

Efficient Prevention of Credit Card Leakage from Enterprise Networks

Matthew Hall¹, Reinoud Koornstra², and Miranda Mowbray³

¹ No Institutional Affiliation, `mhall @ mhcomputing.net`

² HP Networking, USA, `koornstra @ hp.com`

³ HP Labs, UK, `miranda.mowbray @ hp.com`

An extended abstract of this paper has been published in the Proceedings of CMS 2011, B. de Decker et al. (Eds.) CMS 2011, LNCS 7025, 2011 (Boston: Springer), pp 238-240. The extended abstract is ©IFIP, 2011. The definitive version of the extended abstract is the one in the Proceedings. This is the authors' version of the work. It is posted here by permission of IFIP for your personal use. Not for redistribution.

Abstract. We describe some challenges we have found in deploying applications for preventing the leakage of credit card numbers in traffic on a large enterprise network. To address them, we have developed a new approach to this problem. In contrast to a previously-used method, our new method has higher performance—in a benchmarking experiment, it achieved a throughput more than 4.7 times that for a competitive method—and it can be partly implemented in hardware without any additional libraries if this is required for processing high traffic volumes.

Keywords: cloud security, privacy, data leak prevention

1 Introduction

According to Gartner, Inc., the market in content-aware data loss prevention will reach \$400m in 2011 [1]. Products in this market examine the data layer of a packet on an enterprise network and determine whether it contains private or commercially sensitive data items, so as to prevent these items from being leaked (for example, sent into the public Internet or to a less secure part of the enterprise network). Credit card numbers are a very important category of data which these products must cover. The danger of card number leakage is exacerbated both by the rise of targeted phishing attacks, and by the increasing use of cloud computing and consequent increase in data traffic between enterprises and the public cloud.

In this paper we describe some challenges which we encountered when deploying applications for preventing the leakage of credit card numbers in traffic on a large enterprise network. To address these challenges we have developed and prototyped a new approach to this problem, making use of a novel high-speed streaming Luhn check algorithm. We have applied for a US patent for

the algorithm (application number PCT/US2011/022709), but this paper contains some content relating to the algorithm that is not mentioned in the patent application, including discussion of a benchmarking experiment and of some refinements and extensions of our method. Our approach can prevent the leakage of any credit card numbers, not just a predetermined set of such numbers. In the benchmarking experiment, a software implementation using our approach achieved more than 4.7 times the throughput of an implementation that used a competitive approach. For even higher throughput, our method can be partially implemented in hardware without the need for additional libraries.

2 Previous Approaches

There are several existing applications which address the problem of detecting credit card numbers in data traffic, so as to prevent them from being leaked. These include data loss prevention products and services by Symantec [2], Websense [3], Vericept [4], Mimecast [5] and Code Green networks [6], and the OpenDLP open source suite [7]. With one exception, all of these applications use one of two approaches (some companies offer both approaches).

The first approach, which can be used to prevent the leakage of any credit card numbers, begins by performing a first pass on the packet data to identify candidate numbers that fit a regular expression for potential card numbers. For example, all American Express (Amex) card numbers are 15-digit numbers beginning with 34 or 37, and the regular expression used in this pass should match, among other things, all numerical substrings of the data that have this format. Then a second pass is performed to determine if any of the candidate numbers satisfy the Luhn check [8]. All valid credit card numbers satisfy this check, although the converse does not hold.

One way to carry out the Luhn check is to double every alternate digit in the number including the penultimate digit, and check whether the sum of digits of the resulting numbers is divisible by 10. For example, 932152 passes the Luhn check because the sum of the digits of the numbers 18 3 4 1 10 2 is 20, but 93215 does not because the sum of the digits of the numbers 9 6 2 2 5 is 24, which is not divisible by 10.

The candidate numbers that are found to pass the Luhn check are sent to leakage inspectors for Visa, MC, Amex, etc. that are responsible for determining if any of these sequences are indeed valid card numbers issued by the associated provider. These leakage inspectors have full information about the set of valid numbers issued by the provider. If they identify a candidate number sent to them as being a valid credit card number, this is recorded, and further transmission of the packet or the flow containing this number may be blocked.

