Black Penguin: On the Feasibility of Detecting Intrusion with Homogeneous Memory

Ning Zhang* Ruide Zhang* Qiben Yan† Wenjing Lou* Y. Thomas Hou* Danfeng Yao*
* Virginia Polytechnic Institute and State University, VA
† University of Nebraska-Lincoln, Lincoln, NE, USA
{ningzh, rdzhang, wjlou, thou, danfeng}@vt.edu yan@unl.edu

***Abstract*—Growing complexity in modern software is making signature-based intrusion detection an increasing challenge. Many recent intrusion detection systems rely on accurate recovery of application semantics from memory. In this paper, we approach the problem from a different angle. We observe that the user applications in corporate network often run in identical system environments due to standardized IT deployment procedure. The same applications share similar runtime statistics across different workstations through out the time, despite different uses by the end users. When an application is compromised on one workstation, its runtime profile would be different from the rest, similar to how a black penguin would look distinctly different from the rest of the colony.**

In this work, we present our preliminary study on Black Penguin, a compare-view based intrusion detection system leveraging homogeneity of application-level memory statistics in corporate environment. The detection system follows a three-step process that includes memory analysis, unsupervised learning and risk mitigation. To explore the feasibility of Black Penguin, we conduct two types of experiments using Internet Explorer and Firefox as target applications. First, we examine the statistical differences of the same application under different user usage. To this end, we collect and analyze memory statistics of browser when visiting the top 500 websites ranked by Moz. Second, we examine the difference when the application is under attack. Several browser attacks are used to generate the intrusion samples. Our preliminary evaluation demonstrates the feasibility of the approach. Lastly, we also provide discussions on the limitations of the proposed system as well as future directions.

I. Introduction

As information system becomes more and more integrated in our life, the complexity of the software system is also growing rapidly. Despite significant amount of research efforts in intrusion detection [1], [2], [3], [4] and software engineering [5], [6], software vulnerabilities still pose serious threats. As file-based malware scanning tools continue to become widely deployed, attackers are shifting their focus to memory-based techniques [7]. Many commercial virus scanning engines [8], [9] rely on malware signatures, and can be easily evaded via obfuscation and polymorphic techniques. Research on memory based malware scanning often relies on application level semantics [10], [7], [11], [12], [13]. However, recovering accurate application level semantic of memory remains a research problem, and it can often have significant performance impact.

In this paper, we propose to take a different approach to address the problem. Instead of tackling the challenge of semantic recovery to provide a high confidence detection, we try to answer the question: *Can we discover potential intrusion without knowing the fine grained application semantic?* To explore the feasibility of such approach, we present the preliminary study on Black Penguin (BPenguin), a host-based memory scanning framework that is capable of detecting malicious attack without application level semantic information. Memory snapshot of application contains rich program information. Abstractly, each program has a set of invariants that ensure the proper operation of the application. When the program is compromised, some invariants are changed. However, identifying these invariants for arbitrary programs remains an open research problem. Instead of tackling the challenge of invariant identification, we take advantage of the homogeneity in the corporate environment, where workstations have identical configurations on hardware, operating system and user software to ease the IT management tasks. When the application on one of the computers is compromised, the memory snapshot would appear differently compared to the others, similar to how a pure black penguin will stand out among all the black and white ones.

Building on this intuition, BPenguin applies unsupervised machine learning to discover intrusions. There are three steps in BPenguin. First, memory statistics of the target application are collected at the end host by BPenguin agents. The second step is classification using the unsupervised learning. In this step, the collected statistics are transmitted to a centralized intrusion analysis server. In the last step, results from step two are analyzed to provide recommendations for further investigation.

We study the feasibility of the system by asking two questions. The first question is what the variance of memory statistics is under different user usage. The second question is whether the memory statistics are significantly different when the application is under attack. To answer these questions, we collected memory statistics of popular browsers in both Linux and Windows on the top 500 websites listed by Moz [14]. Browser is chosen as the target application, because it is one of the most targeted applications [24]. We apply the proposed system to detect heap spray attacks [15], since it is now one of the most widely used techniques [12] with the recent improvement in defense such as address space layout randomization (ASLR) [16], data execution prevention (DEP) [17], and stack protection [18]. In a heap spray attack, the attacker first allocates a large number of objects in the

heap of the target application. The contents of the objects are then filled with attack code which often consists of a long sled with shell code at the end. While heap spray itself does not directly lead to exploitation, the technique is widely used to significantly simplify and improve attacks.

Our experimental results indicate that the memory statistics among different browsers on different websites are surprisingly similar, while substantial disparity is observed when the application is under attack. The accuracy of detecting heap spray reaches as high as 98%. Furthermore, it is even able to detect different heap spray attacks with high accuracy. We make the following contributions in this work:

- We propose Black Penguin, an unsupervised learning system to detect intrusions using a compare-view philosophy.
- We analyze the memory statistical property of several applications under different application contexts in both Windows and Linux, and show that it is possible to use unsupervised learning on application memory to discover the intrusion.

II. Threat Model and System Overview

BPenguin is an intrusion detection system that examines the temporal variance of the memory of applications. It is capable of detecting memory corruption attacks without low level semantic information, and the overall design of the BPenguin system is shown in Fig. 1. From the system perspective, BPenguin consists of three subsystems. The first subsystem is the collection agent, which is installed as system service at the end host, responsible for collecting memory statistics of the target application. The second subsystem is the analysis engine, which can be placed on a more powerful corporate-wide intrusion analysis engine. The third subsystem is responsible for analyzing the intrusion analysis report and providing further mitigation steps either automatically or via reporting mechanism.

We assume that the adversary can exploit vulnerabilities in the monitored application to execute arbitrary code. However, the adversary is unable to break out the application sandbox set forth by the operating system. Since BPenguin is designed to be a memory scanning tool, attacks that do not modify application memory are not the focus of this study. BPenguin relies upon the trustworthiness of the operating system to obtain an accurate measurement of the memory statistics of the monitored application. Lastly, we also assume most of the samples are good, in that either only a small number of workstations under a corporate network is under attack or a workstation is under attack for a short period.

