of websites both at home and abroad, including different categories such as Search, News, Shopping, E-mail, SNS, and Finance.

As shown in Figure 12, we divided these objects of each browser into three parts by the duration, namely, less than 50 ms, between 50 ms and 1 s, larger than 1 s. It may be difficult to obtain the objects with a lifetime less than 50 ms, because Windows uses a preemptive thread scheduler. Our code runs in user mode. It is hard to control the execution time without interfering with others, because for most multiprocessors the thread quantum is about 15 milliseconds [25]. In Figure 12, the left bar of each group shows the object distribution on the normal network bandwidth and the right on the limited. We can see that the lifetime increases significantly with the reduction of the network bandwidth, since they are network related. Another interesting phenomenon we observed is that a mass of objects will reuse the addresses which have already been allocated in the memory. It means that we can obtain more object information from fewer object addresses.

5.2. Vtable Search. We continuously measure the effect of different search method and mainly inspect three indicators: (a) *detection rate* (DTR), this denotes the result that extracted objects cover the target objects really, (b) *false alarm rate*

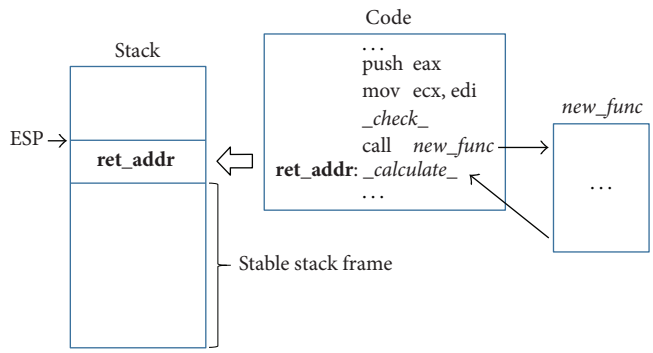

Figure 11: Check object address on the stack and estimate the duration.

Input: P: the target process set, TF: the feature of thread associated with target data, AP: the anchor point set, OT: the distances between anchor points and target points;
Output: *results*: Extracted data from the target process memory;

```
(1)  function SearchThreadStack(P, TF, AP, OT)
(2)      st_list ←GetThread(P, TF);
(3)      for all stack in st_array do
(4)          base ← stack.sb_addr;
(5)          Reading the stack buffer buf of stack;
(6)          s ← stak;
(7)          while s < base do
(8)              if s matches AP.value then
(9)                  t ← s+AP.offset;
(10)                 Extracting data from t;
(11)                 Saving data to results;
(12)             end if
(13)             s + +;
(14)         end while
(15)     end for
(16)     return results
(17) end function
(18)
(19) function GetThread(P, TF)
(20)     Finding PEB address peb;
(21)     Finding TEB addresses teb_array based on peb;
(22)     for all teb in teb_array do
(23)         Reading the stack base sb_addr from teb;
(24)         Reading the stack limit sl_addr from teb;
(25)         calculate size and read the stack buffer s_buf;
(26)         p ← sl_addr;
(27)         while p < sb_addr do
(28)             if p matches the value in TF then
(29)                 save sb_addr and sl_addr to threads;
(30)             end if
(31)             p + +;
(32)         end while
(33)     end for
(34)     return threads
(35) end function
```

Algorithm 1: Thread Stack Search Algorithm for PMCAP.

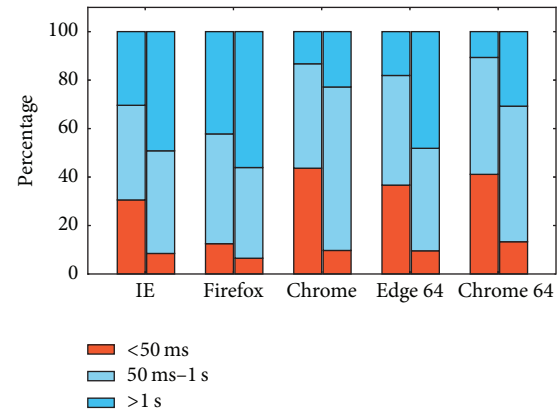

Figure 12: Lifetime of objects under different network environment.

