Journal of Computer Science

Original Research Paper

On Improving Antivirus Scanning Engines: Memory On-Access Scanner

Mohammed I. Al-Saleh and Rasha K. Al-Huthaifi

Department of Computer Science, Jordan University of Science and Technology, Jordan

Article history
Received: 02-06-2017
Revised: 15-07-2017
Accepted: 25-07-2017

Corresponding Author:
Mohammed I. Al-Saleh
Department of Computer Science, Jordan University of Science and Technology, Jordan
Email: misaleh@just.edu.jo

Abstract: The Antivirus (AV) products are utilized by home user's community to attain protection. To some extent, the AV meets users' expectations by detecting previously known malware samples. In this study, we question the set of events which should trigger the AV to scan data. Scanning every single piece of data as it moves from one location into another could be a demanding and performance-killing task. The AV faces a design challenge when deciding what kind of data to scan and when to do so. Typically, the on-access scanner component of the AV scans data upon moving from/to hard drives. Other occurrences of data movements are of equal importance. For example, data moves between different memory locations or between memory and network. In this study, we are motivated to explore what it needs to be done by the AV upon various data movements. We design and implement a system that has a capability of scanning memory when necessary. We recognize and intercept the most effective API calls that involve memory. Afterwards, we extract involved data and scan it if it has not been scanned before. We test our system against 15 real malware and find out that our system is capable of detecting all malware samples. Furthermore, we provide a thorough performance study to present the overhead of our system.

Keywords: Antivirus, On-Access Scanner, Malware, Memory Scanner

Introduction

Computer security is increasingly given attention all levels. Compared with other security tools, the Antivirus (AV) software proves useful and stands the test of time. Security products are usually evaluated against two main metrics: Detection effectiveness and performance overhead. The AV, in turn, tries to reach a good balance between these two factors. To be effective, the AV maintains a database of virus signatures against which it scans data. Not only that having a unique signature for a virus enables the AV to catch the corresponding virus, but also it\makes false positive rates almost negligible. In addition, even though the AV optimizes its scanning engine for performance, unfortunately it skips doing some very important actions for the sake of maintaining the bottom-line of acceptable performance overhead. For example, the AV seems to choose not to scan Only-In-Memory Data. This is the kind of data that will never be written to a secondary storage such as Hard Drives (HD) (Al-Saleh *et al*., 2015; Al-Saleh and Shebaro, 2016; Gionta *et al*., 2014). Interestingly, according to a study (Gupta *et al*., 2010), popular AVs do not scan data received through network into memory. Furthermore, data that is sent from memory through network is not scanned by the AV. Apparently, the AV is triggered upon reading from or writing to files. This kind of AV is called on access scanner, where the AV scans data upon file access. However, this is not always the case. For example, an attacker might assemble a virus from malicious data obtained from several files, network data, other local processes, or from its own runtime data. What is common to all such cases is that malicious data is only assembled in memory from different sources and that it is not coming from or going to a single file. This research aims at finding an efficient way to address this problem by developing what we call Memory-On-Access Scanner (MOAS). We design a system that basically tracks data where-ever a memory data is involved. Tracking data in memory at the byte-level

could not be efficient nor effective. Consequently, we track memory at the API-level, where we have a better view of data at much fewer intrusiveness. We design and implement a system that has a capability of scanning memory when necessary. In our system, we consider scanning data at three different situations that represent data movements: Memory-to-memory, memory-to-network and network-to-memory. We recognize and intercept the most effective API calls that involve memory data. Afterwards, we extract the involved data and scan it if it has not been scanned before. We do that by keeping a database of hash values for already-scanned data. Performance can be further enhanced by developing a better interception methodology, choosing the right APIs to intercept, utilizing a high-performance scanning engine and whitening benign programs.

This paper is organized as follows. In section 2, we give a brief background on different topics we use through this paper. Section 3 illustrates the threat model of the system which we try to address. In section 4 we overview the architecture of our system. Our experiments are detailed in section 5. Our results are shown in section 6. Then, discussion and future work are covered in section 7. This is followed by related work and the conclusion.

Background

This section gives a brief background about the following:

- Malware types and infection techniques
- AVs and detection techniques
- ClamAV, a popular open-source AV that we use in this study to scan data
- WinAppDbg, a general API interception frame-work that we use in this study

This section is not meant to be complete in any way. It just explains need-to-know things for the reader of this paper to help simplify its contents.

Malware

Malware is any software that is developed with a harmful intent. Malware can be classified and named based on the infection technique, spreading technique, or kind of harm it causes. For example, a Worm is a malware that mainly exploits a security vulnerability of a awed program over the network. Then, it propagates itself by finding new victims recursively. A Virus is a malware that attaches itself into file such as EXE and DLL files. Its code stays inactive until the corresponding programs are executed. A Trojan horse is a malware that appears to be useful such as a downloadable game or a well-known utility program, but it hides malicious code inside, which can do any kind of harm. Spyware is a malware that steals information from a system and sends it to a remote attacker. There are obviously many types of information that can be interesting to the attacker such as credit card numbers, contacts, documents and emails. Rootkit is a malware modifies the kernel and hides itself inside. Being inside the kernel gives all privileges and capabilities to the rootkit to potentially do anything possible. A Virus infects files using many techniques. For example, Boot viruses (Rad *et al*., 2011) can infect any machine regardless of what OS is installed by exploiting the boot process of the computers.

