Effective Detection of Credential Thefts from Windows Memory: Learning Access Behaviours to Local Security Authority Subsystem Service

Patrick Ah-Fat
Imperial College London

Michael Huth
Imperial College London

Rob Mead
Microsoft

Tim Burrell
Microsoft

Joshua Neil
Microsoft

Abstract

Malicious actors that have already penetrated an enterprise network will exploit access to launch attacks within that network. Credential theft is a common preparatory action for such attacks, as it enables privilege escalation or lateral movement. Elaborate techniques for extracting credentials from Windows memory have been developed by actors with advanced capabilities. The state of the art in identifying the use of such techniques is based on malware detection, which can only alert on the presence of specific executable files that are known to perform such techniques. Therefore, actors can bypass detection of credential theft by evading the static detection of malicious code. In contrast, our work focuses directly on the memory read access behaviour to the process that enforces the system security policy. We use machine learning techniques driven by data from real enterprise networks to classify memory read behaviours as malicious or benign. As we show that Mimikatz is a popular tool seen across Microsoft Defender Advanced Threat Protection (MDATP) to steal credentials, our aim is to develop a generic model that detects the techniques it employs. Our classifier is based on novel features of memory read events and the characterisation of three popular techniques for credential theft. We integrated this classifier in a detector that is now running in production and is protecting customers of MDATP. Our experiments demonstrate that this detector has excellent false negative and false positive rates, and does alert on true positives that previous detectors were unable to identify.

1 Introduction

The past ten years have seen a dramatic increase in the number and sophistication of cyber-attacks. These changes have as a consequence that the devices and networks of private citizens, businesses, and government agencies can often no longer be protected against all types of such threats. That malicious actors have already infiltrated a network of computers and devices is therefore nowadays a generally accepted wisdom.

This assumption gives great importance to the detection of suspicious or malicious activity that happens within a network perimeter, so that security measures can be taken to contain such activities and the damage they may cause. Modern enterprise security systems thus must ally protection with detection in order to be able to expose intruders that have managed to defeat protection mechanisms and, in doing so, have already compromised the security of the network.

Our research in intrusion detection, like that of others [2, 9, 11], therefore assumes that an attacker has already compromised a network through some vulnerability such as a security-critical bug in system software. For effective intrusion detection we therefore need reliable tools to identify actions within the network that very likely stem from an intruder who has malicious aims.

The development of such tools will benefit from models of intent for attacks. For example, if the intent of an attacker is to steal monetary assets from a bank network, this gives us an idea of a set and possible sequence of actions an attacker would take in that network. Conversely, the identification of malicious actions that were taken can help a security monitor with understanding dynamically the intent of an attacker.

Some of these actions are building blocks of many different types of malicious intent. Detecting actions that are shared in many attacks is therefore particularly valuable for identifying that an actor who already penetrated a network is conducting an attack from the inside; it also provides flexibility in identifying behaviours that are common to a variety of attacks.

In this paper, we focus on one particular such action known as credential theft [3, 6, 14, 17–19]. It is well known that credential theft is a very common behaviour in attackers who have compromised a machine. Indeed, credentials are sensitive information that are valuable since their possession may constitute a means of privilege escalation or lateral movement: an attacker who has managed to steal security-relevant credentials on a machine within that network could use these credentials, for example, to get administrator rights on a targeted machine, access rights to other machines on the network, and so forth. It is therefore hardly surprising that a

large variety of methods have been developed for stealing credentials [4, 5], including phishing, keylogging, password spraying, and wireless traffic monitoring.

Our work reported in this paper investigates a specific, elaborate technique that advanced attackers, and computer forensic analysts, have developed in order to retrieve users' credentials. This technique repeatedly reads from machine memory in search of credentials [8]. Our focus will be on the application of that technique within the Windows operating system, although our contributions may well be transferable to other operating systems and their security processes and monitoring environments. We made this choice because:

- Windows is an important operating system used widely in private, commercial, and government networks, and
- we have the ability to promote the transfer of salient outcomes of this work into Windows products.

Given our focus on Windows and its Security Architecture, we concentrate on the detection of malicious memory accesses to a particularly sensitive process, the *Local Security Authority Subsystem Service* (LSASS). This critical system process enforces security policies on Windows machines.

LSASS is responsible for performing essential tasks related to security such as verifying user logins, managing passwords, and creating access tokens. As such, this process contains a considerable amount of very sensitive data in its memory, making LSASS a prime target on Windows to any internal attacker that seeks to steal credentials.

For our approach, it is important to note that not only attackers with malicious intent will run a process that reads from LSASS memory. Other processes may read from LSASS memory with perfectly good intent, and perform such reads for the benefit and health of the overall network.

In fact, many anti-virus and security software products for Windows scan the LSASS process memory during daily routine checks in order to detect any potentially corrupted or infected files. Such benign reads widely occur on Windows networks. Indeed, on average over all data provided by Microsoft Defender Advanced Threat Protection (MDATP), the LSASS memory is read 40 million times and by many thousand different processes a day.

Our work therefore faces the problem that memory reads to LSASS can be either malicious or benign. The aim of this research is therefore to develop a method that can decide whether a process that is reading from LSASS memory is simply performing a benign action, for example within the context of a virus scan, or whether it is actually trying to steal sensitive information, notably credentials.

The nature of this problem suggests the use of techniques from machine learning, those that can build models for classification. In our setting, we particularly seek models for binary classification that can judge whether a process that is reading from LSASS memory is malicious or not.

We determined a specific methodology in order to solve this classification problem in a manner that would allow us to deploy its solution in a Windows product. That methodology and the contributions made in its execution are described next.

Methodology and Contributions of paper: Our adopted methodology and contributions are as follows:

1. We use real-world data to determine novel features and a model for a precise analysis, control, and characterisation of LSASS memory access.
2. We demonstrate that these features allow us to compute "signatures" of processes, determined by how these processes access LSASS memory.
3. We further show that different sessions of the same process will tend to aggregate into a single linearly identifiable cluster of such read behaviour.
4. We harvest malicious read accesses to LSASS memory by collecting the reads performed by known penetration testing tools run on MDATP customers' machines.
5. We characterise this harvested data more precisely via a set of linear regressions that enable us to distinguish benign LSASS read accesses from malicious ones.
6. Based on this, we design an effective detector that is able to identify suspicious read accesses to LSASS memory.
7. We experimentally confirm that this detector has low false positive and false negative rates. Also, our detector highlights some interesting instances of true positives.
8. Finally, we deployed this detector in production on MDATP, where it now actively protects customers.

These outcomes of our work also have good potential to be integrated with online detection tools, so that the latter can become more effective. For example, the machine-learning models of the detector may be updated based on daily statistical data collected in Windows systems and enriched with security-relevant information from external sources such as cyber-intelligence or cyber-threats centres. This may also lead to the design of non-linear classifiers in future work.

Outline of paper: Related work is reviewed in Section 2. We detail our methodology in Section 3. Section 4 introduces our novel model for memory reads. Patterns in benign read behaviour are explored in Section 5. The harvest and analysis of malicious memory read behaviour is described in Section 6. We build and evaluate our detector in Section 7 and highlight interesting alerts it produces in Section 8. Section 9 studies how our approach deals with Windows updates. Section 10 considers the resiliency of our detector against adversarial manipulation. We suggest future work in Section 11, discuss our work further in Section 12 and conclude in Section 13.