III. BPenguin System Design

As shown in Fig. 1, the workflow of BPenguin contains three stages, *memory feature extraction*, *anomaly detection* and *recommendation*. Memory images and associated information such as mapping details are captured first. The BPenguin agent will then extract features from the memory system as the last step of stage one. The extracted features are then forwarded to the reasoning module for intrusion detection. Finally, the recommendation subsystem examines the result to decide the mitigation strategy.

A. Memory Feature Extraction

There are two steps in the memory feature extraction stage, memory collection and memory processing.

The first step is memory collection. In this step, the process memory map is obtained by the agent, because it indicates the current memory pages mapped by the application along with certain semantic information. Typical semantic information includes the executable that is currently memory mapped, the location of the static and heap, the page for system call kernel export, etc. Using the memory map of the process, the BPenguin agent then obtains the memory contents itself by accessing the process memory.

The second step, memory preprocessing, follows three parts, as shown in Fig. 2. The first part correlates the process memory map along with the process pages accessed. For some of the pages that are shared between multiple processes, it is often desirable to disregard them in feature extraction. Writable data page in display library is a good example, where the contents of the data might have little to do with the control. Furthermore, it will also likely bring in additional noise in the memory collection. This correlation process is application or operating system specific. When there is no prior information to incorporate, the correlation step can be a simple pass-through.

The second part examines the memory and extracts the distribution statistics. In BPenguin prototype, we collect the octet frequency and byte frequency inside the memory as the preliminary feature set to perform the machine learning. Depending on the application to be examined, this step should be carefully considered by examining the distribution of the sample population. The octet frequency is captured with a vector of eight dimensions, each recording the number of appearance of an octet value, and the byte frequency feature is a vector of 256 dimensions. Even with just octet frequency and byte frequency, the dimension of the feature set is too large, and may have the curse of dimensionality problem[19]. Therefore we apply dimension reduction technique in the third step, feature extraction.

In the last part of step two, Principle Component Analysis (PCA) is used to reduce the dimension. More specifically, both the octet frequency feature vector and the byte frequency feature vector are analyzed and transformed into principle component matrix. The most significant N dimension is used.

B. Reasoning Module

Due to polymorphic nature of memory corruptions, it can be challenging to create adequate training set to train a classifier. Therefore, we elect to use unsupervised learning in this step to discover deviation from the normal profile. To be more specific, we use k-means to do the clustering. The data sample can be spatial-temporal in that application under different machine and different time is collected. The reasoning system can apply clustering on the application memory statistics

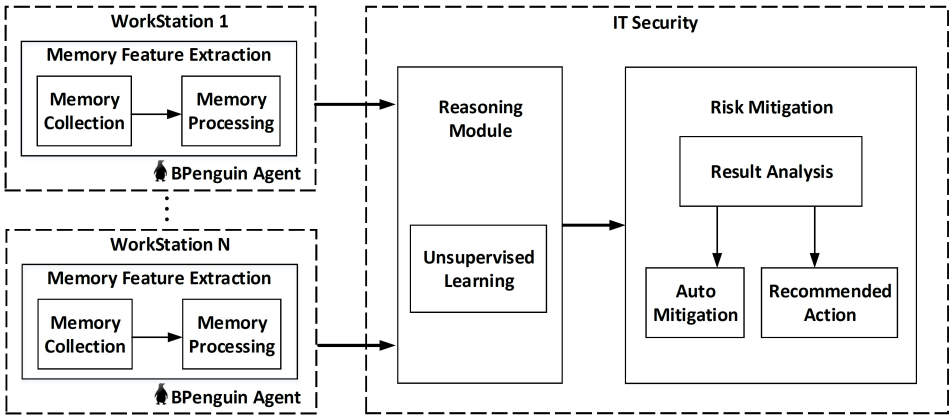

Fig. 1: Black Penguin Workflow Diagram

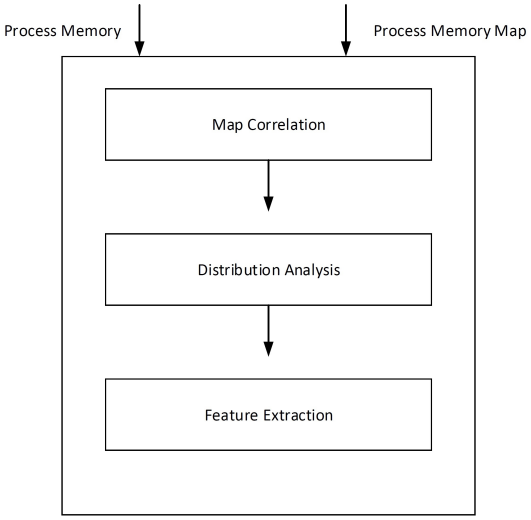

Fig. 2: Memory Processing Workflow

measurements over time on a single node or across all the nodes at a single time. When K is set to two, it is assumed that there is a sample of memory under attack in the population, which could be incorrect, and therefore we need the third step to mitigate the false positive. Note that there are many other more sophisticated machine learning techniques such as OC-SVM, X-Mean clustering that could be used to improve BPenguin, we plan to investigate these techniques as future work. In this paper, we focus on the feasibility aspect.

C. Recommendation System

Since we assume that memory snapshots are continuously monitored in BPenguin, and that all previous observations are normal, when there is a new attack, we should be able to see two clusters, one containing a single member with the memory of the application under attack. The second cluster should contain all the memory samples of a normal execution. Therefore, when the clustering result matches the distribution, we would raise an alarm. It should be noted that the current decision logic is binary and straightforward. In a more complex system, measurements from the clustering result should be correlated back to a policy engine. However, the detailed mapping from machine learning results to actionable items is an open problem [20].