A memory resident malware stays in memory all time. It can be of different types (Szor, 2015): Direct-action, temporary and swapping. Direct-action viruses do not make themselves visible in memory. They load into memory along with the host program and then they seek for new hosts programs to infect. These viruses may infect many files upon execution. The virus of this type can take control by allocating a block of memory, relocates its code in the allocated block and activates itself. It basically hooks the flow of execution, which makes it able to infect new files and systems. Temporary memory-resident viruses do not always reside in memory, but they stay in memory for a short period of time and wait for a particular event to occur. Swapping memory-resident viruses rely on loading a small part of the virus code into memory which enables them to stay active all time. However, this part of the code may be a hook event, so whenever the hook event is triggered, the virus gets a segment of its code from the disk and then infects a new file and finally clears the loaded segment from memory.

Antivirus

The Antivirus (AV) is the last line of defense for a computer system. It is one of the most widely used tools for malware detection. The AV scans data by comparing it against its database of known virus signatures. If a match is found, then the AV blocks the operation and takes appropriate actions to handle the situation (Al-Saleh, 2015).

The AV utilizes different detection technique. These techniques can be classified into many types (Rad *et al*., 2011; Szor, 2015): First-generation scanners, second-generation scanners and algorithmic scanning. First-generation scanners utilize techniques such as string matching, wildcards matching, hashing and fixed-point scanning. Second-generation scanners are more trustable and use smarter ways of detection, such as heuristic scanning. Algorithmic scanning methods (more accurately called virus-specific detection) is not a general technique, but rather it is a method that is created for detecting a specific malware. Code emulation

technique could be utilized in the algorithmic scanning. This is done by simulating the CPU, memory, storage resources and some of the important functions of the OS by a virtual machine. It runs malware in the virtual machine and then investigates its behavior in a controlled environment.

ClamAV Antivirus

ClamAV (Kojm, 2004) is the most popular, open-source AV that is typically utilized by security researchers and practitioners. ClamAV has a very useful feature that enables scanning files from the command line and it can be integrated within a development framework (as we do in this study). An application can use ClamAV scanning capabilities by dynamically linking to it. In this case, ClamAV daemon along with its database are loaded once and then shared with the user agents, so that agents can scan files by calling various scanning functions. ClamAV functions can be called to mainly scan a file, directory, or buffer. Scanning results can be stored and interpreted easily.

WinAppDbg Debugger

WinAppDbg (Vilas) debugger is a python module that is used for general debugging purposes. It can be interfaced with python programs to manipulate processes, threads and libraries. It has many functions such as attaching a script as a debugger, tracing the execution of a script, setting breakpoints and hooking windows API calls. It is easier to modify and maintain than any other windows debuggers, because it does not have any native code.

Threat Model

Figure 1 illustrates the three directions of data while moving. Data flows between the following zones:

- Memory and secondary storage
- Memory and network
- Memory and memory

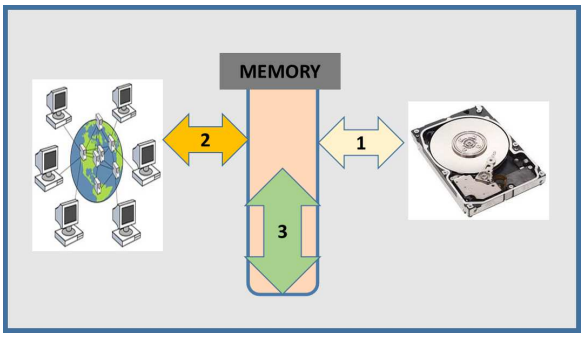


Fig. 1. Threat model

Identifying these zones is essential for security assessment. Apparently, AVs scan data when transferred in the first zone only. This paper complements the AV's job by paying attention to the other zones and scanning the involved data.

System Architecture

This paper aims at developing an effective and efficient memory scanning AV component. Specifically, our goals can be concluded by the following:

- Adding a security component to computing systems to scan memory when necessary
- Our security component should be effective in catching malware. Consequently, testing against real malware instances is necessary
- The performance impact of our system should be minimal. Performance study and enhancement strategy is provided
- Based on the coming results, recommendations to the AV and Operating Systems vendors will be provided

Even though several approaches to develop a MOAS can be suggested, our approach to the problem is through intercepting Windows API calls. This proposed approach is justifiable. Upon data transfer, a process's memory can be a source or a destination. In any case, an API call is needed to transfer data. Consequently, intercepting API calls to scan the involved data is necessary to prevent transferring malicious contents. Scanning data at the API call is efficient because scanning every single memory read or write obviously kills the system performance. However, scanning at the API granularity is more efficient because higher level of data is being scanned. In other words, triggering MOAS happens less frequently (performance gain) and more effectively (mature data is scanned) in case of intercepting API calls compared with scanning every single memory operation (read/write).