(FAR), it denotes the interference of the irrelevant objects in the extracted objects, and (c) *average CPU time* (ACT), it is the CPU usage of the search thread on the target system. The method based on vtable pointer characteristic searches in the virtual memory space. Its efficiency will be impacted by the real memory usage, in which web page content is the major factor. Almost all modern browsers almost have a good garbage collection mechanism, so the memory usage is stable at runtime. The search code runs in a single thread in the experiments; its maximum CPU usage is about 25% on a quad-core environment. The overhead is acceptable and has a relatively low influence on user experience.

In Table 1 we show the search results of the test browsers under different network conditions. The thread switch can bring deviation. Since we can not precisely control the CPU usage of the search code, we can only get an approximate value. In the condition of normal network bandwidth, the detection rate is near 100% for each program with the search thread running at full capacity. We keep the CPU usage of the search thread at below 10%, and the detection rate keeps up, but the false alarm rate will increase slightly, as shown in the row named *Normal b* group in Table 1. It caused by the low CPU usage; then the search code can not fast capture

Table 1: Search results based on vtable feature in different conditions.

Program	Group	TPR (%)	FNR (%)	ACT (%)
IE	Normal a	100	1.72	24.83
	Normal b	100	21.35	4.06
	Limited	100	4.28	4.38
Firefox	Normal a	100	7.22	24.64
	Normal b	100	22.92	9.84
	Limited	100	7.64	4.73
Chrome	Normal a	100	8.95	24.70
	Normal b	100	11.91	6.97
	Limited	100	3.91	8.20
Edge (64)	Normal a	100	3.43	24.68
	Normal b	100	16.73	5.07
	Limited	100	0.58	5.20
Chrome (64)	Normal a	100	3.35	24.52
	Normal b	100	21.98	6.14
	Limited	100	1.48	8.71

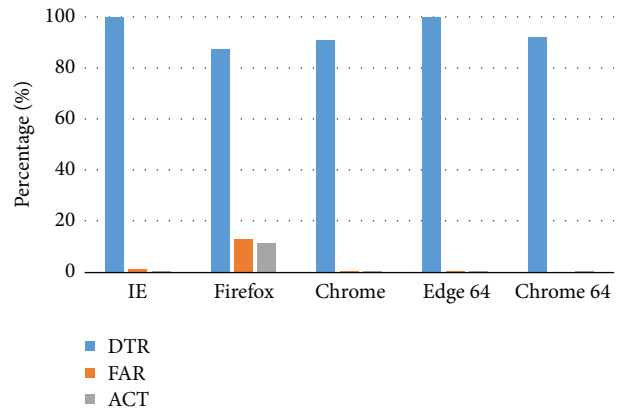

Figure 13: Results based on global variable search.

the transition of target objects. To further verify the network bandwidth influence, we limit the bandwidth to 2 Mbps and at the same time keep the CPU usage. In this case we also get the high detection rate and low false alarm rate, as shown in the third row in Table 1.

The values of false alarm rate are all low, which perhaps is because we choose an additional characteristic besides the vtable pointer, mentioned in Section 4. Furthermore the results of the 64-bit program are similar to that of the 32-bit program. We think it is because the code only searches in the mapped memory address space rather than in the entire space.

5.3. Global Variable Search. We can obtain a high-efficiency search based on global variable because we have definite start address and search direction. It is a special case but we think it should appear frequently in modern programming. In this experiment we do not limit the network bandwidth. To limit the CPU usage of the search thread we add a short sleep in the code after each traversal. The results are shown in Figure 13. We get a high detection rate, a low false alarm rate and a low CPU usage for each experiment. There is a low false alarm for IE/Edge because the linked list structure is stable. The detection rate for Chrome is slightly lower than others that is because of the structural changes caused by insertion and deletion of the red black tree. For the case of Firefox, we infer that the path from the global pointer to the target object is too long. When the browser sends requests frequently, it may cause unpredictable errors and a higher CPU usage.

5.4. Stack Space Search. In the experiment we only choose one *Anchor Point* for each browser, which is a requirement easy to meet in practice. More remarkably, the *Anchor Points* between 32-bit and 64-bit versions are different. Since the target object address only exists on the stack for a short time, to better measure the effect of search method we carry out experiments under different CPU usage. The network bandwidth has a limited impact on the duration of target objects on the stack because the browsers all use asynchronous communication and the network thread does not wait for the server to respond.

We experiment on two cases respectively for the browsers. In case (a), we do not control the CPU usage, so the search thread gets a 25% average CPU time under a quad-core environment. As shown in Figure 14, in this case we get a high detection rate for all browsers. In case (b), we keep the CPU usage to below 10%, and it may be lower in practice. In this case we can also get a detection rate around 80%. Actually we can increase the CPU usage of search thread to obtain higher detection rate or adopt more effective thread scheduler algorithm.