IV. Evaluation

Applications in modern operating systems often have very different memory profiles in various settings. We would like to explore the feasibility of applying BPenguin to different applications under different conditions. We first explore the possibility of applying BPenguin in Linux using SSHD and Firefox as the monitored application. We then further explore its usage in Windows using Internet Explorer and Firefox as monitored applications.

A. BPenguin in Linux Environment

1) Secure Shell Hyperterminal Daemon as a Target: The first process we look at is ssh daemon process, which is considered as one of the simplest network programs in the Linux system. *sshd* is a program that is spawned from the original *sshd -D* in order to handle each individual incoming ssh connection [21].

Lack of spatial data on sshd process, we record the memory snapshot of an sshd process over 9 hour period, by dumping the memory contents into disk every 30 minutes, to collect the samples. Each memory dump is approximately 1.9MB in size, and it takes the bash script approximately 40 seconds to dump all the writable pages of the process. During 9 hours of recording, there are constant activities in the ssh connection for the first two hours by running a script that generates printout on the console, and completely idle except TCP keep-alive message for the next 7 hours. Due to the lack of readily available heap attack on the sshd process, we perform a synthetic attack on the sshd process using the gdb debugger. We spray the heap with 20 spray blocks consisting of a 1000 *0x9090* nop sled, along with a 23 byte shell code. This spray technique is very common in practice [22]. In this experiment, since the heap is directly observed from the memory map, we

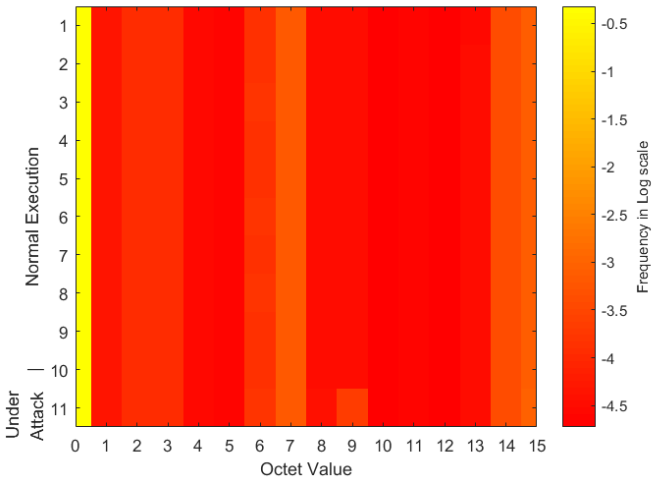

Fig. 3: SSHD Octet Heatmap

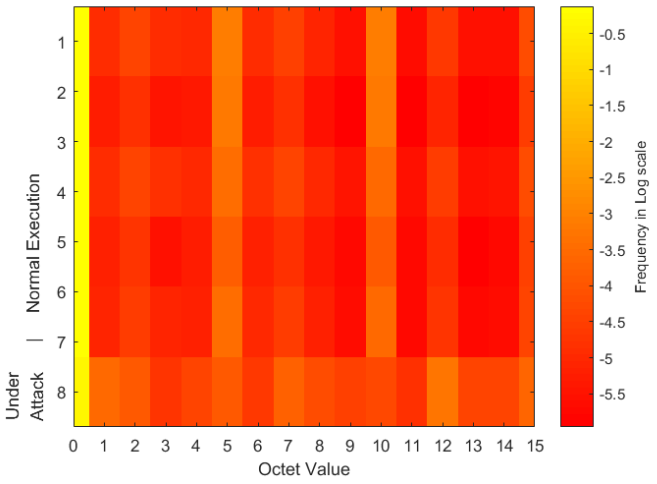

Fig. 5: Firefox Octet Heatmap

decide to only use the heap memory pages when generating the statistics in order to minimize the potential noise.

Fig. 3 show the heat map of the octet values in the memory dump. Due to the insertion of large amount of *0x9090* shell code, it can be observed that in the distribution graph that *0x9* and *0x0* have a distinctive spike compared to others for the intrusion sample at 11.

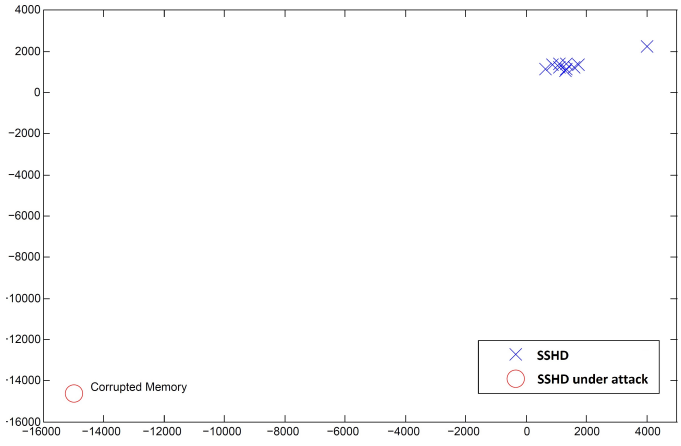

Fig. 4: SSHD Cluster

Lastly we cluster the observations in Fig. 4. K-mean clustering algorithm is used to generate the clustering, k is set to 2. The corrupted heap can be observed, distinctively as cluster of its own.

2) Firefox as a target: Besides the simple program, we are also interested in observing the memory statistics of a more complex program, such as a browser. Furthermore, since heap spray attack is very common in browser, this experiment can demonstrate the effectiveness of the detection mechanism towards real attacks with real usage environment. Firefox 37.02[23] is used for the experiment. There is no add-on in the browser. 7 memory snapshots are recorded in the experiment. We start the browser, and visit the following pages in order.