2 Related Work

2.1 Credential Theft by Memory Dumping

An important way for intruders to retrieve credentials is to read from the LSASS memory. In order to do so, attackers

are required to have the debug privilege to gain access to this protected memory location. Since we assume that the network has been infiltrated by malicious actors, we will also assume that attackers have already gained the debug privilege and are trying to access memory maliciously.

Attackers have designed a variety of tools [12] that can read the memory of the LSASS process on Windows machines to extract credentials – notably, Lslsass, Windows Credential Editor, and Mimikatz. Otherwise benign processes, e.g. procdump or taskmgr, can be abused to perform living-off-the-land attacks by dumping the whole of the LSASS process memory. Such credential theft techniques have been used in large scale cyber attacks, e.g. NotPetya and Olympic Destroyer [13].

As discussed in Section 3, Mimikatz [1] is a common tool of choice for attackers that want to steal Windows credentials, and is the most prevalent of the tools mentioned above in MDATP machines world-wide in 7 months.

Mimikatz offers a variety of Windows system techniques in order to extract sensitive information. These different credential-access techniques can be chosen by passing command line arguments to the Mimikatz executable. Mimikatz can also be launched in an interactive mode without the use of any command line arguments, or it may be invoked remotely via PowerShell – a Windows command-line shell designed for system administrators.

For our work, it makes sense to study the three most popular techniques that can steal credentials from the LSASS memory of a targeted machine. These three techniques are:

- L1 Enumerate logon passwords, with command line argument `sekurlsa::logonpasswords`
- L2 Steal Kerberos tickets, with command line argument `sekurlsa::tickets`
- L3 Pass the hash, with the following command line argument `sekurlsa::pth`

2.2 Memory Dumping Detection

The current detection mechanisms for such credential dumping activity are mainly static [15, 16]. Commonly used antivirus software is able to detect the presence of the executable file mimikatz.exe and contains this threat by quarantining it.

Yara rules are used to detect a malicious software by looking for characteristics or pattern in an executable file. A specific set of Yara rules has been written by Benjamin Delpy, the author of Mimikatz, to recognise Mimkatz executable file. These rules are able to prevent the process Mimikatz from running on a machine. However, these rules cannot be used to prevent attackers from running Mimikatz through remote execution via PowerShell or through process injection. Attackers could also bypass those basic security measures by renaming and recompiling their own custom version of Mimikatz in order to circumvent those detection mechanisms.

Other detection mechanisms are based on the use of so called *honey hashes* that are meant to be stolen and to raise an alert at the time they are reused [7]. However, reliance on that mechanism could leave customers unprotected since this would not detect whenever an attacker steals credentials and retains them for future use in a more elaborate attack.

In contrast, the aim of our work reported in this paper is to analyse and detect the *actual behaviour* that a process exhibits when accessing LSASS memory. Our approach therefore directly aims at detecting malicious behaviour rather than a static proxy for a malicious actor. Our approach is therefore also resilient to the aforementioned countermeasures that an attacker may adopt: process injection and remote execution. This is so since our approach classifies actual behaviour of reads, rather than detecting the presence of programs that are known to facilitate such behaviours.

2.3 LSASS architecture

The Local Security Authority Subsystem Service (LSASS) is a system process present on Microsoft Windows operating systems. Its high level roles include providing services to authenticate to the local computer and domain as well as maintaining information on aspects of security on a machine.

LSASS stores credentials in memory on behalf of users with an active session. These cached credentials allow users to access other resources on a Windows domain that are secured with the same identity, without the user having to re-enter a password every time access is required.

These credentials can take multiple forms, including hashes (NT and LAN Manager) and Kerberos tickets. Credentials are cached locally inside LSASS process memory when a user performs an authenticated action on that machine, such as logging on using Remote Desktop, scheduling a task or executing a process using 'RunAs'.

A handle with `PROCESS_VM_READ` can be opened on the LSASS process object and leveraged to perform a cross-process read of sensitive data. To do this the Security Descriptor of the LSASS process object must explicitly grant the accessor this right. Alternatively the accessor can hold `SeDebugPrivilege`; then access to all non-protected processes is granted regardless of their Security Descriptor.

`SeDebugPrivilege` can be acquired by any member of the Local Administrator group, therefore allowing any local administrator to read the memory of the LSASS process. This technique can be used maliciously to extract from LSASS the cached credentials of other users on a Windows domain. These users may hold higher levels of privilege or access to special resources that the current local administrator does not.

Over the last several years technologies have been introduced to the Windows operating system to prevent this attack vector, such as LSASS as Protected Process Light (PPL) and Virtualisation Based Security (VBS). These works do either prevent access to the LSASS process from a Local Administrator or move sensitive data out of the process entirely. However these technologies are not always enabled on a sys-

tem, as enabling them can present compatibility issues with custom or third party software – and in the case of VBS – can require certain hardware which is not always available. Therefore these technologies are not enabled in all customer environments, leaving cross-process reads of LSASS process memory a workable technique to elevate privilege on a Windows domain that can be leveraged by malicious actors. Our approach to detection thus becomes of particular interest in the cases where PPL and VBS cannot be enabled.

2.4 Learning Intruder Behaviours

Previous works have focused on designing detection mechanisms by learning the behaviour of legitimate and malicious agents. The work in [20] focuses on intrusion detection by learning patterns in system call sequences. It aims at being generic and at identifying different kinds of intrusion such as buffer overflows or denial of service attacks. In [10], the authors focus on analysing system call traces produced by the `sendmail` program and they propose ways of learning rules for detecting normal and abnormal sequences. Other works, which focus on university account theft [22], analyse authentication logs and build on heuristics to introduce features that are sensible to the situation such as temporal-spatial violation or inconsistencies based on resource usage. More specific works have focused on intrusion detection on Android devices [21]. Their machine learning mechanisms rely on features based on dataflow-related APIs which are particularly suited for describing Android application behaviours.

Although these works aim at detecting different kinds of intrusion, our work aims at building precise methods for identifying a specific and particularly common action that intruders may perform which is credential theft from Windows memory. All of those works resorted to heuristics based on security knowledge and domain expertise in order to build features that were suited to analyse their specific situation and that made machine learning algorithms exploitable. Similarly, as our work means to study the memory read behaviours of different programs, we introduce in Section 7 the features that we used in our analyses and we justify our intuition as for why they are meaningful in our scenario.

3 Methodology Driven By Customer Data

We now develop and justify our approach and its methodology, which are motivated by two chief objectives:

- Develop an accurate model for read behaviour of LSASS memory as a basis for classifying malicious reads.
- Protect customers of MDATP against current attacks that steal credentials from LSASS memory through detectors that use such accurate models.

As we aim at protecting MDATP customers, we place great value on the data that originates from machines of genuine MDATP customers, rather than relying on data generated by simulations on test machines. This rationale justifies and shapes the manner in which we collect our data and perform our subsequent analyses. These data are collected as part of an opt-in, commercial relationship between Microsoft and their enterprise customers. The data are collected from individual operating systems as part of the MDATP deployment, in nearly real time, and sent to Azure to provide detection and remediation capabilities. Care is taken to obfuscate personal information before presentation to researchers.