Another obvious case shown in Figure 14 is that the detection rate is low in 32-bit Chrome, but it is normal in 64-bit Chrome. After analyzing, we discover that it is caused by the *Anchor Point* in 32-bit Chrome. We have more work to do on the optimization of the *Anchor Point* selection.

The experiment results above show that the search method based on the global variable has the highest efficiency but requires meeting specific requirements. The other two methods have a similar good effect. The search method based on stack is more generic than the others. It supports the search not only for object structures but also for the arbitrary buffer referenced on the stack. On the other hand, the stack search method maybe has the higher cost of binary analysis than the other two methods. So the three methods can be used together, according to the specific situations in practical.

5.5. Case Study

(1) Get Login Information through HTTP Objects. First, we take a case study on the login procedure of outlook webmail and try to get login information through the HTTP objects mentioned above. One reason for choosing this case is that, in here, the speed of outlook login is relatively slow and can show clearly how network bandwidth affects the memory data. Another reason is that although it transfers under the HTTPS protocol, outlook does not encrypt POST data additionally and that means the password in the HTTPS

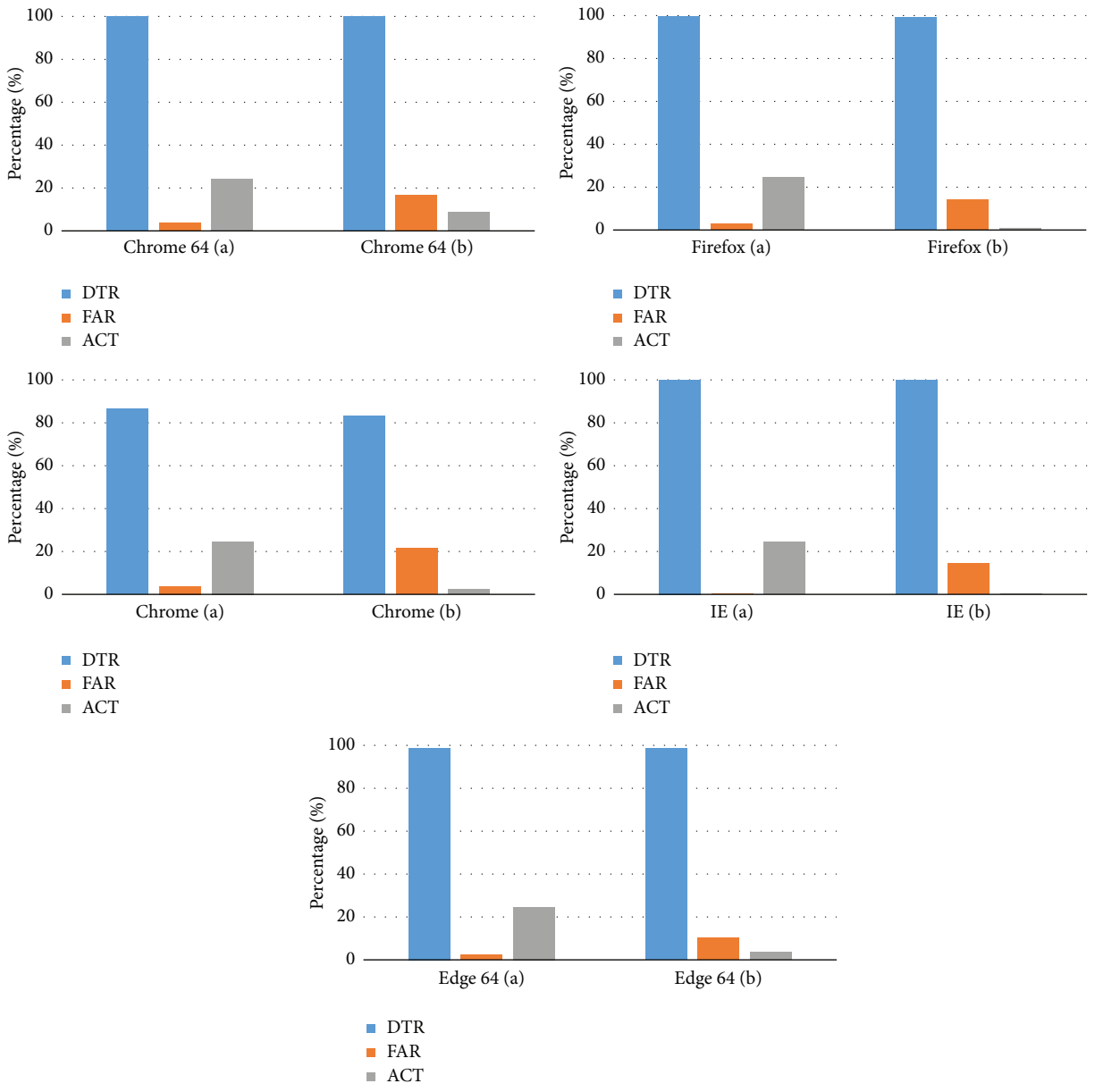

Figure 14: Results based on stack search.

Table 2: Fields obtained from HTTP object structures.

Field	IE/Edge	Chrome	Firefox
URL	√	√	√
Cookie	√	√	√
POST body	×	√	√