1) a html based hell world page
2) cnn.com
3) vt.edu
4) google.com
5) nytimes.com
6) washingpost.com
7) heap spray attack page on local server

The writable memory pages of the entire process are recorded after the page finishes loading, indicated by the spinning wheel on the browser. The size of memory dump varies from 700 MB to 1.3GB. The time it takes to dump varies from 3 minutes to 7 minutes to dump to disk in a virtual machine. For the last heap spray attack page, we use the exploit we found in exploit-db.com [24].

The octet distribution heatmap of the memory image is shown in Fig. 5. Sample number 8 is the intrusion sample. We can observe from the distribution graph that the noise is significantly higher than *sshd*. However, the effect of heap spraying is still quite easy to detect. Fig. 6 shows the clustering result for Firefox. Two distinctive cluster can be observed, however due to the contents of various website, it can also be observed that cluster label 0 can be easily divided into two clusters. This brings a unique challenge of how one can build a robust clustering method to accommodate such difference in browser memory footprints.

B. BPenguin in Windows Environment

To further investigate the application of BPenguin in different operating systems, we conduct a series of experiments on Windows. We choose modern browser as the target application, since it is one of the popular targets of malicious attacks in corporate environments.

1) Windows Samples Collection Process: Memory images of two browsers, Firefox [23] and Internet Explorer [25], are studied. For each target, we first collect the memory

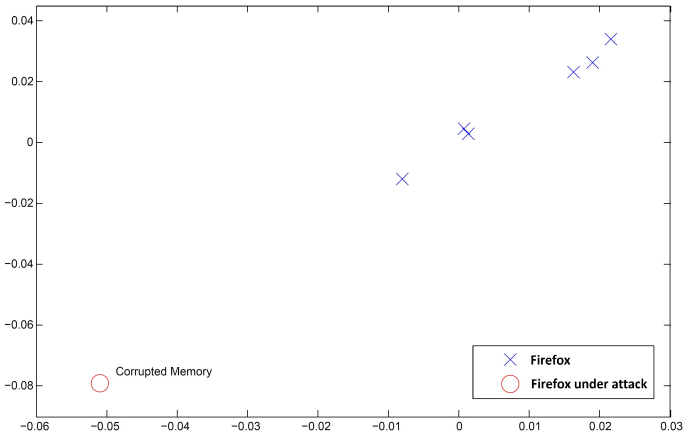

Fig. 6: Firefox Memory Cluster

statistics of the browser visiting top 500 Internet websites on Moz [14] using a powershell script, which we labeled as the benign samples. The intrusion samples are then collected by visiting a locally hosted malicious website immediately after the visit to the benign websites. Before collecting the memory images using scripts, we manually verify that the attack is successful. The effect of the heap spray technique used as part of the attack can be observed using the VMMap tool by Microsoft [26] as shown in Fig. 7. VMMap is a diagnostic application that shows the virtual memory structure of a running process. As shown in Fig. 7, a large amount of identical private memory objects appear in the heap as a result of this exploit. To collect the memory image for the 500 samples, a powershell script is written to automatically navigate the browser to the top 500 pages and dump the memory of the process using ProcDump [27]. Procdump is a process core dump tool provided by Microsoft as a system internal tool. Option -ma is used to dump the memory of the process. The result of procdump however is in minidump format [28], and contains raw memory as well as meta-data about the core dump. To extract the raw memory content, dumpChk is firstly used to find the raw memory offset and length in the core dump file, the memory content is then copied out for feature extraction.

2) Internet Explorer as a Target: To collect the intrusion samples of Internet Explorer, we use the proof of concept (PoC) exploit published in [29] to attack IE11 on Windows 7 x64 SP. The vulnerability is described in CVE-2015-2419 [30].

Fig. 8 shows the normalized octet frequencies (frequency of byte value 0x0 - 0xF appearing in memory) for 200 samples. The upper 100 samples are benign samples, and the lower 100 samples are applications under attack. As shown in Fig. 8, there are significant amount of octets for 0x0 and 0x7-0x8. This is because the memory is filled with the destination address of the shell code. We perform the experiment using 500 malicious sample and 50 intrusion samples. With unsupervised learning method k-means, we achieve 99.85% accuracy and 0% false negative rate. Fig. 9 shows the result of applying K-

