AN IN-DEPTH REVIEW OF POINT-OF-SALE SYSTEM INSECURITY

OH THE POSSIBILITIES

POS INSECURITY

A CASE STUDY

VERSPRITE SECURITY

RESEARCH

POINT-OF-SALE SYSTEM & PCI-DSS

The use of Point-Of-Sale systems can be seen in industries such as retail, hospitality, food service, apparel, grocery, automotive, etc. Any time you swipe a card to make a purchase or utilize a self-checkout kiosk, a Point-Of-Sale system is responsible for handling the intricacies of your transaction in the background.

Given the delicate nature of Point-of-Sale, security standards have been created to protect consumers from malicious actors. The Payment Card Industry Data Security Standard (PCI-DSS) is an information security standard for organizations that handle credit and debit cards issued by major credit card firms [1].

The standard was created to increase controls over cardholder data. The Payment Application Data
Security Standard (PA-DSS), which was created by the Payment Card Industry Security Standards Council (PCI-SSC), dictates that software vendors who develop payment applications must follow a set of best practices to protect cardholder data [2]. However, evidence shows that these standards are not enough to completely thwart the threat of card data compromise. A glossary of terms may be found at the end of the post.

RESEARCH INTO THE STATE OF CARD DATA

Our research into the state of card data within a cardholder data environment (CDE) revealed the following:

1. Data is **not** always encrypted in transit
    - Track data is not encrypted in transit to the payment terminal, allowing for the success of “skimming”.
    - The PA-DSS standard dictates that payment applications “encrypt sensitive traffic over public networks”. However, there is no such requirement for data internal to the CDE.

2. Data is partially encrypted at rest
    - According to the PA-DSS, only the PAN requires encryption at rest. Cardholder names, service codes, and expiration dates may be available in clear-text.
    - Sensitive authentication data such as full track data, CAV/CVC/CVV/CID numbers, and PINs are forbidden from storage, even if encrypted.

3. Data is rarely encrypted in memory
    - The PA-DSS standard recommends that developers create “secure payment applications”. Unfortunately, it does not specifically suggest any secure coding practices, placing the impetus of understanding upon the developer.
    - Despite attempts at safely handling PCI data in memory, RAM scraping remains a viable method for retrieval.

As one might expect, attackers have found ways to exploit the inconsistent state of data protection in POS systems, resulting in a plethora of large-scale breaches. On March 2, 2018, **RMH Franchise Holdings**, an **Applebee's** franchisee, revealed that they had identified "unauthorized software" on their POS systems that "was designed to capture payment card information" [ ]. As an indicator of how far-reaching this compromise was, **RMH Franchise Holdings** owns 167 locations in 15 states. It is unknown at this time how many of the locations were affected by the breach.

Additionally, on November 14, 2017, **Forever21** reported a payment card security incident involving POS malware that "searched only for track data read from a payment card as it was being routed through the POS device". The earliest evidence of infection dated back to April 3, 2017, indicating dwell time of over 7 months [3].

In order to better understand how difficult it is for attackers to compromise these networks, we decided to enumerate the attack surface of a simplified CDE.

ATTACK SURFACE

The customer's initial interaction with a merchant's POS system begins at point-of-interaction (POI) devices. This includes POS terminals and PIN pads. Some vendors such as **Verifone, Ingenico**, and **PAX** offer advanced POI solutions. This is where your cardholder data makes first contact with the POS system. As transit from a card's magnetic stripe to POI is not encrypted, this is a hot target for card skimmers. Each of these devices also includes an underlying operating system capable of running

POS software, introducing additional attack surface.

The next stop for your PII is the POS terminal software. There are thousands of different applications available to merchants for managing their POS needs. Popular POS software include POS solutions offered by **Oracle**'s subsidiary, **MICROS Systems**, as well as **NCR's AlohaPOS**. POS software is responsible for manipulating sensitive cardholder data into payment transaction requests. The real possibility of insecure application configurations or improper handling of sensitive data coupled with the fragmented POS application landscape make this a very interesting component of the attack surface.

The systems hosting POS software have proven to be very lucrative targets for malware authors. They primarily run operating systems such as **Windows Desktop**, **Windows Server Edition**, or **Windows POSReady**. This means that attackers may use well-known techniques to exploit and compromise these targets. Although heavily monitored and rarely left unattended, these systems are often physically accessible to the public. An attacker could potentially leverage physical access to one of these systems in order to gain initial access to the CDE, thus using it as a starting point for lateral movement throughout the CDE. One of the areas of interest to an attacker is the POS database server.