channel is a plaintext. Therefore we can get the plaintext of password through the object structures.

The results are shown in Table 2. It shows with multiple trials that the success rate is high, almost near 100%. The average CPU usage of the search thread is about 12%, which not only gets the plaintext in the POST data but also URLs and Cookies related in the login procedure. However, IE/Edge is an exception for the POST body data, because the data are not directly associated with the *HTTP_REQUEST_HANDLE_OBJECT* structure and are sent to the server as a function parameter.

Further we explore the HTTPS communication of IE/Edge. The POST body data is sent through the *wininet!CSecureSocket::Send* function; then it is encrypted in function *wininet!CSecureSocket::EncryptData* and sent subsequently. Therefore we can get the send buffer based on the method described in Section 4. Figure 15 shows a stack layout example of 64-bit Edge at runtime. We can easily get an *Anchor Point* and locate the send buffer and then extract the POST body data including the outlook login password.

A similar case is also in the login procedure of Skype which has been purchased and reformed by Microsoft. In the

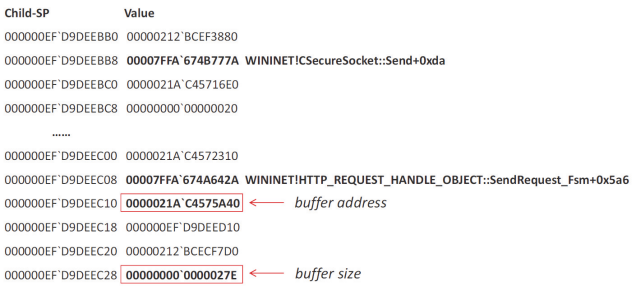

Figure 15: An example of stack frame for 64-bit Edge at runtime.

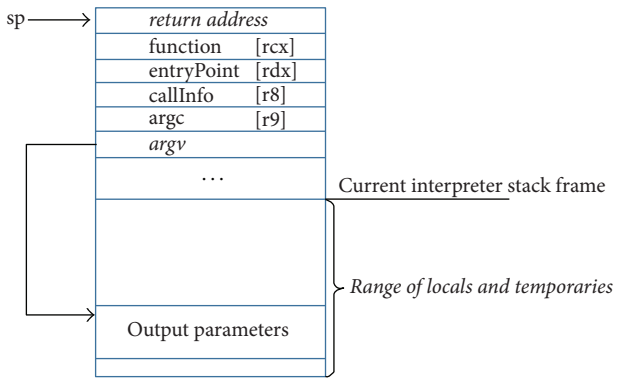

Figure 16: Stack layout when step into amd64_CallFunction.

new version of Skype on Windows 10, *Skypeapp.exe* processes the login using the *WININET* library. The login data is sent by the *WININET!CSecureSocket::Send* function without additional encryption. We can also extract the plaintext including the login name and password in the send buffer.