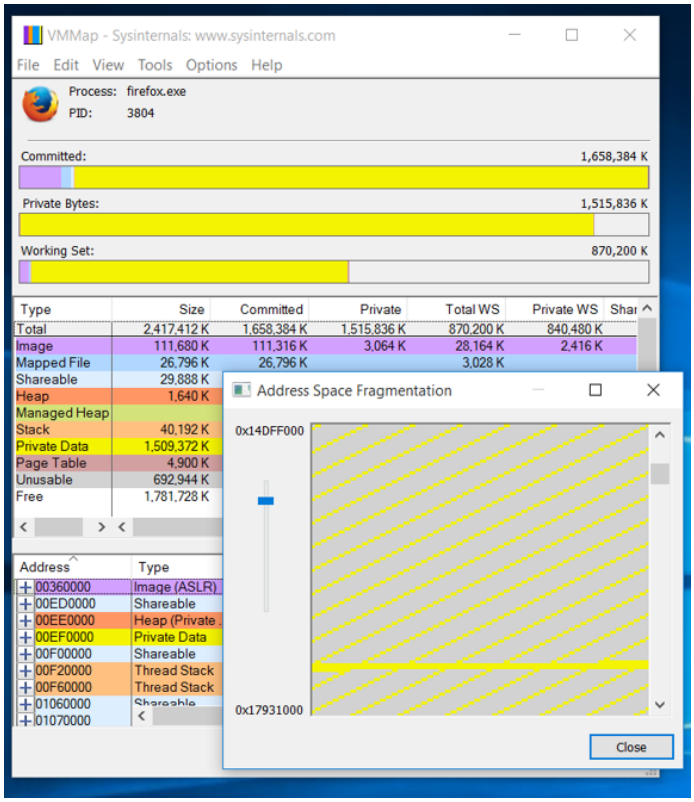

Fig. 7: Heap Spray VMMap for Firefox under attack

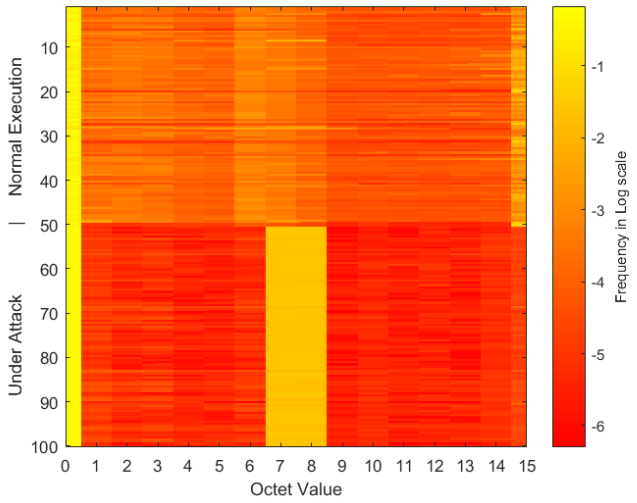

Fig. 8: IE11 Octet Heatmap

mean clustering to the Internet Explorer samples. We apply PCA to extract three features out of the initial 256 byte-frequency features to show the clustering results in Fig. 9.

3) Firefox as a Target: The PoC exploit we use against Firefox 50.0.1 can be found on exploitdb [31]. The vulnerabilities used in the exploit are CVE-2016-9079 and CVE-2017-5375 [22]. We are restricted to use Windows 7 in the Internet Explorer sample collection due to the availability of

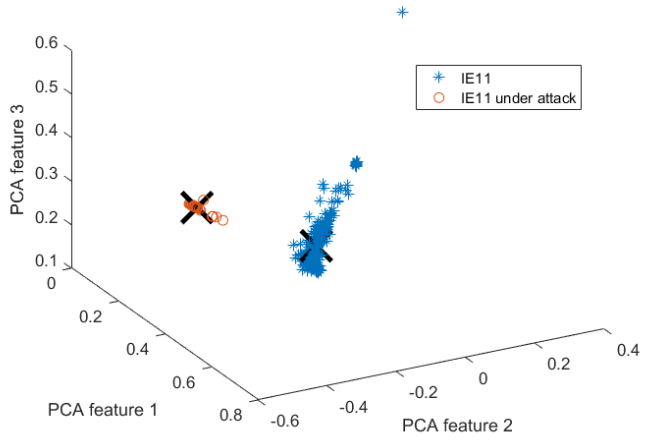

Fig. 9: IE11 Cluster

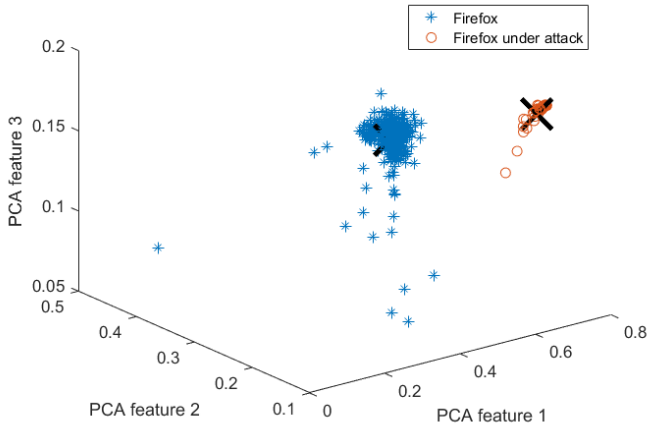

Fig. 11: FireFox Clustering

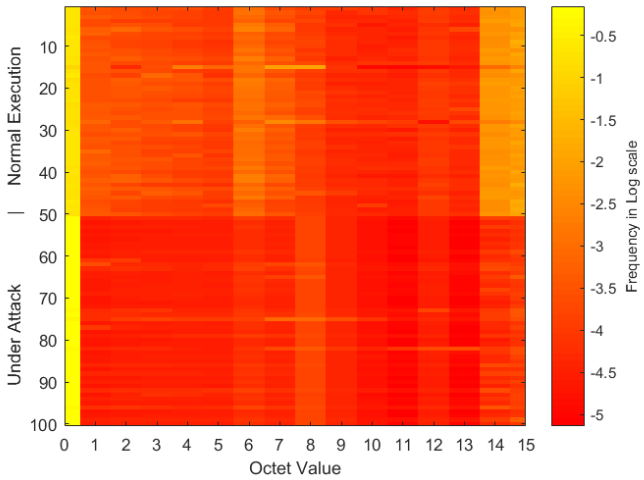

Fig. 10: Firefox(Windows 10) Octet Heatmap

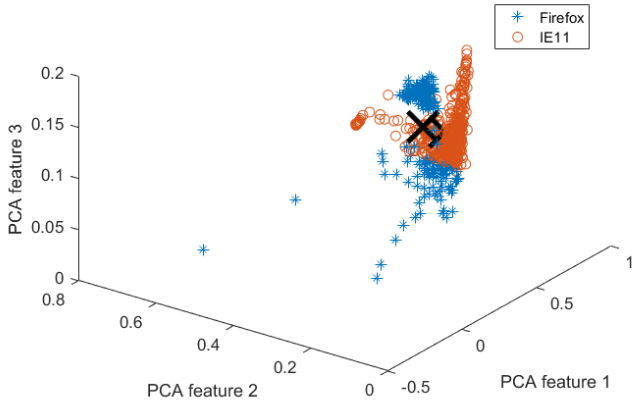

Fig. 12: The Clustering of Benign FireFox and IE Samples

the exploit. For Firefox, we collect the memory statistics in Windows 10, the latest Windows operating system. The octet frequency is depicted in a heatmap shown in Fig.10. It can be observed that 0x0 has significantly higher frequency than other values. Using the 500 normal execution samples and 50 intrusion samples, we evaluate BPenguin using unsupervised learning method k-means. We achieve 98% accuracy and 0% false negative rate. Fig. 11 shows the clustering result for the experiment.