Several frameworks are available to intercept API calls. We use WinAppDbg debugger (see section 2.4) to intercept API calls. It can be inter-faced with Python programs to manipulate processes, threads and libraries. It provides wrappers around API calls in a form of PRE (happens before) and POST (happens after) kind of controls. We utilize these capabilities to hook memory-related API calls. Our system utilizes PRE (happens before) and POST (happens after) controls provided by WinAppDbg to decide whether scanning data is necessary or not. For example, when a monitored process wants to write into memory, it uses an API call (e.g., WriteProcessMemory()). This API call will be intercepted and our PRE control is triggered. In the PRE control, we can collect all information about the write operation, such as the involved range of addresses. Then,

we can extract such data for inspection before the operation is carried out and causes any harm. If the involved data is clean, the operation can pass as usual. Figure 2 explains the basic flow of our system.

There are many types of APIs that can be intercepted by using WinAppDbg. But we are interested in two API categories: Memory-related and network-related. Memory-related APIs can also be classified into three types:

- General Memory APIs: Used to copy data from a memory location into another as a result of (ultimately) calling RtlMoveMemory()
- Virtual Memory APIs: Used in managing virtual memory (allocation/de-allocation). Newly allocated memory is necessarily benign because it has no contents yet. However, we keep track of allocated ranges to be used in scanning data later on when part of such ranges are accessed. In addition, we focus on memory de-allocation APIs. Basically these are: VirtualFree(), VirtualFreeEx()
- Debugging APIs: Used in reading/writing from a process's memory. Both ReadProcessMemory() and WriteProcessMemory() are used to read/write data from/to another process's memory

Network-related APIs are included in the WinSock library. There are many WinSock APIs, but we focus on the APIs that are related to sending and receiving data. These APIs are send(), sendTo(), recv() and recvFrom().

We intercept all these APIs in our PRE or POST mechanism. However, sometimes, it is necessary to extract and scan data before (i.e., in the PRE control) it moves to another location and sometimes it is necessary to extract and scan data after (i.e., in the POST control) it comes from another location. We extract and scan data in the PRE control for the following APIs: RtlMoveMemory(), VirtualFree(), VirtualFreeEx(), WriteProcessMemory(), send() and sendTo(). We extract and scan data in the POST control for the following APIs: Read-ProcessMemory(), recv() and recvFrom().

To be able to scan data online, we built our own MOAS based on ClamAV. Whenever our system determines that certain data needs to be scanned, ClamAV is called to scan data. We run ClamAV as a background service so that we can communicate with it on demand.

To enhance the performance of our system, some optimizations are needed. For example, already- scanned data (if not changed) should not be scanned again should it be involved in future operations. Doing so saves a lot of unneeded memory scanning. In order to account for already-scanned data, our Memory Database Manager (MDBM) component maintains a database through which it checks whether the data has already been scanned or not. The database contains memory range of addresses and a hashed value of the data to indicate if data has been changed since the last scan. Initially, when a new memory is allocated, the corresponding memory ranges are registered in the database during the PRE control of the memory allocation API.

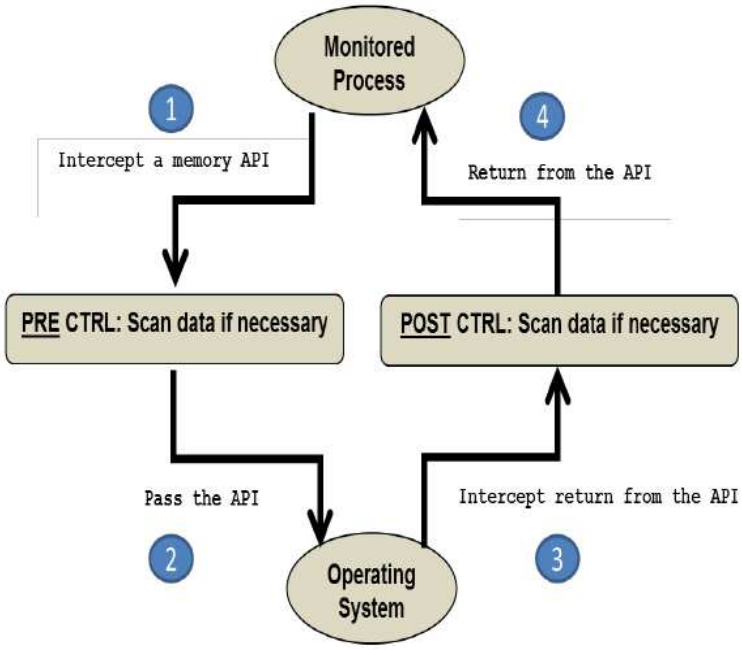


Fig. 2. The basic flow of our protection system

When data is to be transferred, its database entry will be checked whether it is scanned or not. The hash of data will be computed and compared to the database hash value. If a match is found, then there is no need to scan the data again. A database entry is removed upon memory deal location. It is worth mentioning that it is not necessary to monitor every single process in the system. Some processes might be white-listed to avoid performance degradation.

Experiments

In this section, we design the following experiments to handle these goals set in the previous section. Basically, we want to check the effectiveness and performance impact of our solution on the system.