(2) Get the String Trace of the Chakra JavaScript Engine. Chakra is a JavaScript engine developed by Microsoft. It is integrated into the Edge browser and its core components have become open source [42]. Same with other Just-in-time (JIT) engines, Chakra provides an interpreter for JavaScript byte code and also compiles byte code into machine code just in time for an optimization. Chakra engine constructs an interpreter stack frame for JavaScript code, which provides convenience to the search based on stack space. We discover that there is a stable function call in the Chakra engine when JavaScript function is called; for 64-bit platform it is *amd64_CallFunction*. The stack layout when executing *amd64_CallFunction* is shown in Figure 16. In a 64-bit platform the first four arguments are passed through registers in function calls. The *amd64_CallFunction* has five arguments, so the last is stored in the stack, which is shown in Figure 16 as *argv* with italic. The argument is a pointer that pointers to an argument list which will be passed into the next JavaScript function. In the implementation of Chakra, the address of the argument is in the range of locals and temporaries in the current interpreter stack frame. Therefore we can reference the output parameters and locals through the last argument. In this case, we try to extract the strings stored in the

Table 3: Fields obtained from HTTP object structures.

Website	Success rate
www.baidu.com	high
www.icloud.com	high
mail.aliyun.com	high
mail.163.com	low
mail.yahoo.com	null

LiteralString and *LiteralString* variables in *JavascriptArray* and *DynamicObject* defined in the Chakra engine; then we pick the login information in the string trace record.

We select five websites as the target in the experiment and count the success rate of password acquisition. The average CPU usage of Payload is suppressed to about 12%. The results are shown in Table 3, where some websites have a high success rate of nearly 100%. However, there is also low success rate, especially for Yahoo. We think that it is because although the JavaScript code executes fast in an interpreter procedure, in most cases, there are no system calls involved, so it is difficult to get all information. If a website provides additional validation and encryption in the login procedure, it will bring heavy JavaScript code and data propagation into the code. We will get a high success rate in this case and opposite results for other cases. We are not able to get the password in Yahoo's login procedure in the experimental condition, as its password is simply posted to the server as a plaintext.

Since we trace the string information of JavaScript code dynamically, the log contains the semantic information of the login context. From the log we can pick the password easily; an example for iCloud is shown in Figure 17.

6. Conclusion

In this paper we propose a new threat model of process memory data on the Windows platform. The core idea of the model is to take full advantage of the imperfection of current access control mechanism to acquire the live memory data through critical data structures. The impact of network bandwidth is considered in the model as well. We also design several data extraction methods for the model, especially the search method based on stack space. We implement and verify our model through several popular web browsers.

In our model the function of data acquisition is deployed on the OS, so it can use limited system calls to avoid information inferring to some extent. However, it strives for the CPU usage. This paper demonstrates the threat of the HTTP communication structure of browsers, and its impact on other kinds of information such as key structure. The data extraction methods can also be applied to virtualization environment and can provide a threat to virtual private servers.

There are several restricted conditions on the application of our model; for example, code execution on the target system depends on the remote code execution vulnerability or preinstalled malware. Also, we face challenges of attack persistence and process data encryption and isolation, even

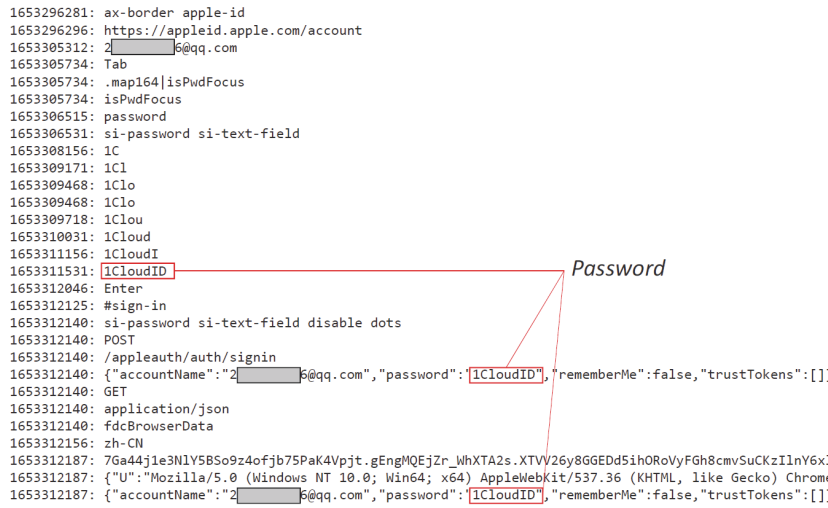

Figure 17: Part string trace log of iCloud login procedure.

though some protection mechanisms are not widespread. Moreover, better automation methods of data structure analysis for target programs are need. Nevertheless, we show the threat of the model through real experiments. The existing research results of memory protection can mitigate the threat to some extent, but we think that more defenses should be integrated into the system. For example, we can introduce more fine-grained permission control mechanisms and take complicate data transformation to reduce the time of plaintext in the memory.