POS database servers are responsible for storing cardholder data as well as other forms of PII.

These servers pose the same risks as database servers in most other environments. In addition, a POS database server compromise may violate the integrity of dependent applications. Some POS software solutions offer a managed POS database solution hosted in the cloud. This black box solution poses risks of its own. The merchant must trust this third-party to properly segment customer data and potentially defend against cloud infrastructure attacks such as “rowhammer” or “spectre" [4,5].

The final egress point for transaction data is from POS payment servers upstream to payment processors. Although there is support for Linux, payment processor applications are primarily supported on the Windows platform. An observant attacker may be able to monitor outbound traffic in order to deduce payment gateway address and attack it directly. Such attacks would affect a much larger number of cardholders, as payment processors handle transactions from multiple merchant CDEs.

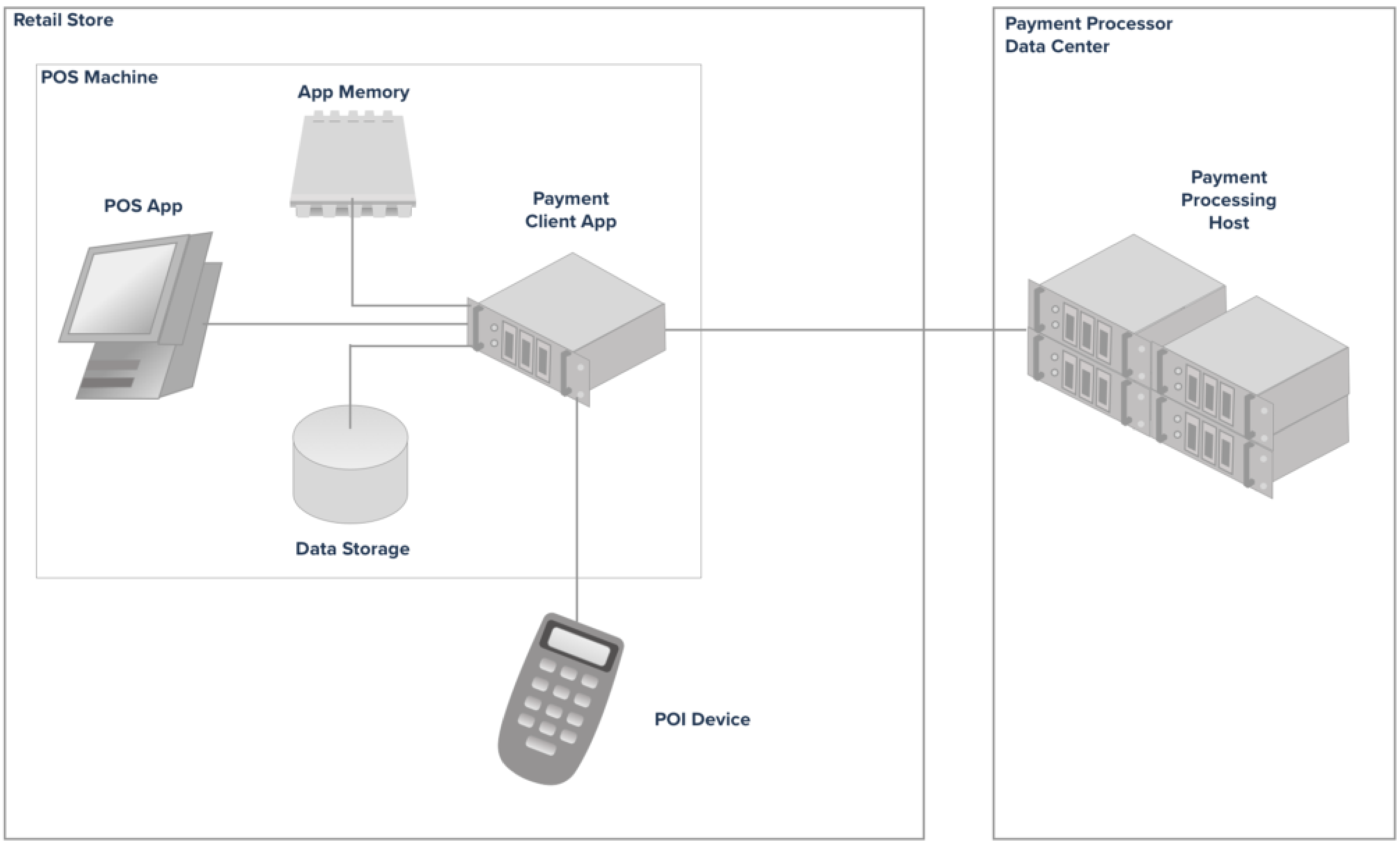
In 2008, **Heartland Payment Systems** suffered a breach that resulted in the compromise of 100 million cards. **Global Payments** also suffered a serious breach in 2011 that affected an estimated 1.5 million payment cards in North America [6]. As you can see, the obscurity of payment processor specifications does not result in their security.