We analysed the processes that were reading from the LSASS memory on all machines that are customers of MDATP and looked for credential dumping tools such as Lslsass, Windows Credential Editor and Mimikatz. The only tool that we effectively observed running in a non-obfuscated manner on customers' machines was Mimikatz.

We thus decided to study the read behaviour of LSASS memory for a set of prominent anti-virus and security software tools, but also for Mimikatz as the most relevant tool an attacker appears to choose. As explained further in Section 4, our intuition for building a model of such read behaviours came from studying the way in which such credential dumping tools work, with a particular emphasis on Mimikatz.

Our data collection and machine learning then proceeded in the following stages:

- We analysed the behaviour of the most prevalent benign software that scan the memory of the LSASS process by examining a random sample of seventy-thousand different machines that run MDATP.
- We then harvested the LSASS read behaviour of credential theft techniques performed by Mimikatz seen on real customers' machines by harvesting such attacks on 244 different machines that run MDATP.
- Based on these data, we trained and tested a detector for credential theft.
- Finally, we analysed the influence of Windows update on the read behaviour of credential theft techniques based on recent data, by collecting and comparing malicious read behaviours on machines running on different Windows update versions.

In Section 11, we discuss potential countermeasures that an attacker could devise in order to bypass our detector. Regardless of bypass techniques, however, the practical benefits of our detector were deemed powerful enough to lead to their integration into the deployed MDATP detection suite.

Our methodology also has the advantage of being rather generic. Therefore, this novel way of characterising malicious processes may be transferable to other kinds of malicious behaviour or may safeguard other parts of the Windows memory.

4 Modelling and Collecting Memory Reads

LSASS contains secrets that are stored in its process memory. Keys and credentials held here are attractive to attackers as they enable them to perform operations such as decrypting

sensitive data, or impersonating users that have a higher level of privilege on the domain.

Tools that read from LSASS memory in order to obtain these secrets, such as Mimikatz, perform a number of cross process reads of the address space of the LSASS process – using Windows API calls such as `ReadProcessMemory`.

These APIs take an address of a target process and size of the portion to be read as their arguments. Assuming the caller has the required privileges to read from the target, the data from this address is then copied to a supplied buffer located in the caller process.

The address space layout of the LSASS process is dynamic and varies depending upon a number of factors, such as Address Space Layout Randomisation (ASLR), the timing of memory allocations, the type and size of data that is stored, and on how long the process has been running. It is therefore extremely difficult to predict the address at which secrets of interest reside within the LSASS memory.

Since one cannot simply provide an address and perform a single read to obtain the required information, tools such as Mimikatz must instead first perform multiple reads of the LSASS process in order to search for addresses at which the sensitive data resides.

Whilst the target process address space is generally unpredictable, there are some fixed frames of reference that can be exploited in this activity. For example, the address of the Process Environment Block (PEB) of the remote process can be obtained by an API call such as `NtQueryInformationProcess`. The target Process Environment Block can then be read to discover the location of certain modules loaded by the process.

Once these module locations are known, pointers from structures that are stored as global variables inside these modules can be read predictably, where this predictability varies slightly with the operating system versions. When these pointers are followed, they lead to other structures that can act as clues that ultimately help to unravel and reveal the address where credentials or other secrets are stored.

For example, the Mimikatz `sekurlsa::logonpasswords` operation enumerates and displays all credentials for logon sessions stored within LSASS and follows this approach:

- It performs multiple remote reads against the LSASS Process Environment Block to retrieve loaded module information for loaded `lsa` package `dll` files.
- For the `MSV1_0` authentication package, it remotely reads all of `msv1_0.dll` into a local buffer. Starting at the base address, it searches for a series of *magic bytes* that are located in close proximity to a pointer to the logon session list.
- It mines a series of magic bytes that are in close proximity to an AES key and Initialisation Vector (IV) located in LSASS memory and remotely reads the key and vector to a local buffer.
- It copies the size of the logon session list out of LSASS memory.
- For each of the items in the session list, it remotely reads various details from the session out of LSASS memory in individual reads – username, domain, SID, etc., and a pointer to where the credentials are stored.
- A number of reads from LSASS are then performed to pull different structures out of LSASS, the last one contains encrypted credentials as a unicode string.
- The encrypted credential material is then decrypted using the previously retrieved key and IV.

These operations result, over short time periods, in a large and varying number of reads of the LSASS process memory.

Since Windows 10 RS3, insights into memory reads of target processes are possible – due to instrumentation of the memory manager. Telemetry from this instrumentation is available to security vendors via the Microsoft-Windows-Threat-Intelligence Event Tracing for Windows provider.

This telemetry information provides details of the calling process, the target process, and the size of the data that was copied. It is not practical to collect all of these events. Even if we were to restrict the collection of these events to the target of LSASS, the volume of data generated from a single host would be too large to be practical at network scale.

Instead, aggregates of this data are created by MDATP. These aggregates summarise data into 5-minute time slices, which we will refer to as *sessions*, that provide the calling process, target process, the total of bytes read and the number of reads that occurred within that time slice.

Given the total number of bytes read and the number of reads within that time period, it seems natural to model the behaviour of credential theft tooling such as Mimikatz by these data aggregates understood as features for machine learning. We conjecture that the manner in which malicious tools extract secrets from LSASS memory is different enough from the read behaviour of LSASS memory for benign tools, when we understand behaviour in terms of those features.

The next sections will report on our experimental work for testing and confirming this conjecture.

5 Patterns in Benign Read Behaviours

Based on the model of behaviour and the features of such behaviour introduced in the previous section, we next investigate the behaviour of LSASS memory reads for benign processes that run anti-virus or other security software. This investigation means to identify whether benign read behaviour follows any particular statistical patterns that could differentiate them from processes that run tools which aim to steal credentials.

To that end, we performed the following experiment:

Experiment 1. *We collected 100k read sessions randomly sampled over 10 days on 70k different machines running MDATP. We plot in Figure 1 the behaviour of 5 of the most prevalent processes to read memory from LSASS: WmiPrvSE,*

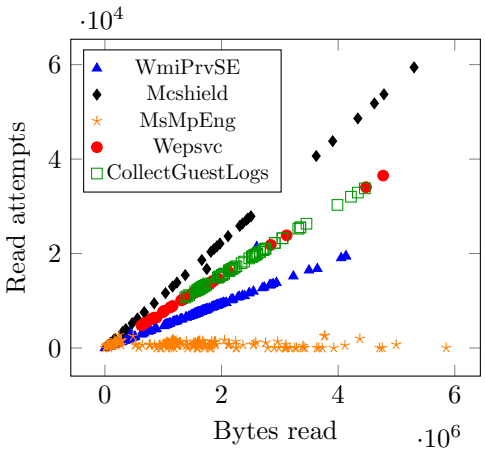

Figure 1: Read behaviour of LSASS memory for the 5 most prevalent security software and anti-virus applications, with data collected over 10 days across 70k different machines. Units are millions (x-axis) ten-thousands (y-axis).