Experiment I: Testing Effectiveness

To test the effectiveness of our scanner, we test our system against 15 different malware types in all APIs which we are interested in. Table 1 summarizes the APIs that are tested. In order to do that, we created nine programs, each of which has a call to one of the nine API routines. An API call involves copying a malware instance from/to memory. We collected these popular malware samples from Internet repositories that are available to the public. We used the same naming convention of these mal-ware samples as ClamAV. We test all 15 malware instances on every API call to check the scanner's capability to detect malware. We run each program under the control of our system.

Experiment II: Measuring Performance Overhead

In this experiment, we want to expose the performance impact of our system. Our system works by intercepting specific APIs and then scanning the data involved in these APIs. Consequently, in order to measure the performance impact, we deal with two different scenarios:

- Scenario 1: Performance impact of both intercepting APIs and scanning data in the APIs
- Scenario 2: Performance impact of only intercepting APIs (i.e., without scanning)

For Scenario 1, we design 9 programs. Each program utilizes one of the 9 memory-related APIs. We vary data sizes to be used in each API to be 1 KB, 10 KB, 100 KB, 1 MB and 10 MB. This is to show the curve of overhead when the involved data becomes bigger. Our system intercepts an API in the PRE control, records a timestamp (t1), extracts and scans data when necessary and passes the API through. After the operating system completes the API, our system gets the control again in the POST control, extracts and scans data when necessary, takes a timestamp (t2), computes the elapsed time (difference between t1 and t2) and passes the API through. Figure 3 illustrates this process.

Sometimes, the same data can be read or written several times. In this case, there will be no need to scan the same data over and over again. In order to avoid unneeded scanning, our system maintains a database of Message Digests hashes (using MD5 algorithm) of the already-scanned data. Prior to scan data, an MD5 hash of the data is computed and looked up in the database of hashes. In case of a match, scanning will not be triggered. To evaluate this performance enhancement, we design our experiment to call the same API twice in a row such that in the two calls the same data will be involved. We want to measure how much performance over-head our system could avoid in case of facing the same data again.

One important aspect of measuring the performance of our system is the AV engine that is used to scan data. In this study, we use ClamAV because it can be integrated in our test programs easily. However, in terms of performance, ClamAV might not be the best choice because other commercial AVs are optimized for performance and dedicated to fulfill users' expectations. To pinpoint this issue, we design Scenario 2 to completely disable scanning while intercepting an API call. Here, we only want to measure the performance overhead of just intercepting the APIs and taking control of running programs. This way we create a performance base-line for our intercepting methodology and isolate the scanning time because it directly depends on the AV scanning engine. We still compute elapsed time as in Scenario 1, but without scanning data. Because we don’t scan data, there is no point of varying data sizes in Scenario 2. We also use the same 9 programs we design for Scenario 1.

Table 1. Memory-related APIs and their description according to Microsoft Software Developer Network (MSDN)

API	Description
MoveMemory()	Used to transfer data from one memory into another.
VirtualFree()	Used to free a "region of pages" in virtual address space of a process.
VirtualFreeEX()	Used to free a "region of memory" in virtual address space of a process.
WriteProcessMemory()	Used to write data into another processs memory.
ReadProcessMemory()	Used to read data from another processs memory.
Send()	Used to send data through network sockets.
Recv()	Used to receive data through network sockets.
SendTo()	Used to sends data to a specific target.
RecvFrom()	Used to receive data from a specific source.

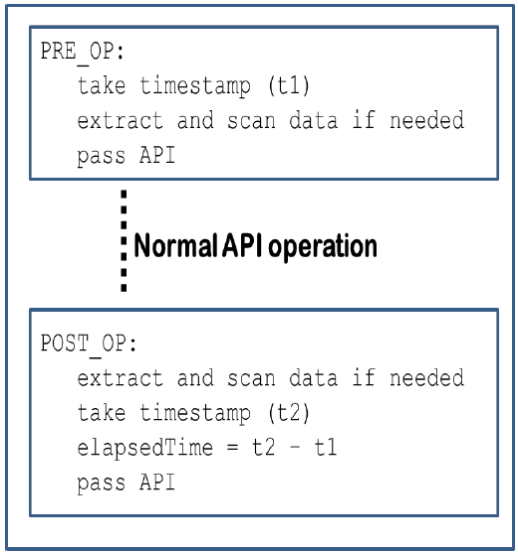


Fig. 3. Elapsed time computation

Experimental Setup

Here are more information about machine specification and how we conduct our experiments:

- Machine specifications:
  - OS: Microsoft Windows 7 Enterprise edition with Service Pack 1
  - RAM: 8 GB
  - Hard Drive: 500 GB
  - CPU: Core i7-860 at 2.80 GHZ
- Scenario 1: For each API (of the nine), we developed a program that utilizes that API while varying data sizes: 1 KB, 10 KB, 100 KB, 1 MB and 10 MB. The execution of each program is repeated 10 times for each data size and the average elapsed time is computed for the 10 runs. A machine restart is conducted right after each run to ensure a fresh starting point
- Scenario 2: We use the same setup of Scenario 1 except that we do not vary data sizes and we do not scan data

Results

In this Section, we present our results for the experiments which are explained in the previous section. Our aim is to check how effective our system is and how much performance overhead it imposes on programs.