Although attack surface enumeration reveals potential points of vulnerability, it remains a theoretical perspective. In order to understand how this attack surface translates to the real world, we decided to take a look at multiple POS malware samples and their functionality. Apparently, POS malware is not special in terms of initial infection vectors; phishing, brute force attacks, and credential theft shine in popularity. What makes POS malware different is their post-exploitation objectives. In an attempt to steal credit card data from compromised hosts, POS malware includes either RAM scraping or a keylogging functionality.

BlackPOS is a very popular example of RAM scraping malware. A variant of this BlackPOS was responsible for the devastation seen in the **Target** breach. After the attacker accesses the sensitive data, there remains the trouble of data exfiltration. Some malware families, such as **Dexter**, choose to exfiltrate over HTTP. The more recent **UDPoS** malware family employs DNS exfiltration for additional stealth. Overall, POS malware does not appear to have evolved much since its discovery around 2008. This suggests that the risk model for POS systems has not yet changed significantly enough for cyber criminals to adapt new techniques.

RESEARCH FOCUS

Following the enumeration of this attack surface and POS malware analysis, we decided to audit various freeware and software trials for validated PA-DSS compliant POS applications. PCI SSC maintains a list of thousands of validated POS solutions. The applications we reviewed primarily support the **POS EPS Deployment Model** as described by Gomzin in “Hacking Point of Sale” (1st ed., pg. 45) [7].



COMMON MISCONFIGURATIONS IN POS SOFTWARE

Our research revealed a multitude of concerns regarding the secure development of payment applications. We found a POS software application using a custom authentication algorithm which was easy to reverse engineer.

The product had an authentication override system, in which you were required to contact the vendor to receive a proper override code in order to authenticate. The problem was in the way the product implements its override code generation algorithm. This algorithm was easily reverse engineered and could be used to continually generate valid authentication codes which would allow an on-site attacker to access the administrative functionality of the POS. Not only would this allow an attacker full access to personally identifiable information, it would also give the attacker the ability to open up the cash drawer.

Another issue we found was the use of hard-coded credentials in applications. These could be easily reverse engineered and is a grave mistake in the software engineering discipline. We found an example of this in a POS product. The product contained hard-coded credentials for its backend database system, which contained PII, such as customer, order, and partial credit card information.

Additionally, processes should run with the least amount of privileges necessary. If an application is running with elevated privileges and the attacker can leverage a flaw in said application to execute code, then the attacker would have the same access to the system as those elevated privileges would allow. We were able to run our own elevated code in a few instances

Finally, only allow write access to files and directories to the user accounts that need it. With write access to the executable file as a low privileged user, we were able to perform the attack that is laid out in our case study.

RESEARCH FOCUS

A large portion of the payment applications we looked at were implemented in Java. This made for trivial analysis, as Java bytecode may be decompiled quite accurately. These applications are also quite susceptible to tampering by a malicious actor. This becomes even more concerning when payment applications are installed with improper file permissions.

We found one such validated payment application that allowed for any authenticated user to modify its resources in whatever way they see fit. We decided to implement a proof-of-concept attack chain to demonstrate the capabilities an attacker could leverage from this scenario. Although this may sound simple at a high level, the struggles and headaches that we encountered were very humbling as new security researchers.