Mcshield, MsMpEng, Wepsvc and CollectGuestLogs which are security software tools which scan the LSASS memory for potentially infected files. Recall that for each of those 5-minute sessions performed by those tools, we recorded the number of bytes read and the total number of read attempts performed during that session. Figure 1 shows the read sessions whose number of bytes read was between 0 and 6 million. We chose this range as it will be of particular interest when comparing benign with malicious behaviours in Section 6.

The plots in Figure 1 suggest that data are very sparse, and that the large majority of the data points appear to aggregate on four different highly linear relationships passing through the origin. For each of those four clusters, we could fit a linear model for classification in terms of the selected features. Some of these linear clusters contain a single process, such as the one formed by `WmiPrvSE`. This suggests that it is reasonable to devise a method for identifying whether a process reading from LSASS memory is indeed `WmiPrvSE` or not – based on the data point that it leaves on this graph.

The processes such as `CollectGuestLogs` and `Wepsvc` seem to lie on the same line, and it would thus be more difficult to distinguish both processes by looking at their read behaviours of LSASS memory, especially for those data points that could represent either of those processes.

However, it is perfectly reasonable that two security software applications may perform similar routine sanity checks on the same area of memory, and therefore may have similar read behaviour of LSASS memory. Fortunately, we have no need to distinguish between different benign processes. At the abstract level of classification of malicious read behaviour of LSASS memory, it is safe to identify such benign processes if their behaviour is sufficiently similar.

Also, the fact that the benign processes we are studying are meant to perform similar tasks might explain why the data that we observe for them is localised into specific clusters.

Our conjecture suggests to study the read behaviour of LSASS memory for malicious processes that are trying to steal credentials, and to determine whether the patterns of such malicious reads are sufficiently different from those of benign read behaviours. This is the topic of the next section.

The classification and the resulting detector that we develop in this paper have uses beyond the mere classification into benign or malicious process. For example, a process whose behaviour seems anomalous compared to all data clusters of applications that are known to be benign may be deemed to be suspicious even if no prior data about that process is available. Similarly, when the actual read behaviour of an application that is known to be benign deviates from what its data cluster would suggest, this can indicate that the application has become infected – triggering an appropriate response.

6 Harvest and Analysis of Malicious Reads

We have seen that the behavioural model of reads from LSASS memory, introduced in Section 4, provides us a useful and effective characterisation of how benign processes access LSASS memory. In this section, we analyse the read behaviour of LSASS memory deemed most important to credential theft techniques. This requires an experiment to collect data from real Windows networks, and a subsequent analysis of that data in order to assess whether our behavioural model allows us to distinguish benign from malicious processes, as claimed in our conjecture.

As already emphasised in Section 3, our research and business objective is to protect Windows customers from threats that are existing at present. Since many current credential thefts from LSASS memory are based on use of the Mimikatz tool, we focus our attention on studying the behaviour of that tool when it is invoked on machines of customers.

To that end, we conducted the following experiment:

Experiment 2. *We collected some data for all instances of Mimikatz invocations that we observed on machines that use MDATP, from January to July 2019. This exercise gathered a total of 1600 Mimikatz instances that were collected on 244 different machines within that 7-month period. Out of these 1600 instances, 256 were interactive sessions and were launched without any command line arguments. We could apply an unsupervised classification technique in order to label these data points, but for simplicity purposes we discarded those entries from our analysis. Thus, we obtained 1344 labelled data points from this experiment, instances of Mimikatz invocations that have one of three possible labels L1, L2 or L3 – denoting which of the three commands listed in items L1-3 in Section 2 were used in these respective instances.*

Out of those 1344 labelled instances, 39 instances were isolated data points which we decided to discard for the purpose

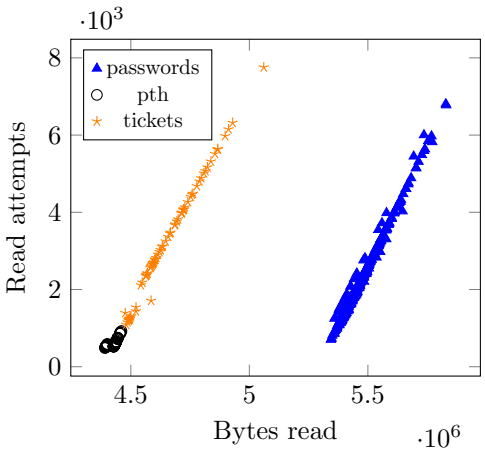

Figure 2: Read behaviour of LSASS memory under credential theft technique L1 (password), L2 (ticket), and L3 (pass the hash). The 1344 data points were collected on 244 different machines during 7 months, as described in Experiment 2.

of our analysis. Most of those discarded were data points with a particularly low number of bytes read, which we considered as failed attempts to run Mimikatz by users who possibly did not have the required privilege to do so. We were therefore left with 1305 labelled instances of Mimikatz invocations.

A plot of the resulting dataset is seen in Figure 2; data is labelled with the particular memory read technique that was used. We compare in Figure 8 in Appendix A the position of those malicious data points to those of benign read instances that were studied in the previous section. This geometric visualisation enables us to make two important observations.

First, the data in Figure 8 shows that processes that steal credentials from LSASS memory follow a read behaviour that is clearly identifiable by our behavioural model introduced in Section 4. Indeed, we can see that the data points corresponding to read access of LSASS memory by Mimikatz are easily distinguishable from those that correspond to read access of LSASS memory made by benign and legitimate security software applications, whose task is to perform regular sanity checks. The data points of malicious processes lie on a very specific and localised area of the plot. In contrast, processes of benign security software create data points that span other regions of the feature space.

This analysis therefore encouraged us to build a detection method based on the fact that malicious activity can be identified by looking at the LSASS memory read behaviour of a process. This detector is developed in the next section.

Second, from the data shown in Figure 2 we can clearly see that our behavioural model of LSASS memory reads enables us to further characterise malicious processes by distinguishing between different credential theft techniques used in such attacks. Therefore, it seems possible to develop a detector that can not only identify that malicious read behaviour of LSASS memory takes place, but that can also classify which credential theft technique is being used in that attack. Such information is valuable as a guide for security response and to improve security intelligence data.

7 Modelling Credential Theft

The experiments and analysis of the collected data on the read behaviours of LSASS memory, described in Sections 5 and 6, suggest that it is feasible to build a detection mechanism for credential theft based on such read behaviour of processes. This section discusses how we built such a detector, based on the data collected in Section 6.

We split the cleaned dataset, containing the 1305 labelled memory reads performed by Mimikatz instances, into 3 datasets following a standard 60%-20%-20% distribution: a training set of size 783, and a validation and a test set both of size 261. In order to build this detector, we studied the different memory read techniques L1-L3 individually and we noticed that each of them was producing data that seemed to aggregate on linear clusters.

The idea of our detector is thus to consider that a data point witnesses a malicious memory read technique if it lies within a certain interval around the corresponding line. To realise this idea, we performed logistic regressions for the data points on the training set, for each of the techniques L1-L3.

We then used the validation set to analyse the influence of the width of the detection interval on the classification rates. Then, we use the test set to establish that we obtain low false positive (FP) and false negative (FN) rates.