Unfortunately, performing the first regular-expression pass is an expensive operation in terms of resource requirements when trying to handle the high packet volumes on modern enterprise networks. Our experience with deploying applications that use this approach is that as packet volumes increase, it quickly becomes impossible or prohibitively expensive to perform the regular-expression

pass on all the packets in the relevant network flows. To use this approach at all, it becomes necessary to sample a fraction of the packets in these flows, and only search the sampled packets for strings fitting the regular expression. Any credit card numbers in the unsampled packets remain undetected.

We considered the possibility of speeding up the first pass by implementing it in hardware. However this would require the regex library, which has considerable size, to be stored in the hardware. We concluded that this was impractical.

The second approach is to store digital fingerprints of a set of card numbers (for example, the card numbers in a customer database) and check the data for exact matches to these digital fingerprints. This eliminates the need to do a Luhn check, but requires all the contents of the data to be checked for matches. The fundamental limitation of this approach is that it can only detect credit card numbers whose fingerprints are in the stored set. It requires a system for obtaining a list of all the credit card numbers that might be leaked, and for keeping this list updated. We rejected this approach because we consider it important to be able to prevent the leakage of any credit card numbers.

The one exception is a data loss prevention product offered by Lancope [9] which uses traffic sampling and looks at the flow telemetry without considering the data layer. This approach provides only an educated guess as to whether there is leakage, and thus suffers from false positives.

3 Our Approach

To address these challenges, we have developed and prototyped a new approach to detecting credit card numbers in data traffic on an enterprise network. In contrast to the approach using regular expression matching, our approach carries out Luhn checks in a first rather than a second pass of the data.

The process begins by collecting network traffic for analysis. Any traffic flows that do not need to be checked are excluded from analysis using Access Control Lists defined by the customer to fit their network layout. Where packet volumes are so high that it is still necessary to use sampling, we use the sFlow sampling standard [10]. The packets collected are decoded into the character sets used in the customer's locale.

As the batch of packets is collected, we run a novel high-speed streaming algorithm on the resulting data stream. In a single pass of a data stream, this algorithm identifies all 14, 15 or 16-digit numbers that satisfy the Luhn check, and which are substrings of the stream obtained by deleting all spaces and dashes from the original stream. The algorithm is simple and easy to implement in hardware, without the need for additional libraries, if this is required for processing very high traffic volumes.

Lightweight custom string check functions (in software) are then run on the set of numbers that are reported by the algorithm as passing the Luhn check, for example to identify potential Amex card numbers by checking whether any of these numbers have length 15 and begin with 34 or 37. The combination of the Luhn check algorithm and string check performs the same end function as

the first two passes of the approach that uses regular expressions, but with much less computation. This allows us to inspect more (possibly all) of the relevant packets in a network with high packet volumes than would be feasible using the same amount of resources and regular expression matching. In networks with low enough packet volumes that the previous approach could inspect all relevant packets, it allows us to do the same with less resources.

The numbers that satisfy the checks are saved into a table which is directed to leakage inspectors. Any sequences that are identified by an inspector as being valid card numbers are saved into another table along with key data from the packet to produce an event report for the user.

Our prototype implements the streaming Luhn check, the string check, and the event report generation, but not the final step of blocking a packet from further transmission if it has been found to contain a credit card.

The event reports generated by our prototype include the IP and MAC addresses of the sender and, if resolvable, the recipient; the last four digits of the credit card number; the sampling device used, if relevant; vendor-specific data, for example a measure of the magnitude of the leakage; and an event ID number. We can identify IP, ICMP, TCP or UDP data as relevant. If the leak is found in IP data and the transport layer protocol is unknown, IP data is reported. For known IP subprotocols, the report includes data for that protocol, e.g. type(s) and codes(s) for ICMP or Source→Dest port mappings for TCP or UDP.

3.1 Streaming Luhn Check Algorithm

The pseudocode for the algorithm is below. The notation $\text{sd}(a, b)$ is shorthand for the string consisting of entries $d[a], d[a+1], \dots, d[b]$, where $a, b \in \mathbb{Z}$ with $0 \leq a \leq b$.

The underlying idea of the algorithm is as follows. The vector d stores the sequence of digits received from the stream since the beginning or the last character other than a space, dash or digit, and i records this sequence's length. When a new digit is read in from the string, i is updated and the variable $x[i]$ is set such that $x[i]$ is equivalent mod 10 to $L(\text{sd}(1, i))$. Then the algorithm determines whether the substrings of length 13, 14 or 15 ending at this new digit pass the Luhn check.