Results for Experiment I

The purpose of this experiment is to check the effectiveness of our system in malware detection when a memory API is involved. We tested each memory API against 15 different malware samples. For example, ReadProcessMemory() API was used to read all the 15 malware samples. Similarly, WriteProcessMemory() was used in the same manner. The same applies for the remaining APIs. Table 2 shows the overall results of this experiment. It is obvious that our system was able to detect all of the mal-ware samples inside all APIs. This indicates that our system is pretty effective in malware detection.

Results for Experiment II

This experiment measures the performance impact of our system. As explained in the previous section, this experiment has two scenarios. So, we present their results separately.

Scenario 1 results

In this scenario, we want to show how much over-head our system imposes on the API calls. This includes the overhead of intercepting an API and scanning data that is involved in that API. The interception depends directly on the methods of interception (WinAppDbg in our case). After interception, we utilized ClamAV to scan the involved data. Apparently, our system depends directly on these two frameworks. Enhancing them would has a direct improvement on the overall performance of our system. Furthermore, a piece of data might be transferred several times in the system. In that case, we should not re-scan the same data again. Keeping hash values of scanned data could save us time by looking it up before scanning it. To expose this fact, in each run, we call an API twice on the same data. Our goal is to check how the performance of the second call (call2) could be different from that of the first call (call1).

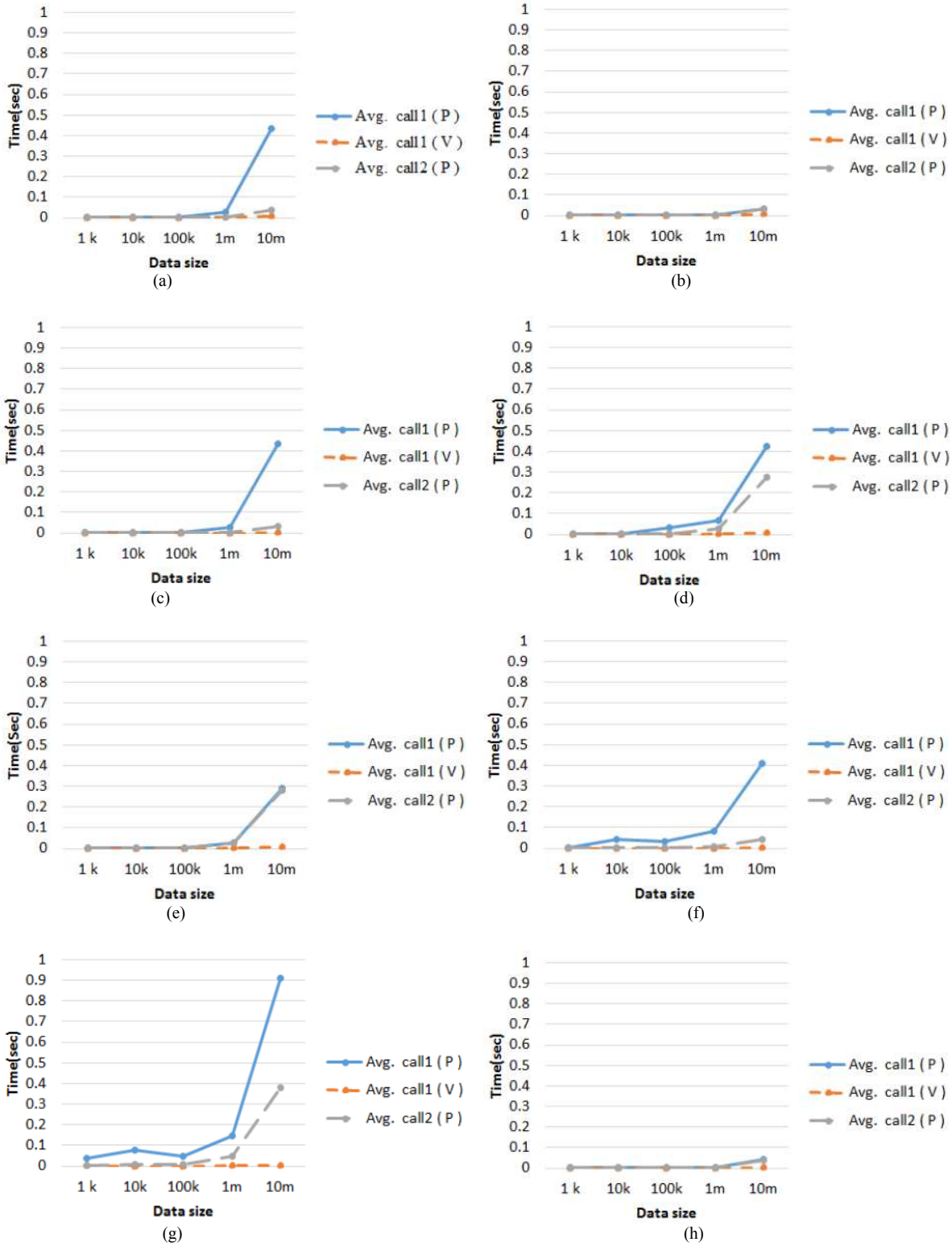
Figure 4 shows the elapsed times of each API separately. We have "Avg. call1 (P)" and "Avg. call2 (P)" when our system is applied (P in the figures is for Protected). Also, we have ”Avg. call1(V)” when our system is not applied (V in the figures is for Vanilla). The average is taken over 10 different runs where a machine restart is conducted between runs. We also vary the data sizes to expose overhead increase over data growth. In all the figures, the overhead imposed by call1 is obvious. It is less than half a second except for the network-related APIs, which are comprise non-deterministic factors that affect delay while sending or receiving data. It is also obvious from the figures that hash lookups improves call2 compared with call1.