7.1 Training

We display in Figure 3 a zoomed-in view of the data points that we collected for the memory read technique L1, which steals logon passwords from LSASS memory. We can see that those points seem to lie on a line, which confirms the effectiveness of our behavioural model. For these data, we performed a logistic regression with the least squares method based on the following model:

$$X = a \cdot Y + b + \varepsilon \tag{1}$$

where X and Y represent the number of bytes read and the number of read attempts respectively, a and b are two constants that we wish to estimate and ε represents the error term. The obtained regression line is depicted on the plot. For all positive real α, we also define the *alert interval* $\mathscr{I}_\alpha$ to encompass all the points which lie within a vertical distance of α standard deviations from the regression line, i.e. all points whose error term defined in (1) verifies $|\varepsilon| \leq \alpha \cdot \sigma$ where σ represents the sampled standard deviation of the errors for that regression. The graph also shows the alert interval $\mathscr{I}_3$.

For the technique L2, which enumerates Kerberos tickets, a very similar plot is shown in Figure 9 in Appendix B. The

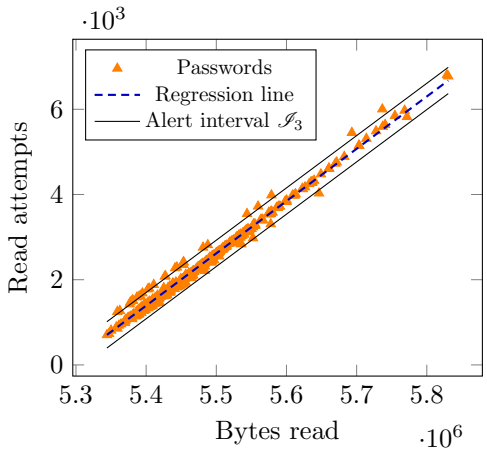

Figure 3: Memory read behaviour of passwords theft.

data acquired for the pass the hash technique (L3) is displayed in Figure 10 in Appendix B. The data points corresponding to this technique are divided into 2 different linear clusters for which we therefore performed separate logistic regressions.

We can empirically observe on those plots that the alert interval seems to include a large majority of the data points. A natural question that arises in this context is what interval width should be chosen in order to build an effective detector.

Large widths would lead to a higher true positive rate but would also increase the false positive rate. On the other hand, more narrow intervals would reduce the false positive rate but would also be more likely to miss out on some true positives.

We study the influence of this interval width on the false positive and true positive rates in the next subsection.

7.2 Validation

For validation, we study how the false positive and true positive rates vary as the margin for our detection interval varies. The aim of this study is to find an ideal trade-off between a low false positive and a high true positive rate.

For this we use a parameter α, a positive real number that is used to determine a threshold value for our detector. We build this detector as follows: for each of the techniques L1-L3, a data point is considered as belonging to that technique if it lies within the corresponding alert interval $\mathscr{I}_\alpha$.

The pseudo-code for this is shown in Figure 4. In this algorithm, α is a positive real number that characterises the detector interval width. The set Γ refers to the set of the 4 linear regressions that we perform for the 3 credential theft techniques, including 2 separate ones for the pass the hash technique. For each regression τ:

- x_τ^{min}, and x_τ^{max} refer to the minimal and maximal number of bytes read for regression τ,
- a_τ, b_τ and σ_τ (resp.) represent the slope, the y-intersect and the standard deviation of the regression residuals.

Input: Data point (x,y) modelling a memory read session
Output: None or alert on malicious memory read

1: **for** τ in Γ **do**
2: **if** $x \in [x_\tau^{min}, x_\tau^{max}]$ **then**
3: **if** $|a_\tau \cdot x + b_\tau - y| < \alpha \cdot \sigma_\tau$ **then**
4: Raise alert for technique τ
5: **end if**
6: **end if**
7: **end for**

Figure 4: Algorithm that detects malicious LSASS memory read behaviour, parametrised by the interval width and a hyper-parameter α for the alert threshold.

This algorithm takes as input a pair (x,y) as representation of a read session: x is the total number of bytes read, and y the total number of read attempts during the process session. For each technique, the algorithm raises an alert if the input data point lies within the respective alert interval $\mathscr{I}_\alpha$.

In order to compute the true positive rate that this algorithm yields, we applied this detection on each of the $N = 261$ Mimikatz instances of the validation set, and then computed the true positive rate as n/N where n was the number of detections that correctly classified these a instances.

Experiment 3. *For the computation of the false positive rate, we ran our detector on all the benign (legitimate) read events that we collected over 7 days from MDATP machines world-wide. This amounted to a total M of 273 million reads from MDATP customers' machines world-wide.*

We then computed the number m of alerts that our algorithm generated for these events, and computed the false positive rate as m/M. Since the number of true positives obtained in a single day is negligible compared to the daily number of read events, the total number of benign read events can be approximated by the total number of reads. In order to compute the false positive rate, our security experts manually examined all the positive cases that our detector alerted on and excluded the true positive ones by inspection.

We also ran our detection algorithm for different values of α and recorded the values of the false positive and true positive rates for these variations, depicted in Figure 11 in Appendix C. Naturally, we can see that, as the detector interval width increases, so do the false positive and true positive rates. After the analysis of this graph, we decided to select the value $\alpha = 3$ for our detector, which seems to yield an excellent trade-off between a high 95% true positive rate and an outstanding false positive rate in the order of magnitude of 10^{-6}.

This chosen value of α corresponds to the alert interval $\mathscr{I}_3$ highlighted in Figures 3, 9 and 10, and we can graphically confirm that this interval covers a large majority of the data points we collected – without being too wide.

Technique	Passwords	PTH	Tickets	Total
# alerts	118	120	11	249
Total	125	125	11	261
TP rate	0.94	0.96	1	**0.95**

Figure 5: True positive rates over the test set for the techniques L1-L3, and the true positive rate across all those techniques.

7.3 Testing

Let us now summarise the results that our detection algorithm yielded on the test set in terms of the true positive rate. Our detector alerted on 249 out of the 261 instances of Mimikatz in the test set. That corresponds to a recall number $249/261 = 95\%$, i.e. an excellent 5% false negative rate. The detailed results that we obtained by running our algorithm on the test set are displayed in Figure 5.

Experiment 4. *In order to test the false positive rate of our detection algorithm, we ran that algorithm on a total of* 80 *million read events that we collected over* 2 *days from* 1.5 *million MDATP customers' machines world-wide and we observed the alerts that it produced. Over the course of these two days, our detection algorithm only alerted on 230 benign read events, which yields an excellent false positive rate of* $2.86 \cdot 10^{-6}$*. This is an acceptable FP rate for production detectors within the MDATP product. Benign processes, such as MsMpEng, can be identified as such based on other techniques including some based on their file name and file hash.*

8 Detecting Malicious Read Access to LSASS

So far, we have built a detection algorithm that models read behaviours as linear regressions. The width of the detection interval is set to achieve satisfactory recall and false positive rate. This detector is running in production in the MDATP code base and helps to protect thousands of customers.

Next, we provide further evidence of the effectiveness of our detector by discussing interesting instances of true positives that it found, including two instances that other detection tools used at present cannot identify. Further information about those processes is displayed in Appendix D.