To determine this, the algorithm uses the fact that if $s1, s2$ are digital strings and $s2$ is of even length, and $s1 \cdot s2$ is the concatenation of $s1$ with $s2$, it follows from the definition of L that $L(s2) = (L(s1 \cdot s2) - L(s1))\%10$. If $i > 13$, putting $s1=\text{sd}(1, i-14)$, $s2=\text{sd}(i-13, i)$ in this equation implies that $\text{sd}(i-13, i)$ passes the Luhn check if and only if $(x[i]-x[i-14])\%10 = 0$. The checks for $\text{sd}(i-14, i)$ and $\text{sd}(i-15, i)$ can be derived similarly by setting $s1=d[i-14]$, $s2=\text{sd}(i-13, i)$ and $s1=\text{sd}(1, i-16)$, $s2=\text{sd}(i-15, i)$ respectively.

Start by setting $i=0$, $d[0]=0$, $x[0]=0$.

While there are more entries in the string, repeat the following:

 Get the next entry, and set e to it

```

if e is other than a base-10 digit, space or dash
  set i = 0
if e is a base-10 digit
  increase i by 1
  set d[i] = e
  if i == 1 set x[1] = e
  if i > 1
    set x[i] = d[i] + 2d[i-1] + x[i-2]
    if d[i-1] > 4 increase x[i] by 1
  if i > 13
    set c = (x[i] - x[i-14]) % 10
    if c == 0 report sd(i-13,i) as passing the check
  if i > 14
    add d[i-14] to c
    if c % 10 == 0 report sd(i-14,i) as passing the check
  if i > 15 and (x[i] - x[i-16]) % 10 == 0
    report sd(i-15,i) as passing the check

```

The underlying idea of the algorithm is as follows. The vector d stores the digits received from the stream since the beginning or the last character other than a space, dash or digit, and i records this vector's length. When a new digit is read in from the string, i is updated and $x[i]$ is set such that $x[i]$ is equivalent mod 10 to $L(\text{sd}(1,i))$. Then the algorithm determines whether the substrings of length 13, 14 or 15 ending at this new digit pass the Luhn check.

To determine this, the algorithm uses the fact that if s_1, s_2 are digital strings and s_2 is of even length, and $s_1 \cdot s_2$ is the concatenation of s_1 with s_2 , it follows from the definition of L that $L(s_2) = (L(s_1 \cdot s_2) - L(s_1))\%10$. If $i > 13$, putting $s_1=\text{sd}(1,i-14), s_2=\text{sd}(i-13,i)$ in this equation implies that $\text{sd}(i-13,i)$ passes the Luhn check if and only if $(x[i]-x[i-14])\%10 = 0$. The checks for $\text{sd}(i-14,i)$ and $\text{sd}(i-15,i)$ can be derived similarly by setting $s_1=d[i-14], s_2=\text{sd}(i-13,i)$ and $s_1=\text{sd}(1,i-16), s_2=\text{sd}(i-15,i)$ respectively.

3.2 Benchmarking Experiment

The performance advantage of our approach depends on how frequently credit card numbers and other numerical strings appear in the input data. For a benchmarking experiment we used two data files, one in which they appear frequently and the other in which they appear infrequently. Both files consisted of a million lines of 80 ASCII characters each. Every line of File 1 contained a 14, 15 or 16-digit string, at a random offset. Half of these strings were chosen so that they would pass the streaming Luhn check and string check—or equivalently, the regular expression pass and Luhn check pass. The other characters in the lines were random. File 2 consisted just of random ASCII characters.

We processed the two files 40 times with our prototype (which is in software) and 40 times with a software implementation of the approach using a regular expression pass followed by a Luhn check pass.

The experiment used an Intel Core 2 Duo E8500 CPU, with 3.166Ghz and 6144 KB of layer 2 cache. It had 4 GB of RAM, 2.9 GB of which was available to the operating system (32-bit Linux Ubuntu 10.4).

For each run we measured the time taken from when the packet data was available in memory until the two checks or two passes had been completed for the whole file. Every run took within 15% of the average time for the file and method, which for File 1 was 8.32988 seconds with the regular expression method and 1.70410 seconds with our new method, and for File 2 was 8.61835 and 0.88082 seconds respectively. All but one of the runs took within 6% of the average time. The shortest time for the regular expression method divided by the longest time for our method was 4.72 for File 1, and 9.37 for File 2: thus, the implementation using our method achieved more than 4.7 times the throughput of the implementation using the competitive method.