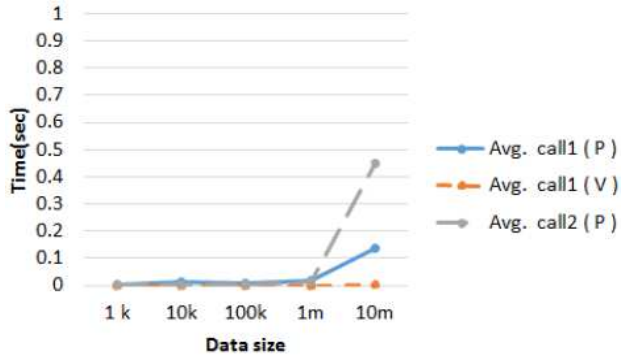
Scenario 2 Results

In this scenario, we want to show how much over-head our system incurs on the API calls by only intercepting API and without scanning data. This establishes a performance baseline and shows a room for scanning enhancement. Trying AVs other than

DOI: 10.3844/jcssp.2017.290.300

ClamAV could substantially improve performance. To get accurate results for the elapsed time of an API, we call it 1000 times in a raw and compute the elapsed time. Then, we divide the elapsed time over 1000 to report an average elapsed time over each API. Table 3 shows the elapsed times for all API calls. Table 3 Average times (in seconds) and over-head of calling APIs as in Scenario 2.





(i)

Fig. 4. Average elapsed times for all APIs; (a) MoveMemory() case; (b) ReadProcessMemory() case; (c) WriteProcessMemory() case; (d) VirtualFree() case; (e) VirtualFreeEx() case; (f) Send() case; (g) Recv() case; (h) SendTo() case; (i) RecvFrom() case

Table 2. Results for Experiment I

Malware name	Malware size (in bytes)	Detected in all APIs?
Win.Torgan.Zeus-2	676	yes
Win.Worm.Blaster-1	5,408	yes
Win.Worm.CodeRed-1	2,196	yes
Win.Worm.kido-22	158,843	yes
Win.Trojan.FlashBack-19	171,286	yes
Win.Worm.Stuxnet-9	15,198	yes
Win.Trojan.Dexter-1	3,967	yes
Win.Trojan.Zmist-2	13,923	yes
Win.Trojan.Agent.1395295	57,112	yes
Win.Trojan.Fari-481	49,094	yes
Win.Trojan.Clicker-1995	611,672	yes
Win.Worm.Koobface-22	11,624	yes
Win.Trojan.Bancos-8327	220,041	yes
Trojan.JS.StartPage.A	522	yes
Trojan.Downloader-88603	74,911	yes

Table 3. Results for experiment I

API name	Vanilla (no interception)	With interception
ReadProcessMemory()	0.000116234	0.000326422
WriteProcessMemory()	0.000117641	0.000331643
MoveMemory()	0.000138767	0.000324888
VirtualFree()	0.000194849	0.000428281
VirtualFreeEx()	0.000196746	0.000323634
Send()	0.001067821	0.000327566
Recv()	0.000963239	0.000325270
SendTo()	0.001068574	0.000325150
RecvFrom()	0.000885000	0.000329546

It is clear in the table that the imposed overhead is less than a millisecond per an API of interest. Given that this overhead is only imposed on the APIs of interest and not any other API, this can be considered minimal given the importance of scanning data inside these APIs. Network-related APIs show almost no difference because of the reason that calling such functions depends directly on the responsiveness of the system network stack, which is non-deterministic.

Study Recommendation

This study show that scanning only-in-memory data is necessary and doable. Scanning at the API level can be effective as most applications need to utilize the API provided by the Operating System (OS) to accomplish their tasks. Consequently, we recommend that the AV and OS vendors cooperate towards providing an efficient MOAS. The OS could support the AV by natively integrating a more

effective intercepting technique than using the general debugging technique in order to intercept API calls. The AV could register their own callback code with the AV to be integrated inside the API functions directly. Doing so could make the interception process much more reliable and efficient.

Discussion and Future Work

The effectiveness and performance of our system can be further enhanced:

Enhancing Effectiveness

Commercially-grade AVs can be utilized to scan memory data. The AV vendors could modify their scanning engines to specially deal with memory data. Furthermore, because our system works at the user-mode level, kernel memory is not being scanned. As a future work, we intend to develop a kernel module that is capable of tracing memory API calls at the kernel level.