OUR ATTACK PLAN

1. Inject attacker implant code into the Payment Application JAR from low privilege context

2. Modify a Java CLASS file within the Payment Application JAR to launch our implant code

3. Demonstrate capabilities (Command Execution, Persistence, RAM Scraping, Disinfection)

4. Report to vendor

We developed a Java implant and Python command-and-control server using Flask. The implant communicates to the command-and-control server over an encrypted channel and handles simple commands such as command execution and file upload/download. It also includes more advanced capabilities such as persistence toggling, disinfection, and RAM scraping. The implant CLASS uses the singleton design pattern to avoid multiple instances. All of its functionality is also handled outside of the payment application's primary thread; if the implant crashes, the application continues. We made these design decisions in order to accurately portray the types of capabilities that cyber criminals are using in the wild, thereby gaining insights into how system administrators and software developers can greater protect their assets.

MEMORY DUMP/RAM SCRAPER

In this case, we observed through reverse engineering that the POS application creates String objects for the track data that is being processed by the POS application. We just needed to learn how to get at that data. It is a common tactic used by POS malware to search for CC track data in the address space of the processes running on a system. They use this approach because the CC track data in memory can be read as clear text and can be captured to be processed later. While malware campaigns generally search through the address space of all or most of the processes on the system, we wanted to have a targeted approach when it came to the RAM scraping capability.

Our first thought was to just enumerate all of the memory sections created by the operation system that were given to the Java process, but this was not targeted enough. One thing that we had to learn about was how the JVM actually managed its memory internally, and we found that the JVM handles multiple types of memory. The JVM creates memory sections for its methods, thread stacks, native handlers, and JVM internal data structures. The only memory segment that we were interested was the one that contained the JVM's heap memory. We came to this conclusion because Java's heap memory is used to store class object instantiations and instance variables and would therefore contain the objects of interest. Now in order to target objects on the JVM heap, we need to leak an address of an object that is already allocated on the heap.

Next we had to figure out how to leak an address of an object that was allocated in the JVM heap, thus giving us a target region of memory to scrape for our data of interest. We found a solution that involved the use of a relatively undocumented class. **sun.misc.Unsafe** is used to gain access to low-level mechanisms that are intended to be used by the core Java library. This was all new to us, so we had to learn how to get access to the **Unsafe** class, and we found that you have to use reflection in order to gain access. This is because the `Unsafe`constructor is private, and the caller of the class factory method **getUnsafe()** can only be called by the bootloader. With reflection, you can get access to the field **theUnsafe** which you have to set to accessible with the method call - you guessed it - **setAccessable(true)**. Now that all of that is done, we can retrieve an instance of the Unsafe class via a method call to get(). Our first battle was fought and won, but our journey was just beginning.

>

```
.... Field theUnsafe = null; try { theUnsafe =
Unsafe.class.getDeclaredField("theUnsafe"); } catch
(NoSuchFieldException e) { e.printStackTrace(); }
theUnsafe.setAccessible(true); long objectAddress; …
Unsafe unsafe = (Unsafe) theUnsafe.get(null); String
oracle = "research”; Object[] objects = { oracle }; long
baseOffset = unsafe.arrayBaseOffset(objects.getClass());
....
```

Our next battle was that of dealing with Java’s Ordinary Object Pointers (**OOPs**) and **Compressed OOPs**. The system's architecture, the JRE's architecture, the amount of memory available, and special JRE command line arguments all help to determine how compressed OOPs are used in a given environment. To ensure cross-compatibility, we query this information from the current runtime and decompress the pointer leaked by **Unsafe** appropriately.

We used JNA to bridge C and Java as C would have more powerful process memory capabilities. With our custom code injected into the main POS **JAR**, we start making a few native calls to set up our memory scraping tactic, which is just a more targeted approach to what most malware campaigns use. First, we make sure to get a process handle to the process that we are running as. This is done via calls to **GetCurrentProcessID()** and **OpenProcess()**. **GetCurrentProcessID()** is used to get the process ID of the process that we are running as. That process ID is passed into the call to **OpenProcess()** in order for us to get our process handle. Now that we have our process handle and our leaked Java heap address, we can enumerate the Java heap memory segments via calls to **VirtualQueryEx()**. The **VirtualQueryEx()** function retrieves information about a range of pages within the virtual address space of a specified process. For the **VirtualQueryEx()** calls, we make sure that we actually have access to memory regions returned by checking the **protect** member of the **MEMORY_BASIC_INFORMATION** structure. The **protect** member contains the protection flags used by the memory region that we are querying for.