Figure 12 depicts a renamed instance of Mimikatz. An attacker performs such renaming in the hope that it obfuscates the presence of this malware and so avoids detection. We can see in the metadata that the flags identifying Mimikatz have not been erased or modified. Therefore, static methods that anti-viruses use or methods based on Yara rules are able to identify that this tool is malicious. The `sha1` hash of the calling process is also identifiable as being for Mimikatz – making for example use of the online tool VirusTotal.

Figure 13 describes a custom process named `lolz.exe` that we suspect is a bespoke and recompiled version of Mimikatz. We can see that all metadata related to Mimikatz have been replaced with a suspicious "Microsoft Windows" flag. As this executable file has been customised and recompiled, Yara rules and anti-virus software would not be able to detect the running of this process.

Finally, Figure 14 shows an instance of Mimikatz being invoked remotely via PowerShell. As the calling process is the legitimate Microsoft tool PowerShell, this process will not be identified as being malicious – neither by Yara rules nor by anti-virus software. This state of affairs would be similar to a situation in which an attacker applies process injection: injecting the code of Mimikatz into another benign process P means that Mimikatz can hide within process P as static rules of anti-virus software or Yara rules cannot identify that process P contains injected code.

In both situations, the advantage of our approach is that it is independent of the calling process and its executable file. Our approach only focuses on the *behaviour* of that process, more particularly on the way that this process reads memory from the LSASS process, which holds very sensitive information.

9 Handling Windows Updates

The analyses performed in the previous sections were based on the data that MDATP collected from their customers from January to July 2019. On July 9th 2019, a major Windows update was publicly released, namely Windows 10.0.17763.615. Most MDATP customers migrated to that version in August 2019 and we noticed a slight impact of that migration on the malicious data points that we were continuously collecting. This led to the following experiment:

Experiment 5. *We collected a total of 1009 Mimikatz instances from 157 different machines over July and August 2019 in order to study the influence of this OS update. We plot in Figure 6 the malicious data corresponding to the technique L1 that steals logon passwords, collected over this period, and we colour each data point according to the version of Windows that was running. Windows versions that correspond to version 10.0.17763.615 or later are denoted as* 10^+ *while others are denoted as* 10^-.

In this section, we only report data from the technique for stealing logon passwords but we made very similar observations for the two other techniques. In Figure 6, we see that the data seems to be clearly separated into two different lines that correspond to credential theft actions on machines with Windows 10^+ and Windows 10^- respectively.

In order to protect current customers on Windows 10^+ against those threats, we reran our analyses on this newly collected data and built a new model for fitting the line corresponding to the new OS version, following a similar approach to that outlined in Section 7.

In order to handle future Windows updates on which the memory read behaviour of credential theft would potentially create new linear clusters, it would be of interest to develop

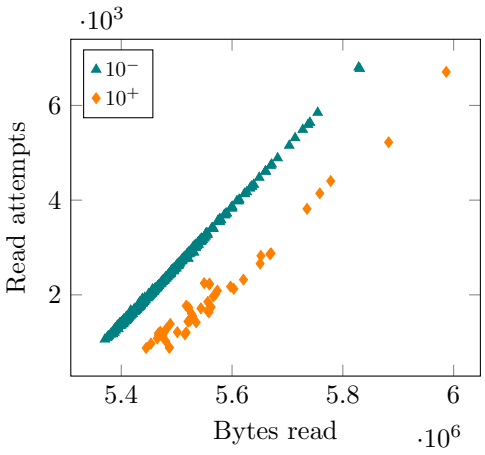

Figure 6: Influence of OS updates on memory read behaviours of logon passwords theft: 1009 password stealing behaviours collected from 157 different machines in Experiment 5 over July and August 2019. The label 10^- refers to the OS prior to the update, whereas 10^+ refers to the OS after the update.

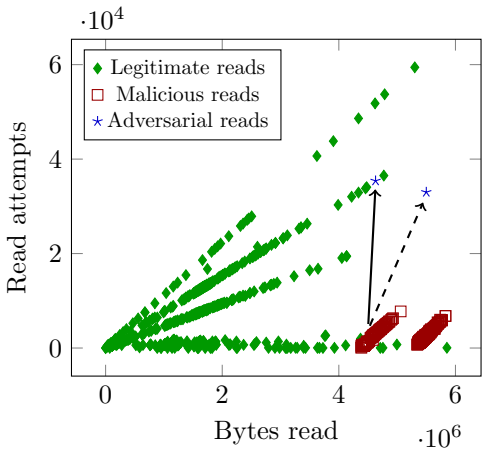

Figure 7: Adversarial memory read behaviours. The dashed arrow represents random read injections required to bypass our current model. The solid arrow represents additional read injections required to impersonate a benign software.

online machine-learning algorithms that can learn how to model new linear clusters automatically as new network data is ingested. One possible idea would be to store all the malicious data points labelled with the corresponding Windows version that they were collected on. For each of the Windows versions, one would then perform a linear regression on all the corresponding data points, possibly by updating already existing regression lines. An optional optimisation step would be to merge different regression lines that would appear to be close, since memory read behaviours do not systematically change when a new OS version is released. We could also consider a multi-class classifier.

10 Minimising our Detector's Attack Surface

In this section, we discuss how an attacker who is aware of our detection mechanism could defeat our detector, and we suggest some improvements that we could implement in order to tackle such evasions. This discussion is set within the context of *adversarial* machine learning.

Injecting random memory reads. A direct and intuitive method for an attacker to bypass our detection model is to inject random memory reads so as to fall outside of the width of the detection interval. Such a potential behaviour is depicted by the dashed arrow in Figure 7. However, we recall that we have also studied in Section 5 the behaviour of legitimate software which follow characteristic patterns. A malicious process injecting random memory reads would highly likely appear as an "isolated" point as shown in Figure 7 that we could detect using *unsupervised* detection methods.

Impersonating benign processes. More interestingly, an advanced attacker could reprogram his attack software so that its read behaviour meets the linear model for some non-malicious behaviour while still being able to steal credentials. Such a potential behaviour is depicted by the solid arrow in Figure 7. Our model in its current form would be unable to detect such an attacker since it would be indistinguishable from the benign software it is impersonating.

However, refining the telemetry that we get from MDATP would enable us to detect such evasions. In addition to its *volume*, we could in particular request more information about the *location* of the data that processes read.

For example, we could look at the addresses at which each read begins: we could expect legitimate processes to always start reading at some predictable – albeit randomised – set of addresses, as they always perform the same tasks.

We could also try to identify the portions of LSASS memory that are read. As legitimate processes read specific portions of memory, we could expect a malicious process to read some portions that are usually unread. Although the address space is not fixed, we could e.g. set some random points in the LSASS memory and receive flags for whether or not a process has read at this address. This would add an interesting dimension to our detector since malicious software would likely raise more flags than legitimate software, or unusual combinations of flags, since they are more likely to browse a large – and maybe contiguous – part of LSASS.

Breaking the 5min window. Our model aggregates read events over a 5-minute window. An attacker could therefore wait 5 minutes in the middle of an attack in order to bypass our detection mechanism. This would split the corresponding data point into several points that would fall outside the detection interval. In order to remedy this, it would be interesting to study the aggregation of events over an adjustable period of time, e.g. by aggregating sessions by process.