Enhancing Performance

To enhance the performance and scalability of our system, some genuine programs that are tagged by the OS or by administrators should run without any intervention. Furthermore, centric database of scanned data should be created instead of having a separate database for each process. Finally, scanning data with the help of the Graphics Processing Units (GPUs) could greatly enhance the overall performance.

Related Work

Generally, the AV research has got a little attention from the security research community. Part of that is the fact that AVs are commercial, closed-source products. This has a negative impact on the general
security of systems. Having deep insights from security researchers and engineers could enrich these products and enhance their performance and security.

AVs have been researched from two perspectives: Security and performance. Each one has almost the equal importance with respect to end users. Consequently, getting into a balanced point between these two aspects is a vital goal for AV vendors. In the AV research, proposals are usually to enhance the scanning engine of AVs, to provide an efficient way to analyze malware, to attack the AV itself, or to measure the AVs overhead.

Research has been conducted to enhance the AVs scanning capability based on software solutions (Al-Saleh and Shebaro, 2016; Edwards *et al*., 2001; Edwards and Turner, 2010; Gassoway, 2013). Al-Saleh and Shebaro (2016) proposed an enhancement to the AV to make it scan memory-to-network and network-to-memory data. Their solution is based on an AV add-on feature that is capable of collecting TCP flows and scanning them incrementally upon flow changes. They showed that their method is effective and practical for scanning network data. Edwards *et al*. (2001) scanned the process's memory each time the process runs. Furthermore, the work in this study is an extension to the project we have started (Al-Huthai and Al-Saleh, 2017). Edwards and Turner (2010) produced an on-access scanner that delays writing to files until getting scanned. Gassoway (2013) developed a methodology to detect malware in the kernel memory (rootkits).

Several works deal with the problem of memory investigation for the purpose of malware analysis or disinfection (Sallam, 2013; Lengyel *et al*., 2014; Jiang *et al*., 2007; Gupta *et al*., 2010). Gionta *et al*. (2014) provides an efficient cloud service for virtual machines' memories. Szor and Ferrie (2009) analyze memory dumps looking for malware. Furthermore, studying malware behaviors by watching the API calls that called has been proposed (Ravi and Manoharan, 2012; Alazab *et al*., 2011; Grégio *et al*., 2012; Fujino *et al*., 2015; Ahmed *et al*., 2009). The difference between our work and all the above mentioned ones is that they either scan memory offline for malware analysis or they scan virtual machines' memories from hypervisors, which they are not always the case. Our solution provides real-time MOAS.

An attack vector on the AV software is conducted by bypassing the whole scanning process. Al-Saleh *et al*. (2015) showed that AVs can be bypassed through concurrent attacks. Rad *et al*. (2012) showed that polymorphism, metamorphism, encryption and obfuscation techniques can be utilized to bypass AVs. Ramilli *et al*. (2011) showed that the detection of Avs can be avoided by splitting it into parts that are distributed over several processes. Finally, performance studies on AVs to find their bottlenecks or improve scanning times have also been conducted (Vasiliadis and Ioannidis, 2010; Miretskiy *et al*., 2004; Lin *et al*., 2011; Al-Saleh *et al*., 2013).

Conclusion

In this study, we developed a framework(MOAS) which scans data that is sourced from or destined to a computer memory. Only-In-Memory Data can be dangerous if not scanned. We recognized and intercepted 9 APIs that involve memory data. The data involved in these APIs are extracted and scanned using ClamAV. Testing our system, we were capable of detecting 15 real malware samples when used with the APIs of interest. Furthermore, we examined the system performance and presented the overhead imposed by it. To even enhance performance, we maintained a database of data hashes to track already-scanned data so that we do not scan the same data again. In addition, we found out that replacing ClamAV with a more efficient AV will have substantial

performance improvement as our results suggested. Performance can be further enhanced by whitening benign programs so that there is no need to intercept or scan data when marked as benign. This paper could have a potential effect on systems security.

Author's Contributions

All authors equally contributed in this work.

Ethics

This paper is an extension to the work presented in the CSCEET2017 conference (Al-Huthai and Al-Saleh, 2017).

References

Ahmed, F., H. Hameed, M.Z. Shafiq and M. Farooq, 2009. Using spatiotemporal information in API calls with machine learning algorithms for malware detection. Proceedings of the 2nd ACM Workshop on Security and Artificial Intelligence, Nov. 09-09, ACM, Chicago, Illinois, USA, pp: 55-62. DOI: 10.1145/1654988.1655003

Al-Huthai, R.K. and M.I. Al-Saleh, 2017. Towards providing memory on-access scanners. Proceedings of the International Conference on Computer Science, Computer Engineering and Education Technologies, (CSCEET' 17).

Al-Saleh, M.I., 2015. Towards extending the antivirus capability to scan network traffic. Proceedings of the International Technology Management Conference, (TMC' 15), Antalya, Turkey, pp: 18-23.