4) Detecting Multiple Attacks: While the system is effective in detecting a single type of attacks that employ heap spray technique, we are also interested in evaluating how feasible it is to apply unsupervised learning across different attacks on different applications. Fig. 12 shows the clustering result of benign samples of IE and Firefox. It can be observed that the memory statistics of Firefox and Internet Explorer are not clearly distinguishable through K-means clustering. Fig. 13 shows the result of applying clustering to all of our samples, including both benign and intrusion samples of Internet Explorer and Firefox. From Fig. 13, we can see that it is still possible to distinguish normal applications from applications under attack. With K-means clustering, we achieve 97.17% accuracy and 2.22% false negative rate.

V. Discussion

BPenguin is originally inspired by the idea of detecting memory corruption attacks with only the memory snapshots of an application. The detection mechanism exploits the spatial-temporal homogeneity of an running application, it raises an alarm when the current memory profile deviates significantly from the past. By only comparing the memory profile, the system does not require the low level semantic information of the memory which is often difficult to obtain, and more difficult to keep up-to-date. However, BPenguin also has limitations due to its focus on memory homogeneity. In the rest

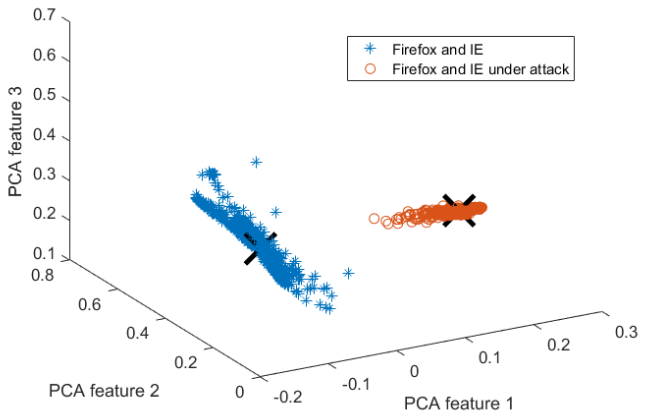

Fig. 13: The Clustering of Benign and Malicious FireFox and IE Samples

of the session, we will provide some discussions on potential pitfalls of BPenguin and the path forward.

A. Limited Detection

While incorporating machine learning in cyber attack and defense is very attractive, the machine learning method as well as the model we used to present the application state do pose limitations on the ability to detect attacks that leads to false positives and false negatives. There are cases where high level semantic information of the memory layout is not available. In the case of Firefox browser[23], unlike Chrominium[32] browser, all sub systems of the complex browser resides in a single process space. Therefore when the memory image is captured, the memory content becomes very large, and precise memory attacks that corrupt a small portion of the stack will most likely be statistically insignificant, and therefore is able to evade the detection of BPenguin.

Another problem with the current detection mechanism is that it has the underlying assumption that memory footprint under attacks are statistically significantly different from the footprint in normal execution, and this condition might not be necessarily correct. Since the initial establishment of the concept of mimicry attack[33], there has been a significant amount of interest in applying the same concept to different attacks [34], [35], [36]. Furthermore, similar to the polymorphic shell codes, there are several types of NOP sleds, including one-byte or multi-byte NOP equivalent instructions. Our current method relies on the n-gram statistics of the memory footprints. In the case where the adversary has the ability to perform adversarial learning, it is possible for the attacker to choose a polymorphic nop sled along with polymorphic shell code to closely mimic the statistical property of the memory of the application. Further experiments should be performed to investigate the resistance of the method against such powerful attackers. Emulation component should be added to the system if the detection is no longer effective under polymorphic shell code [37].

B. Utilization of High Level Semantic

Even though the goal of BPenguin is to perform intrusion detection without the need for low level semantic of the applications. Needless to say, semantic information on the memory can often be extremely helpful in reducing the false negative and false positive of the system. In the current BPenguin system, the system memory is captured using the process memory map in the *procfs*. And the system still utilized high level semantic provided by the operating system to reduce the amount of data it collects. The read/write flag was used to determine whether the memory should be included in the snapshot or not in BPenguin system, significantly improving the time it takes to generate a snap shot.

C. Time of Memory Collection

Time of collection can also be a problem with periodical scanning. In our experiment, we dump the memory before it exits. This strategy might not always work if the attacker anticipate such detection and cleans up after exploitation. It is also possible to perform memory check during large memory allocations, which is often accompanied with heap sprays. However, legitimate applications can also allocate large memory when it is executing resource intensive tasks such as web-based video games.

D. Conflicts with Memory Management System

While memory scanning techniques are very attractive in its ability to detect in-memory corruption attacks. It can however be quite expensive to run especially if the memory page has been stored in the non-volatile storage to free up memory space for other running process in the system. In order to scan the application memory, the operating system will have to load the process memory back from the disk, and in the case of Linux, the swap space. Once the memory scanning is complete, these recently loaded memory are then stored back in the disk due to its infrequent usage. On the other hand, for process memory that are used so infrequently to the point that is relinquished by the operating system to free up memory space for other process. The possibility of finding an exploitation in such area is slim.

E. Utilization of Spatial-Temporal Features

Besides studying how to overcome the limitation discussed above, it is also possible to study the spatial-temporal aspect of application memory. Information systems nowadays are increasingly homogeneous. More specifically, in today's corporate-centric and cloud computing driven environment, it is quite common to have a very small set of configurations across a large number of computing systems across the network. The configuration includes hardware configuration, operating system version, applications and the version of the applications. Furthermore, the day to day usage of a user often remains the same. Therefore, BPenguin can be extended to

capture not only the states across time in the same machine but the states across different machines and different users.