Anticipating OS updates. In the online machine learning solution described in Section 9, when a new OS version is released, defensive tools would need a source of new labelled data points in order to re-calibrate the detector as depicted in Figure 6. During this learning phase, an attacker could steal credentials without being detected, provided there is not enough data for this new OS version yet. A possible solution for this problem would be to use a set of training machines, say virtual machines set up with that new OS version, in order to run a couple of credential theft techniques on them and gather the data required to train our model.

11 Future Work

This work focused on detecting credential theft from LSASS memory. We introduced a novel method for characterising memory read behaviour and demonstrated that we can indeed build effective detectors based on this model. A natural and interesting question to pursue is whether or not we can generalise our method. In particular, we would like to study if we could adapt our model to study other parts of the memory of a Windows machine. We would also like to understand whether or not our method could generalise to other operating systems. For example, we could study system files on Linux that play a similar role in Linux to that of LSASS on Windows.

Another avenue would be to study the memory read behaviour of processes accessing a particular area in memory, in particular the memory of web browsers. This memory often contains a wide variety of sensitive information stored in different ways. This would be a particularly fascinating line of inquiry since a detection mechanism that would target a particular browser may potentially be independent of the used operating system, opening up the possibility of building cross-platform detectors. In that context, it would be of interest to study the attacker tool Lazagne, which is designed to steal passwords from web browsers.

12 Discussion

One key contribution of the work reported here is to introduce a model for analysing memory reads and to highlight two important features of such a model that enable us to characterise read sessions precisely. This precision then allowed us to build an effective detector that was not in production already.

These two features have been selected following a detailed analysis of security experts, as explained in Section 4. Those analyses were guided by the thorough examination of credential theft tools and analytical expertise of the Windows security architecture. Experimental data then naturally suggested linear models for classification. In contrast, modelling and selecting features for characterising read sessions using general machine learning techniques might not have given such successful detection results. In particular, this work demonstrates that the application of machine learning techniques can benefit from input given by specific domain experts.

In addition, MDATP has access to a large variety of telemetry from its customers and has therefore the potential to consider read events in a much wider context where read behaviours would depend on a richer dataset such as historical data or network data. In such a case, one could use more advanced machine-learning techniques such as k-means clustering or convolutional neural networks in order to build a model for read sessions, extract sensible features, and train a classification algorithm.

Finally, we discussed in Section 10 some refinements that could improve our method. However, when building a robust detector, we believe that another important aspect to consider is to combine low level signals coming from different sources to produce more reliable – and more meaningful – alerts.

13 Conclusion

In the assumed breach scenario where an attacker has already infiltrated a system, credential theft is a common attack behaviour that allows an attacker to successfully prepare lateral network movement or privilege escalation in preparation of more advanced attacks. A particularly sensitive process targeted by credential thieves on Windows is LSASS, a process that enforces the Windows security policy. Credential theft from LSASS is currently detected based on static methods that recognise the presence of particular binary executable files that are known to perform such actions.

In this work, we introduce a model of memory read behaviour that enables us to study the read accesses of processes to LSASS memory in order to judge whether such processes are malicious or benign. Based on that model and extensive data collected from real Windows networks, we trained and tuned a linear classifier that identifies the actual act of stealing credentials from LSASS memory, rather than detecting the presence of particular malicious executable files. This enables us to detect malicious actors that were previously undetected by currently used tools. In particular, we were able to detect attacks by custom tools or malicious programs invoked remotely by PowerShell. The detection tool that contains our classifier is now running in production and we demonstrated on real customer data that it provides excellent TP and FP rates. We also established that our classifier and therefore our detection tool can cope well with operating system upgrades. Moreover, we highlighted how applying unsupervised machine learning techniques and refining the telemetry that we receive from customers could help us to protect against attackers who would try to evade our detection mechanism.

Acknowledgements. We thank anonymous reviewers and our shepherd William Robertson for their insightful advice.

References

[1] Benjamin Delpy. Mimikatz, 2014.

[2] Dorothy E Denning. An intrusion-detection model. *IEEE Transactions on software engineering*, 13(2):222–232, 1987.

[3] Dinei Florencio and Cormac Herley. Is everything we know about password stealing wrong? *IEEE Security & Privacy*, 10(6):63–69, 2012.

[4] Thorsten Holz, Markus Engelberth, and Felix Freiling. Learning more about the underground economy: A case-study of keyloggers and dropzones. In *European Symposium on Research in Computer Security*, pages 1–18. Springer, 2009.

[5] Chun-Ying Huang, Shang-Pin Ma, and Kuan-Ta Chen. Using one-time passwords to prevent password phishing attacks. *Journal of Network and Computer Applications*, 34(4):1292–1301, 2011.

[6] Markus Jakobsson and Steven Myers. *Phishing and countermeasures: understanding the increasing problem of electronic identity theft*. John Wiley & Sons, 2006.

[7] Parisa Kaghazgaran and Hassan Takabi. Toward an insider threat detection framework using honey permissions. *J. Internet Serv. Inf. Secur.*, 5(3):19–36, 2015.

[8] Abhishek Kumar. Discovering passwords in the memory. *White Paper, Paladion Networks (November 2003)*, 2004.

[9] Wenke Lee and Salvatore J. Stolfo. Data mining approaches for intrusion detection. In Aviel D. Rubin, editor, *Proceedings of the 7th USENIX Security Symposium, San Antonio, TX, USA, January 26-29, 1998*. USENIX Association, 1998.

[10] Wenke Lee, Salvatore J Stolfo, and Philip K Chan. Learning patterns from unix process execution traces for intrusion detection. In *AAAI Workshop on AI Approaches to Fraud Detection and Risk Management*, pages 50–56. New York;, 1997.

[11] Hung-Jen Liao, Chun-Hung Richard Lin, Ying-Chih Lin, and Kuang-Yuan Tung. Intrusion detection system: A comprehensive review. *Journal of Network and Computer Applications*, 36(1):16–24, 2013.

[12] Michael Hale Ligh, Andrew Case, Jamie Levy, and Aaron Walters. *The art of memory forensics: detecting malware and threats in windows, linux, and Mac memory*. John Wiley & Sons, 2014.

[13] Mike McQuade. The untold story of NotPetya, the most devastating cyberattack in history, 2018.

[14] William Melicher, Blase Ur, Sean M Segreti, Saranga Komanduri, Lujo Bauer, Nicolas Christin, and Lorrie Faith Cranor. Fast, lean, and accurate: Modeling password guessability using neural networks. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 175–191, 2016.

[15] J Mulder. Mimikatz overview, defenses and detection, 2016.

[16] David Patten. The evolution to fileless malware. *Retrieved from*, 2017.

[17] David Silver, Suman Jana, Dan Boneh, Eric Chen, and Collin Jackson. Password managers: Attacks and defenses. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 449–464, 2014.

[18] Hung-Min Sun, Yao-Hsin Chen, and Yue-Hsun Lin. opass: A user authentication protocol resistant to password stealing and password reuse attacks. *IEEE Transactions on Information Forensics and Security*, 7(2):651–663, 2011.

[19] Kurt Thomas, Frank Li, Ali Zand, Jacob Barrett, Juri Ranieri, Luca Invernizzi, Yarik Markov, Oxana Comanescu, Vijay Eranti, Angelika Moscicki, et al. Data breaches, phishing, or malware?: Understanding the risks of stolen credentials. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, pages 1421–1434. ACM, 2017.