Al-Saleh, M.I., F.M. AbuHjeela and Z.A. Al-Sharif, 2015. Investigating the detection capabilities of antiviruses under concurrent attacks. Int. J. Inform. Security, 14: 387-396. DOI: 10.1007/s10207-014-0261-x

Al-Saleh, M.I., A.M. Espinoza and J.R. Crandall, 2013. Antivirus performance characterisation: System-wide view. IET Inform. Security, 7: 126-133. DOI: 10.1049/iet-ifs.2012.0192

Al-Saleh, M.I. and B. Shebaro, 2016. Enhancing malware detection: Clients deserve more protection. Int. J. Electr. Security Digital Forens., 8: 1-16. DOI: 10.1504/IJESDF.2016.073728

Alazab, M., S. Venkatraman, P. Watters and M. Alazab, 2011. Zero-day malware detection based on supervised learning algorithms of API call signatures. Proceedings of the 9th Australasian Data Mining Conference, Dec. 1-2, Australian Computer Society, Inc., Australia, pp: 171-182.

Edwards, J. and S. Turner, 2010. Method and system for delayed write scanning for detecting computer malwares. US Patent 7,757,361.

Edwards, J., S. Turner and J. Spurlock, 2001. Method and system for detecting computer malwares by scan of process memory after process initialization. US Patent App. 10/014,874.

Fujino, A., J. Murakami and T. Mori, 2015. Discovering similar malware samples using API call topics. Proceedings of the 12th Annual IEEE Consumer Communications and Networking Conference, Jan. 9-12, IEEE Xplore Press, USA, pp: 140-147. DOI: 10.1109/CCNC.2015.7157960

Gassoway, P.A., 2013. Discovery of kernel root kits with memory scan. US Patent 8,572,371.

Gionta, J., A. Azab, W. Enck, P. Ning and X. Zhang, 2014. Seer: Practical memory virus scanning as a service. Proceedings of the 30th Annual Computer Security Applications Conference, Dec. 08-12, ACM, Louisiana, USA, pp: 186-195. DOI: 10.1145/2664243.2664271

Grégio, P.L. De Geus, C. Kruegel and G. Vigna, 2012. Tracking memory writes for malware classification and code reuse identification. Proceedings of the 9th International Conference on Detection of Intrusions and Malware and Vulnerability Assessment, Jul. 26-27, Springer, Greece, pp: 134-143. DOI: 10.1007/978-3-642-37300-8_8

Gupta, D., S. Lee, M. Vrable, S. Savage and A.C. Snoeren *et al*., 2010. Difference engine: Harnessing memory redundancy in virtual machines. Commun. ACM, 53: 85-93. DOI: 10.1145/1831407.1831429

Jiang, X., X. Wang and D. Xu, 2007. Stealthy malware detection through VMM-based "out-of-the-box" semantic view reconstruction. Proceedings of the 14th ACM Conference on Computer and Communications Security, Oct. 29-Nov. 02, ACM, Virginia, USA, pp: 128-138. DOI: 10.1145/1315245.1315262

Kojm, T., 2004. Clamav.

Lengyel, T.K., S. Maresca, B.D. Payne, G.D. Webster and S. Vogl *et al*., 2014. Scalability, fidelity and stealth in the drakvuf dynamic malware analysis system. Proceedings of the 30th Annual Computer Security Applications Conference, Dec. 08-12, ACM, Louisiana, USA, pp: 386-395. DOI: 10.1145/2664243.2664252

Lin, P.C., Y.D. Lin and Y.C. Lai, 2011. A hybrid algorithm of backward hashing and automaton tracking for virus scanning. IEEE Trans. Comput., 60: 594-601. DOI: 10.1109/TC.2010.95

Miretskiy, Y., A. Das, C.P. Wright and E. Zadok, 2004. Avfs: An on-access anti-virus file system. Proceedings of the 13th Conference on USENIX Security Symposium, Aug. 09-13, USENIX Association Berkeley, San Diego, CA., pp: 73-88.

Rad, B.B., M. Masrom and S. Ibrahim, 2011. Evolution of computer virus concealment and antivirus techniques: a short survey. arXiv preprint arXiv:1104.1070.

Rad, B.B., M. Masrom and S. Ibrahim, 2012. Camouage in malware: from encryption to metamorphism. Int. J. Comput. Sci. Netw. Security, 12: 74-83.

Ramilli, M., M. Bishop and S. Sun, 2011. Multiprocess malware. Proceedings of the 6th International Conference on Malicious and Unwanted Software, Oct. 18-19, IEEE Xplore Press, Fajardo, Puerto Rico, pp: 8-13.
DOI: 10.1109/MALWARE.2011.6112320

Ravi, C. and R. Manoharan, 2012. Malware detection using windows API sequence and machine learning. Int. J. Comput. Applic., 43: 12-16.
DOI: 10.5120/6194-8715

Sallam, A.S., 2013. System and method for proactive detection and repair of malware memory infection via a remote memory reputation system. US Patent 8,474,039.

Szor, P., 2015. The art of computer virus research and defense. Pearson Education.

Szor, P. and P. Ferrie, 2009. Detecting malicious software through process dump scanning. US Patent 7,568,233.

Vasiliadis, G. and S. Ioannidis, 2010. Gravity: A massively parallel antivirus engine. Proceedings of the 13th International Conference on Recent Advances in Intrusion Detection, Sept. 15-17, Springer, Ontario, Canada, pp: 79-96.

Vilas, M., Winappdbg. http://winappdbg.sourceforge.net