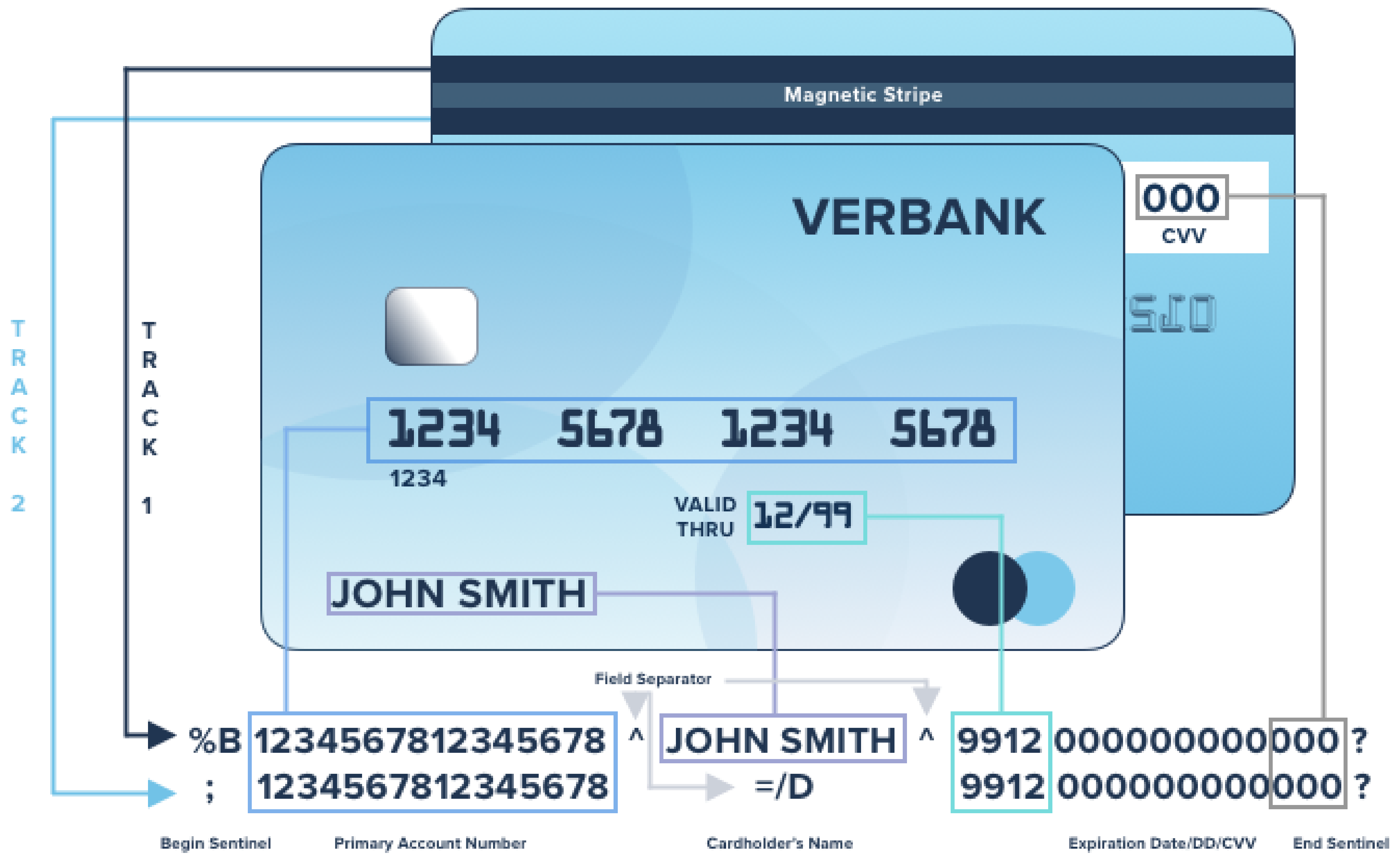
```
.... Payload lib = (Payload) Native.loadLibrary("payload",
Payload.class); DWORD pid =
Kernel32.INSTANCE.GetCurrentProcessId(); Pointer
pHandle =
Kernel32.INSTANCE.OpenProcess(PROCESS_VM_READ |
PROCESS_QUERY_INFORMATION, false, pid);
MEMORY_BASIC_INFORMATION mbi = new
MEMORY_BASIC_INFORMATION(); // Populate the
MEMORY_BASIC_INFORMATION structure with info from the
leaked address
Kernel32.INSTANCE.VirtualQueryEx(pHandle, new
Pointer(objectAddress), mbi, mbi.size()); long address =
Pointer.nativeValue(mbi.allocationBase); long
originalAllocBase =
Pointer.nativeValue(mbi.allocationBase); // Use the
allocation base address from the first query to walk all of
the regions that we interested in. while
((Kernel32.INSTANCE.VirtualQueryEx(pHandle, new
Pointer(address), mbi, mbi.size()) != 0) &amp;&amp;
```

```
originalAllocBase ==
Pointer.nativeValue(mbi.allocationBase)) { if
((mbi.protect.intValue() &amp; PAGE_NOACCESS) != 0 ||
(mbi.protect.intValue() &amp; PAGE_GUARD) != 0) {
System.out.println("[!] Can't read this memory
segment."); ...
```