[20] Christina Warrender, Stephanie Forrest, and Barak Pearlmutter. Detecting intrusions using system calls: Alternative data models. In *Proceedings of the 1999 IEEE symposium on security and privacy (Cat. No. 99CB36344)*, pages 133–145. IEEE, 1999.

[21] Songyang Wu, Pan Wang, Xun Li, and Yong Zhang. Effective detection of android malware based on the usage of data flow apis and machine learning. *Information and software technology*, 75:17–25, 2016.

[22] Jing Zhang, Robin Berthier, Will Rhee, Michael Bailey, Partha Pal, Farnam Jahanian, and William H Sanders. Safeguarding academic accounts and resources with the university credential abuse auditing system. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)*, pages 1–8. IEEE, 2012.

Appendices

A Comparing Benign and Malicious Reads

This section contains Figure 8 which compares the read behaviour of legitimate and malicious software.

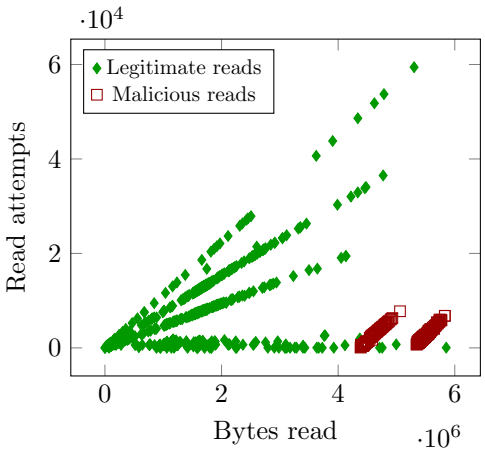

Figure 8: Comparison between benign (i.e. legitimate) and malicious memory read behaviours collected from Experiments 1 and 2.

B Modelling Malicious Read Behaviour

This section contains Figures 9 and 10 which illustrate our regression model for the memory read behaviour of Kerberos ticket thefts and pass-the-hash technique respectively.

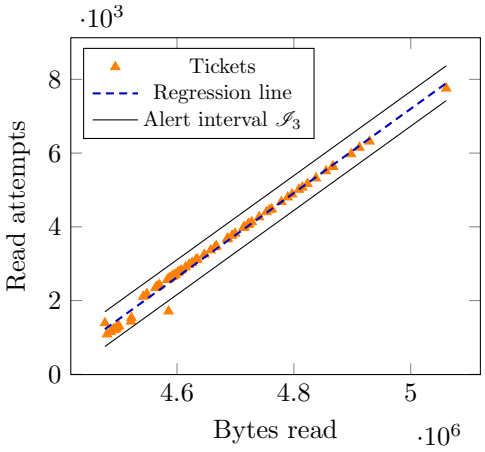

Figure 9: Memory read behaviour of Kerberos ticket thefts.

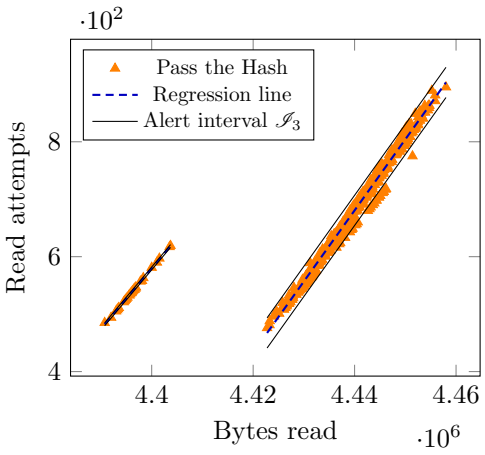

Figure 10: Memory read behaviour of passing the hash.

C Calibrating our Detector

This section contains Figure 11 which describes the influence of the length of the detection interval on the true positive and false positive rates.

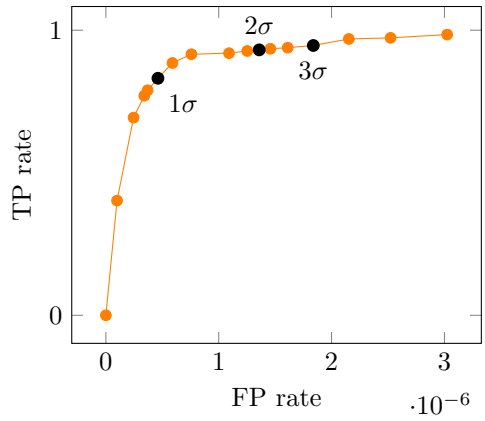

Figure 11: ROC curve: relationship between the true positive and false positive rates as the detector interval width varies.

D Detecting Malicious Memory Reads

This section contains three interesting instances of malicious memory read sessions that our mechanisms detected, including two instances that other detection tools used at present cannot identify. Figure 12 depicts a renamed instance of Mimikatz. Figure 13 describes a custom process named `lolz.exe` that we suspect is a bespoke and recompiled version of Mimikatz. Finally, Figure 14 shows an instance of Mimikatz being invoked remotely via PowerShell.


```
"FileName": "turtle.exe",
"CommandLine": "turtle.exe",
"FilePath": "E:\\",
"Sha1":
 "cb58316a65f7fe954adf864b678c694fadceb759",
"CompanyName": "gentilkiwi(Benjamin DELPY)",
"ProductName": "mimikatz",
"OriginalFileName": "mimikatz.exe",
"InternalFileName": "mimikatz",
"FileDescription": "mimikatz for Windows",
"ParentProcessName": "cmd.exe"
```


Figure 12: Credential theft tool that was renamed in order to avoid static detection but whose executable file can still be identified as malicious via its metadata.


```
"FileName": "lolz.exe",
"CommandLine": "lolz.exe",
"FilePath": "E:\\",
"Sha1":
 "fc03697be2ed32f844aa153460ef5c82f58b06a0",
"CompanyName": "Microsoft Windows",
"ProductName": "Microsoft Windows",
"OriginalFileName": "Microsoft Windows",
"InternalFileName": "Microsoft Windows",
"FileDescription": "Microsoft Windows"
"ParentProcessName": "cmd.exe"
```


Figure 13: Process performing credential theft that has been customised and recompiled so that its executable file bypasses *static* detection mechanisms. Our detector does detect this process as malicious by looking at its memory read behaviour.


```
"FileName":"powershell.exe",
"CommandLine":"\"powershell.exe\" ... { IEX
 $mimi; Invoke-Mimikatz -DumpCreds } ...",
"FilePath":"C:\\Windows\\System32\\
 WindowsPowerShell\\v1.0",
"Sha1":
 "6cbce4a295c163791b60fc23d285e6d84f28ee4c"
"CompanyName":"Microsoft Corporation",
"ProductName":"Microsoft Windows Operating
 System",
"OriginalFileName":"PowerShell.EXE",
"InternalFileName":"POWERSHELL",
"FileDescription":"Windows PowerShell"
"ParentProcessName":"ai_python.exe"
```


Figure 14: Credential theft tool invoked remotely via PowerShell. The malicious executable file is not physically on the machine, so its presence is not detected by current detectors. But our detector triggers an alert when PowerShell performs this action, as its memory read behaviour becomes malicious.