VI. Related Work

A. Intrusion Detection

Intrusion detection has been a very active field in the past decade. The detection mechanism can be generalized into three types: host-based [38], [4], [39], [38], network-based [40], [41] and hybrid-based approaches [2], [3]. The host-based approach [38] generally instruments the program execution to obtain more information. Network-based approach utilizes network traffic for program behavioral analysis [40], [41]. Lastly, the third type attempts to combine host based and network based approach, usually by combining the information from both approaches [2], [3].

B. Memory Analysis

Another method of detecting intrusion is through memory analysis of applications. This method also belongs to the family of host based intrusion detection. While the behavior based intrusion detection systems examines system call as a model of program behavior. The memory analysis approaches on the other hand examine the internal memory to uncover the program status with application semantic information. The semantic information can be obtained by binary analysis or source code examination. The construction of high level semantic information from low level details has been an active area of research [42], [43], [44], [45], [11], [7], [10], [46].

There have been many previous research works in the area of memory introspection [11], [10], [7], and BPenguin certainly falls in the category of memory introspection based approach. Jason et al. proposed SEER in [7], which is a malware scanning service framework that uniquely utilizes cloud environment to deduplicate memory scanning efforts. In [12], Ratanaworabhan et al. proposed to perform static analysis on the heap by treating bytes in the heap as code. Different from these approaches, BPenguin uses statistical method to automatically discover deviation from normal execution instead of matching memory to an existing attack pattern. Recognizing the performance overhead in Nozzle [12], Zozzle [13] was proposed to be a lightweight detection with an efficient emulation engine to counter javascript obfuscation. Closely related to BPenguin is the black sheep system proposed by Bianchi et. al. [11]. It uses compare view technique to detect kernel rootkit through crowdsourcing. However, they need to conduct extensive reverse engineering on the Windows operating system kernel to be able to make a semi-automatic comparison based on the semantic knowledge of the Windows kernel memory structure. The compare-view approach at the kernel level for detecting injected or tampered kernel data was also demonstrated in [47]. It deploys multiple checkpoints at the kernel data paths and utilizes lightweight cryptographic mechanisms to ensure kernel data integrity and consistency. However, BPenguin focus on intrusion detection at the application level.

VII. Conclusion

In this work, we propose BPenguin, a memory-based intrusion detection system. BPenguin utilizes unsupervised machine learning that compares memory statistics within a group with identical applications, similar to how a pure black penguin will stand out within a colony. It is capable of uncovering previously unknown attacks due to the compare-view nature. Furthermore, because only the statistics are used to perform anomaly detection, it does not require low level semantics of the applications. We use multiple applications in both Linux and Windows systems as targets for memory spray attacks to illustrate the effectiveness of BPenguin. However, the proposed system has its own limitations, and will be our future research focus.

Acknowledgments

This work was supported in part by NSF under Grants CNS-1446478, CNS-1405747, and CNS-1443889. Dr. Yan is supported by NSF under Grant CNS-1566388. The opinions expressed in this article are the authors own and do not reflect the view of the National Science Foundation or any agency of the U.S. government.

References

[1] M. Z. Rafique and J. Caballero, "Firma: Malware clustering and network signature generation with mixed network behaviors," in *Research in Attacks, Intrusions, and Defenses*. Springer, 2013, pp. 144–163.

[2] S. Shin, Z. Xu, and G. Gu, "Effort: A new host–network cooperated framework for efficient and effective bot malware detection," *Computer Networks*, vol. 57, no. 13, pp. 2628–2642, 2013.

[3] Y. Zeng, X. Hu, and K. G. Shin, "Detection of botnets using combined host-and network-level information," in *Dependable Systems and Networks (DSN), 2010 IEEE/IFIP International Conference on*. IEEE, 2010, pp. 291–300.

[4] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff, "A sense of self for unix processes," in *Security and Privacy, 1996. Proceedings., 1996 IEEE Symposium on*. IEEE, 1996, pp. 120–128.

[5] R. J. Anderson, *Security engineering: a guide to building dependable distributed systems*. John Wiley & Sons, 2010.

[6] L. D. Fosdick and L. J. Osterweil, "Data flow analysis in software reliability," *ACM Computing Surveys (CSUR)*, vol. 8, no. 3, pp. 305–330, 1976.

[7] J. Gionta, A. Azab, W. Enck, P. Ning, and X. Zhang, "Seer: practical memory virus scanning as a service," in *Proceedings of the 30th Annual Computer Security Applications Conference*. ACM, 2014, pp. 186–195.

[8] "Symantec anti-virus," http://www.symantec.com, accessed: 2015-04-30.

[9] "Mcafee anti-virus," http://home.mcafee.com, accessed: 2015-04-30.

[10] X. Jiang, X. Wang, and D. Xu, "Stealthy malware detection through vmm-based out-of-the-box semantic view reconstruction," in *Proceedings of the 14th ACM conference on Computer and communications security*. ACM, 2007, pp. 128–138.

[11] A. Bianchi, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Blacksheep: detecting compromised hosts in homogeneous crowds," in *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 2012, pp. 341–352.

[12] P. Ratanaworabhan, V. B. Livshits, and B. G. Zorn, "Nozzle: A defense against heap-spraying code injection attacks." in *USENIX Security Symposium*, 2009, pp. 169–186.

[13] C. Curtsinger, B. Livshits, B. G. Zorn, and C. Seifert, "Zozzle: Fast and precise in-browser javascript malware detection." in *USENIX Security Symposium*, 2011.

[14] "The moz top 500," https://moz.com/top500, accessed: 2017-04-30.

[15] SkyLined, "Internet explorer iframe src&name parameter bof remote compromise," https://goo.gl/wUaV3X, accessed: 2017-06-30.

[16] B. Spengler, "Pax: The guaranteed end of arbitrary code execution," *G-Con2: Mexico City, Mexico*, 2003.