Then we are able to pass the leaked heap address to our C code to proceed with memory scraping for credit card track data.

```
System.out.println(lib.ScanProcessMemory(buffer,
mbi.regionSize.longValue()).getString(0));
```

Instead of searching through memory with a regular expression, we used a byte-pattern search for the field separators used in track 1 and track 2 data as defined by the **ISO/EIC 7813** standard, similar to the technique used by the **DexterPOS** malware. The track 1 and track 2 data is easily parsed by searching for the beginning/ending sentinel values and field separators. Track 1 contains the following sentinel values: “**%**” for the beginning sentinel, “**^**” for the field separator, and “**?**” for the end sentinel. For track 2 we have the following sentinel values: “**;**” for the beginning sentinel, “**=**” for the field separator, and “**?**” for the end separator. Once the credit card track data is found and validated, it is sent back, as a native pointer, to the backdoored Java client for further operations.



Before we could hijack a **CLASS** file within the **JAR** file, we had to determine which **CLASS** to target. Reviewing source code generated through decompilation revealed an ideal **CLASS**.

1. It is called near the very beginning of the application's start.

2. It is only method performs the very simple operation of returning a version string.

3. It is called very often. This means that if our implant thread dies for any reason, basic use of the application will revitalize our foothold.

Our modifications to this **CLASS** included the code that creates an instance of our implant class and starts the thread prior to returning the version number. In addition, we had to ensure our **CLASS** files are compatible with the

JRE present on the targeted host. To make our implant compatible with **JRE 7**, we kept the implementation of our capabilities very simple. Following the modification of the target **CLASS**, we use the JDK's **jar** utility to update the target with the new and modified **CLASS** files.

Our next step was to automate the workflow of this attack. We wrote an infector script in Ruby designed for compatibility with the **Metasploit Framework**. The infector generates a shared key to be used by both the command and control server and the implant for initial authentication. It then patches the command and control server's hostname, port, and callback interval (as defined within the **Metasploit** configuration) into the implant **CLASS** file. Next, it registers the target host with the command and control server, as unregistered hosts are unable to communicate with implant endpoints. As we compiled our **CLASS** files using **JRE 9**, the infector then patches the implant **CLASS** files with the **JRE** version number of the target payment application, infects a local copy of the payment application **JAR**, stores a backup of the original **JAR** on the command and control server, and replaces the target hosts **JAR** with the infected copy. This process happens within seconds and does not interrupt the POS operator.

Watch the video automating the workflow of this attack.

At long last, we were victorious, but this victory was bittersweet. It is hard to believe that with all of the standards put into place by organizations such as PCI-SSC, something like this was still possible.

MITIGATIONS

Attacks such as this may be prevented in several ways. Ensuring that payment applications are configured with reasonable file permissions would have stopped the this attack chain in it's tracks. Our attack leveraged the fact that weak file permissions allow to inject additional Java code into the POS **JAR** file. Additionally, file integrity monitoring software could be used to trigger alerts upon the modification of critical files. The ideal preventative measure would be to use **P2PE** such that payment applications are never entrusted with cardholder data.

FUTURE RESEARCH

Though the frequency of POS breaches has reduced in recent time, POS systems remain a valuable target for cyber criminals. With the increasing implementation of secure countermeasures to attacks against POS systems such as P2PE and EMV cards, attackers will likely migrate to the next available path of least resistance. The continuation of our POS security research will mirror the attacker's approach in an effort to identify undisclosed weakness in this vast attack surface. This has been quite an exciting project which included many learning opportunities.