4 Refinements and Extensions

There are some possible refinements to the Luhn check algorithm that will make our approach more effective in certain circumstances.

- If it is known that the digits of card numbers appearing in the data will be consecutive, rather than potentially separated by spaces or dashes, the condition `if e is other than a base-10 digit, space or dash` can be replaced by the condition `if e is other than a base-10 digit`.
- The computation requirements can be further reduced, at the expense of a small increase in memory use, by using lookup tables in the calculation of `x[i]`.
- We did not find the amount of memory used to be a problem in our experiments with the algorithm. However, only the 17 most recent entries of the `x` vector and the 16 most recent entries of the `d` vector are ever used, so if memory resources are scarce they can be conserved by over-writing earlier entries of these vectors. If necessary, memory requirements could be further reduced at the expense of a small increase in computation by setting `x[i]` to `(d[i] + 2d[i-1] + x[i-2]) % 10` rather than `d[i] + 2d[i-1] + x[i-2]`.
- The international standard for identification card numbers [11] requires all numbers issued by the banking or financial industries to begin with the digits 3, 4, 5 or 6. Therefore the number of strings processed by the string check can be reduced at the expense of slightly more computation during the Luhn check pass, by replacing the condition `if e is a base-10 digit` by the condition `if i>0 and e is a base-10 digit, or i=0 and e = 3,4,5 or 6`. This tradeoff can be further extended by modifying the Luhn check algorithm to only report numbers beginning with 3, 4, 5 or 6.

These refinements would still allow the algorithm to be easily implemented in hardware without additional libraries.

An obvious extension of our method for detecting credit card numbers is to apply our approach of using a streaming algorithm rather than a regular

expression check to the detection of some other types of personal data. For example, International Bank Account Numbers (IBAN, [12]) consist of a two-letter country code followed by a numerical string of a length l that depends on the country, and satisfy a checksum. It is straightforward to write a streaming algorithm which looks up l for the appropriate country if the last two characters received are a country code, and checks whether the next l characters are numerical digits; and if they are, checks the checksum requirement and reports the relevant string of length $l + 2$ as a possible IBAN if it is satisfied. Similarly, a streaming algorithm might be used to detect any numbers in a particular national ID or tax number scheme, with higher throughput than would be possible with the use of regular expression matching on packet data.

The prototype detects card numbers in data traffic. There is increasing use of protocols such as SSL which transmit data in encrypted form. This protects data while it is in transit, but leaves open the possibility that personal data may be transmitted by mistake and misused after the packet containing it has been decrypted by the recipient. Several companies offer products that can intercept data before transmission (they are known as Endpoint DLP products), for example Symantec, Code Green Networks and Trend Micro. If used in combination with some means of intercepting data, our method could be used to inspect data before it is transmitted, and block its transmission where necessary.

Our approach could also be used to inspect static data files in areas of the enterprise network that are not supposed to contain personal data.

References

1. Ouellet, E., Proctor, P.E.: Magic Quadrant for Content-Aware Data Loss Protection. Gartner RAS Core Research Note G00200788, 2 June 2010. Gartner, Inc. (2010).
2. Symantec Data Loss Prevention products and services, <http://www.symantec.com/business/theme.jsp?themeid=vontu>
3. Websense data security, <http://www.websense.com/Content/DataMonitor.aspx>
4. Vericept Corporation: Vericept Awarded Patent For Data Loss Prevention Technology, Press Release, 15 January 2008. <https://www.vericept.com/index.php?id=913>
5. Mimecast data leak prevention solutions, <http://www.mimecast.com/data-leak-prevention-solutions/>
6. Code Green networks, <http://www.codegreennetworks.com/index.htm>
7. OpenDLP Data Loss Prevention suite, <http://code.google.com/p/opendlp/>
8. Luhn, H.P: Computer for Verifying Numbers. US Patent 2,950,048, 23 August 1960.
9. Lancope Inc. Data Loss Prevention. Market Brief MB07092010 (2010).
10. sFlow.org—Making the Network Visible, <http://www.sflow.org>
11. International Standards Organization. Identification cards - Identification of issuers - Part 1: Numbering system. ISO/IEC 7812-1:2006.
12. International Standards Organization. Financial services - International bank account number (IBAN) - Part 1: Structure of the IBAN. ISO standard 13616-1:2007.