[17] "Data execution prevention (dep)," https://support.microsoft.com/en-us/help/875352/a-detailed-description-of-the-data-execution-prevention-dep-feature-in, accessed: 2017-04-30.

[18] C. Cowan, F. Wagle, C. Pu, S. Beattie, and J. Walpole, "Buffer overflows: Attacks and defenses for the vulnerability of the decade," in *DARPA Information Survivability Conference and Exposition, 2000. DISCEX'00. Proceedings*, vol. 2. IEEE, 2000, pp. 119–129.

[19] P. Indyk and R. Motwani, "Approximate nearest neighbors: towards removing the curse of dimensionality," in *Proceedings of the thirtieth annual ACM symposium on Theory of computing*. ACM, 1998, pp. 604–613.

[20] R. Sommer and V. Paxson, "Outside the closed world: On using machine learning for network intrusion detection," in *Security and Privacy (SP), 2010 IEEE Symposium on*. IEEE, 2010, pp. 305–316.

[21] N. Provos, M. Friedl, and P. Honeyman, "Preventing privilege escalation." in *USENIX Security*, vol. 3, 2003.

[22] "Firefox svg animation remote code execution," https://www.mozilla.org/en-US/security/advisories/mfsa2016-92/.

[23] "Firefox browser," https://www.mozilla.org/en-US/firefox/new/, accessed: 2015-04-30.

[24] "Firefox exploit," https://www.exploit-db.com/exploits/9181/, accessed: 2015-04-30.

[25] "Internet explorer," https://www.microsoft.com/en-us/download/internet-explorer.aspx.

[26] "Vmmap," https://docs.microsoft.com/zh-cn/sysinternals/downloads/vmmap, accessed: 2017-04-30.

[27] "Procdump," https://docs.microsoft.com/zh-cn/sysinternals/downloads/procdump, accessed: 2017-04-30.

[28] "Minidump files," https://msdn.microsoft.com/en-us/library/windows/desktop/ms680369, accessed: 2017-04-30.

[29] "Too much freedom is dangerous, understanding ie11 cve-2015-2419 exploitation," https://goo.gl/5YCzLA, accessed: 2017-04-30.

[30] "cve-2015-2419," http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2015-2419.

[31] "Firefox 50.0.1 - asm.js jit-spray remote code execution," https://www.exploit-db.com/exploits/42327/, accessed: 2017-04-30.

[32] A. Barth, C. Jackson, C. Reis, T. Team *et al.*, "The security architecture of the chromium browser," 2008.

[33] D. Wagner and P. Soto, "Mimicry attacks on host-based intrusion detection systems," in *Proceedings of the 9th ACM Conference on Computer and Communications Security*. ACM, 2002, pp. 255–264.

[34] C. Parampalli, R. Sekar, and R. Johnson, "A practical mimicry attack against powerful system-call monitors," in *Proceedings of the 2008 ACM symposium on Information, computer and communications security*. ACM, 2008, pp. 156–167.

[35] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna, "Automating mimicry attacks using static binary analysis," in *Proceedings of the 14th conference on USENIX Security Symposium-Volume 14*. USENIX Association, 2005, pp. 11–11.

[36] Y. Song, M. E. Locasto, A. Stavrou, A. D. Keromytis, and S. J. Stolfo, "On the infeasibility of modeling polymorphic shellcode," *Machine learning*, vol. 81, no. 2, pp. 179–205, 2010.

[37] T. Toth and C. Kruegel, "Accurate buffer overflow detection via abstract payload execution," in *Proceedings of the 5th international conference on Recent advances in intrusion detection*. Springer-Verlag, 2002, pp. 274–291.

[38] C. Kolbitsch, P. M. Comparetti, C. Kruegel, E. Kirda, X.-y. Zhou, and X. Wang, "Effective and efficient malware detection at the end host." in *USENIX security symposium*, 2009, pp. 351–366.

[39] J. Demme, M. Maycock, J. Schmitz, A. Tang, A. Waksman, S. Sethumadhavan, and S. Stolfo, "On the feasibility of online malware detection with performance counters," *ACM SIGARCH Computer Architecture News*, vol. 41, no. 3, pp. 559–570, 2013.

[40] D. Whyte, E. Kranakis, and P. C. van Oorschot, "Dns-based detection of scanning worms in an enterprise network." in *NDSS*, 2005.

[41] G. Gu, R. Perdisci, J. Zhang, and W. Lee, "Botminer: Clustering analysis of network traffic for protocol- and structure-independent botnet detection," in *Proceedings of the 17th Conference on Security Symposium*, ser. SS'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 139–154. [Online]. Available: http://dl.acm.org/citation.cfm?id=1496711.1496721

[42] Z. Lin, J. Rhee, X. Zhang, D. Xu, and X. Jiang, "Siggraph: Brute force scanning of kernel data structure instances using graph-based signatures." in *NDSS*, 2011.

[43] P. Movall, W. Nelson, and S. Wetzstein, "Linux physical memory analysis." in *USENIX Annual Technical Conference, FREENIX Track*, 2005, pp. 23–32.

[44] Z. Lin, J. Rhee, C. Wu, X. Zhang, and D. Xu, "Dimsum: Discovering semantic data of interest from un-mappable memory with confidence," in *Proc. ISOC Network and Distributed System Security Symposium*, 2012.

[45] 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 conference on Security Symposium*. USENIX Association, 2014, pp. 255–269.

[46] ——, "Dscrete: Automatic rendering of forensic information from memory images via application logic reuse." in *USENIX Security Symposium*, 2014, pp. 255–269.

[47] K. Xu, H. Xiong, C. Wu, D. Stefan, and D. Yao, "Data-provenance verification for secure hosts," *IEEE Transactions on Dependable and Secure Computing*, vol. 9, no. 2, pp. 173–183, 2012.