GLOSSARY

CAV: Acronym for “Card Authentication Value”. Data element on a card’s magnetic stripe that uses secure cryptographic processes to protect data integrity on the stripe and reveals any alteration or counterfeiting.
CID: Acronym for “Card Identification Number”. See CAV.
CDE: Acronym for “cardholder data environment”. The people, processes and technology that store, process, or transmit cardholder data or sensitive authentication data.
Chip card: Also referred as “EMV card”, “Smart card”, or “Chip and Pin card”. Card that stores its data on an integrated circuit.
CVC: Acronym for “Card Validation Code”. See CAV.
CVV: Acronym for “Card Verification Value”. See CAV.
ISO/EIC 7813: An international standard codified by the International Organization for Standardization and International Electrotechnical Commission that defines properties of financial transaction cards, such as debit or credit cards.
JAR: Acronym for “Java Archive”. A file format used to aggregate many Java class files, associated metadata, and resources into one file for distribution.
Java class file: A file which contains Java bytecode that can be executed on the Java Virtual Machine.
JDK: Acronym for “Java Development Kit”. A superset of the Java Runtime Environment which contains tools for Java programmers.
JNA: Acronym for “Java Native Access”. Provides easy access from Java to things like the Windows API and other external native shared libraries.

JRE: Acronym for “Java Runtime Environment”. Is a software package that contains what is required to run a Java program. It includes a Java Virtual Machine implementation and an implementation of the Java Class Library.

JVM: Acronym for “Java Virtual Machine”. An abstract computing machine that enables a computer to run a Java program.

Merchant: An entity that accepts payment cards as a form of payment for good and/or services.

P2PE: Acronym for “point to point encryption”. Instantaneously encrypts payment card data at the time the card is swiped.

PAN: Acronym for “primary account number” and also referred to as “account number”. Unique payment card number (typically for credit or debit cards) that identifies the issuer and the particular cardholder account.

PII: Acronym for “personal identifiable information”. Information that can be utilized to identify an individual, including but not limited to name, address, Social Security number, phone number, etc.

PIN: Acronym for “personal identification number”. Secret numeric password known to the only to the user and a system to authenticate the user to the system. The user is only granted access if the PIN the user provided matches the PIN in the system. Typical PINs are used for automated teller machines for cash advance transactions. Another type of PIN is one used in EMV chip cards where the PIN replaces the cardholder’s signature.

POI: Acronym for “point of interaction”. The initial point where data is read from a card. An electronic transaction-acceptance product, a POI consists of hardware and software and is hosted in acceptance equipment to enable a cardholder to perform a card transition. The POI may be attended or unattended. POI transactions are typically integrated circuit (chip) and/or magnetic-stripe card-based payment transactions.
POS: Acronym for “point of sale”. Hardware and/or software used to process payment card transactions at merchant locations.
RAM Scraping: Also referred to as “memory scraping”. The activity of examining and extracting data the resides in memory as it is being processed or which has not been properly flushed or overwritten.
Rowhammer: A hardware-based attack that leverages an unintended side effect in dynamic random-access memory that causes memory cells to leak their charges and interact electronically between themselves, possibly altering the contents of nearby memory rows that were not addressed in the original memory access. Has been used in some privilege escalation exploits.
Skimming: The crime of getting private information about somebody else’s credit card used in an otherwise normal transaction. Generally, this requires the use of a physical device, often attached to a legitimate card-reading device.
Spectre: A hardware vulnerability that affects modern microprocessors that perform branch prediction. On most systems, the speculative execution resulting from a branch misprediction may leave observable side effects that may reveal private data to attackers.

Track Data: Also referred to as “full track data” or “magnetic-stipe data”. Data encoded in the magnetic stripe or chip used for authentication and/or authorization during payment transactions. Can be the magnetic-stripe image on a chip or the data on the track 1 and/or track 2 portions of the magnetic stripe.

REFERENCES

[1] Official PCI Security Standards Document Library - PCI-DSS
[2] Official PCI Security Standards Document Library - PA-DSS
[3] Forever 21 Reports Findings from Investigation of Payment Card Security Incident
[4] Exploiting the DRAM rowhammer bug to gain kernel privileges
[5] Meltdown and Spectre
[6] Global Payments Breach Now Dates Back to Jan. 2011
[7] Point of Sale: Payment Application Secrets, Threats, and Solutions, Gomzin 2014