In fact our terminals are faced with cyber threats such as ransomware all the time. We explore the security weakness of the endpoint and hope to enhance its security with more practical defense.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This work is supported by the National Natural Science Foundation of China (General Program) under Grant no. 61572253 and the Innovation Program for Graduate Students of Jiangsu Province, China (Grant no. KYLX16_0384).

References

[1] "Meanwhile in Britain, Qadars v3 Hardens Evasion, Targets 18 UK Banks. Sep 20, 2016," https://securityintelligence.com/meanwhile-britain-qadars-v3-hardens-evasion-targets-18-uk-banks/.

[2] "Lurk Banker Trojan: Exclusively for Russia. June 10, 2016," https://www.securelist.com/blog/research/75040/lurk-banker-trojan-exclusively-for-russia/.

[3] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda, "Panorama: Capturing system-wide information flow for malware detection and analysis," in *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS'07)*, pp. 116–127, usa, November 2007.

[4] Z. Shan and X. Wang, "Growing grapes in your computer to defend against malware," *IEEE Transactions on Information Forensics & Security*, vol. 9, no. 2, pp. 196–207, 2014.

[5] C. Spensky, H. Hu, and K. Leach, "Lo-phi: low-observable physical host instrumentation for malware analysis," in *Proceedings of the 23nd Annual Network and Distributed System Security Symposium (NDSS '16)*, 2016.

[6] V. v. Veen, E. Goktas, M. Contag et al., "A tough call: mitigating advanced code-reuse attacks at the binary level," in *Proceedings of the IEEE Symposium on Security and Privacy (SP '16)*, San Jose, CA, USA, May 2016.

[7] A. Gupta, J. Habibi, M. S. Kirkpatrick et al., "Marlin: mitigating code reuse attacks using code randomization," *IEEE Transactions on Dependable & Secure Computing*, vol. 12, no. 3, 2014.

[8] M. Zhang, A. Raghunathan, and N. K. Jha, "A defense framework against malware and vulnerability exploits," *International Journal of Information Security*, vol. 13, no. 5, pp. 439–452, 2014.

[9] L. Szekeres, M. Payer, T. Wei, and D. Song, "SoK: Eternal war in memory," in *Proceedings of the 34th IEEE Symposium on Security and Privacy, (SP '13)*, pp. 48–62, May 2013.

[10] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz, "Counterfeit object-oriented programming: on the difficulty of preventing code reuse attacks in C++ applications," in *Proceedings of the 36th IEEE Symposium on Security and Privacy, (SP '15)*, pp. 745–762, May 2015.

[11] H. Hu, S. Shinde, S. Adrian et al., "Data-oriented programming: on the expressiveness of non-control data attacks," in *Proceedings of the IEEE Symposium on Security and Privacy (SP '16)*, pp. 969–986, San Jose, CA, USA, May 2016.

[12] "Mitigation bypass and bounty for defense terms," 2016, https://technet.microsoft.com/en-us/security/dn425049.

[13] "TLS heartbeat read overrun (CVE-2014-0160)," https://www.openssl.org/news/secadv/20140407.txt, 2014.

[14] "Mimikatz tools," https://github.com/gentilkiwi/mimikatz.

[15] S. Vömel and F. C. Freiling, "A survey of main memory acquisition and analysis techniques for the windows operating system," *Digital Investigation*, vol. 8, no. 1, pp. 3–22, 2011.

[16] X. Fu, X. Du, and B. Luo, "Data correlation-based analysis methods for automatic memory forensic," *Security & Communication Networks*, vol. 8, no. 18, pp. 4213–4226, 2015.

[17] J. T. Sylve, V. Marziale, and G. G. Richard, "Pool tag quick scanning for windows memory analysis," *Digital Investigation*, vol. 16, pp. S25-S32, 2016.

[18] B. Taubmann, C. Frädrich, D. Dusold, and H. P. Reiser, "TLSkex: harnessing virtual machine introspection for decrypting TLS communication," *Digital Investigation*, vol. 16, pp. S114-S123, 2016.

[19] "Volatility framework maintained by volatility foundation," https://github.com/volatilityfoundation/volatility.

[20] W. K. Sze and R. Sekar, "Provenance-based integrity protection for windows," in *Proceedings of the 31st Annual Computer Security Applications Conference, (ACSAC '15)*, pp. 211–220, usa, December 2015.

[21] A. Luțaș, A. Coleșa, S. Lukács, and D. Luțaș, "U-HIPE: hypervisor-based protection of user-mode processes in Windows," *Journal of Computer Virology and Hacking Techniques*, vol. 12, no. 1, pp. 23–36, 2016.

[22] "The increased use of PowerShell in attacks," Symantec Corporation, 2016, https://www.symantec.com/content/dam/symantec/docs/security-center/white-papers/increased-use-of-powershell-in-attacks-16-en.pdf.

[23] IL. Kit/BlueHat Decks/MattGraeber. CaseySmith. pdf, "Device Guard Attack Surface, Bypasses, and Mitigations. BlueHat IL Jan, 2017," https://microsoftrnd.co.il/Press%20Kit/BlueHat%20IL%20Decks/MattGraeber.CaseySmith.pdf.

[24] "Fileless attacks against enterprise networks, 2017.2," https://securelist.com/blog/research/77403/fileless-attacks-against-enterprise-networks/.

[25] E. Mark, Russinovich, D. A. Solomon, and A. Ionescu, "Windows internals," *Microsoft Press*, pp. 423–429, March 25, 2012.

[26] Y. Jia, L. C. Zheng, H. Hu et al., "'The Web/Local' Boundary Is Fuzzy: A Security Study of Chrome's Process-based Sandboxing," in *Proceedings of the ACM Sigsac Conference on Computer and Communications Security (CCS '16)*, pp. 791–804, 2016.

[27] R. Babiarz, "Chakra Jit Cfg Bypass," 2016, http://theori.io/research/chakra-jit-cfg-bypass.

[28] C. Hargreaves and H. Chivers, "Recovery of encryption keys from memory using a linear scan," in *Proceedings of the 3rd International Conference on Availability, Security, and Reliability, ARES 2008*, pp. 1369–1376, esp, March 2008.

[29] I. Haller, A. Slowinska, and H. Bos, "MemPick: high-level data structure detection in C/C++ binaries," in *Proceedings of the 20th Working Conference on Reverse Engineering, (WCRE '13)*, pp. 32–41, deu, October 2013.

[30] B. Saltaformaggio, Z. Gu, X. Zhang, and D. Xu, "Dscrete: automatic rendering of forensic information from memory images via application logic reuse," in *Proceedings of the 23rd USENIX Security Symposium*, 2014.

[31] C. Neasbitt, B. Li, R. Perdisci, L. Lu, K. Singh, and K. Li, "WebCapsule: towards a lightweight forensic engine for web browsers," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, (CCS '15)*, pp. 133–145, usa, October 2015.

[32] X. Xie, B. Lu, D. Gong, X. Luo, and F. Liu, "Random table and hash coding-based binary code obfuscation against stack trace analysis," *IET Information Security*, vol. 10, no. 1, pp. 18–27, 2016.

[33] Y. Chen, S. Reymondjohnson, Z. Sun, and L. Lu, "Shreds: fine-grained execution units with private memory," in *Proceedings of the IEEE Symposium on Security and Privacy (SP '16)*, pp. 56–71, 2016.

[34] "Intel® Software Guard Extensions (Intel® SGX), A Researcher's Primer," https://www.nccgroup.trust/uk/about-us/newsroom-and-events/blogs/2015/january/intel-software-guard-extensions-sgx-a-researchers-primer.

[35] "CreateEnclave function," https://msdn.microsoft.com/en-us/library/windows/desktop/mt592866(v=vs.85).aspx.

[36] "Metasploit," https://github.com/rapid7/metasploit-framework.

[37] "Symantec/Norton AntiVirus—ASPack Remote Heap/Pool Memory Corruption, CVE-2016-2208," 2016, https://www.exploit-db.com/exploits/39835/.

[38] "Pwn2Own 2010 Windows 7 Internet Explorer 8 exploit," https://www.exploit-db.com/docs/11921.pdf.

[39] M. I. Cohen, "Characterization of the windows kernel version variability for accurate memory analysis," *Digital Investigation*, vol. 12, no. 1, pp. S38-S49, 2015.

[40] "Parameter Passing," https://msdn.microsoft.com/en-us/library/zthk2dkh.aspx.

[41] J. D. Valois, "Implementing lock-free queues," in *Proceedings of the International Conference on Parallel Distributed Computing Systems*, pp. 64–69, 1995.

[42] "ChakraCore," https://github.com/Microsoft/ChakraCore.