Towards Blocking Outgoing Malicious Impostor Emails[*]

Erhan J. Kartaltepe   Shouhuai Xu

Department of Computer Science, University of Texas at San Antonio

{ekartalt,shxu}@cs.utsa.edu

Abstract

Electronic mails (emails) have become an indispensable part of most people's daily routines. However, they were not designed for deployment in an adversarial environment, which explains why there have been so many incidents such as spamming and phishing. Malicious impostor emails sent by sophisticated attackers are perhaps even more damaging, because their contents, except the attachments, may look perfectly legitimate while silently targeting certain critical information such as cryptographic keys and passwords. In this paper, we explore a mechanism for blocking malicious impostor emails called ContAining Malicious Emails Locally *(CAMEL), which aims at blocking compromised victim user machines from further infecting others.*

1. Introduction

Emails were not originally designed as a utility in an adversarial environment, which may explain why there have been so many incidents such as spams and phishing emails. Both spams and phishing emails may still be seen as benign because their contents do not automatically attack other machines. Email worms [1] could do more damage *automatically* because an infected machine could duplicate itself to all the users in its address book.

Malicious impostor emails are a type of email worms introduced in [4]. Such emails look perfectly legitimate, except their attachments, when "double-clicked," could infect the victim's machine and further spread by themselves. Compared with traditional email worms, they could adopt more sophisticated and crafty spreading strategies other than, e.g., a simple-minded fast-spreading that may be easily detected and contained [8]. Moreover, malicious impostor emails may be sent by an adversary that has compromised a legiti-

mate user's machine and obtained all its cryptographic secrets. This means that the adversary could perfectly impersonate the victim user to its email servers, and thus some non-cryptographic mechanisms are needed.

A mechanism dealing with malicious impostor emails was explored in [4], which essentially is based on the associations of the outgoing-incoming email servers. However, it does not address the case that the "claimed" email senders could have been compromised by the adversary. For example, Alice may receive a malicious impostor email that claims to be sent by her friend Bob, while it was actually sent by an attacker, who has compromised Bob's machine, via Bob's affiliated email server. This paper aims to address such an attack by containing such impostor emails. Specifically, we explore a mechanism called "ContAining Malicious Emails Locally" (CAMEL), which consists of two system components. One is a profiler specifying an honest user's pattern of requesting an outgoing email server to send emails. The other is a Reverse Turing Test (RTT) [6] that aims to distinguish a request issued by a human being from one issued by a program. We implement a prototype system by integrating a concrete implementation of CAMEL into the SMTP server. Further, we explore its usefulness via simulations. Experimental results suggest that CAMEL would be practical and effective.

Outline. This paper is organized as follows. In Section 2 we briefly review the malicious impostor email problem. In Section 3 we explore the CAMEL mechanism. We discuss related work in Section 4, and conclude the paper in Section 5.

2. Malicious Impostor Emails

Consider an email network consisting of users and servers. There is an adversary that may break into some users' computers and thus compromise their secret information such as passwords and cryptographic keys. The adversary is so crafty that the corruption may not be detected within a certain

---

period of time. For simplicity, we assume that the adversary is a probabilistic polynomial-time algorithm that cannot compromise the email servers or tackle certain hard AI problems [6].

Suppose an email consists of three parts: a *header* (including the `From` field), a *non-attachment content*, and an *attachment*. For a given email EMAIL, let sender(EMAIL) be the sender id (i.e., the value in the `From` field of the header). For each email user U, let whitelist$_U$ be the whitelist of email addresses that will be accepted by U, Filter$_U$ be a filter that takes as input an email, analyzes its non-attachment content, and outputs a decision on whether it thinks the email is suspicious (e.g., a spam or phishing email), wormScanner$_U$ be a worm/virus scanner that takes as input an email, analyzes its attachment, and outputs a decision on whether the attachment is suspicious. We assume that Filter$_U$ is perfect, meaning that any spam or phishing email will be detected. However, wormScanner$_U$ is not perfect, meaning that it can only detect the malicious attachments that possess known signatures; they have limited success in dealing with polymorphism worms/virus and cannot deal with unknown (zero-day) worms.

Definition 1 ([4]) *A malicious impostor email is an email,* EMAIL, *sent to a recipient U with* (whitelist$_U$, Filter$_U$, wormScanner$_U$) *such that*

1. $\Pr[\text{sender}(\text{EMAIL}) \in \text{whitelist}_U] = 1$, *meaning that the email possesses a sender address that is on U's whitelist.*

2. $\Pr[\text{Filter}_U(\text{EMAIL})$ *outputs "suspicious"*$]$ $=$ 0, *meaning that the non-attachment content is perfect and cannot even be detected by a human being.*

3. $\Pr[\text{wormScanner}_U(\text{EMAIL})$ *outputs "suspicious"*$] \geq$ θ *for some* $0 \leq \theta < 1$.

3. The CAMEL **Mechanism**

Basic idea. The CAMEL mechanism is enforced at the legitimate outgoing email servers. It consists of profilers for the affiliated users and a RTT system. A profiler can be obtained by applying an appropriate machine learning algorithm to the history data of a user. A popular RTT system is the CMU CAPTCHA project. Given a user's profiler and a RTT system, CAMEL operates as follows: When the user requests its outgoing email server to send an email, the server decides, according to the profiler, whether to issue a RTT challenge. If an issued RTT challenge is not correctly solved, the request is dropped. If no RTT challenge is issued or the

issued RTT challenge is correctly solved, the email is sent.

Implementation. We integrated a concrete implementation of the CAMEL mechanism, namely machine learning based profilers and the CAPTCHA system, into a standard SMTP server. The resulting system consists of two computers connected to a 100M Ethernet LAN: one machine `hermes` acted as the user's machine using an email client called Pooka [7], and another machine `jupiter` acted as the outgoing email server's machine using the sendmail server version 8.12.9. Both machines run the Linux operating system.

Figure 1 highlights the control/information flow of the resulting system. The functionality of the CAPTCHA server is to provide a random CAPTCHA challenge, verify the answer, and return a predicate indicating whether the answer is correct. The functionality of the CAPTCHA Tester running on the user's machine is to display a CAPTCHA challenge to the user and gets the entered answer.
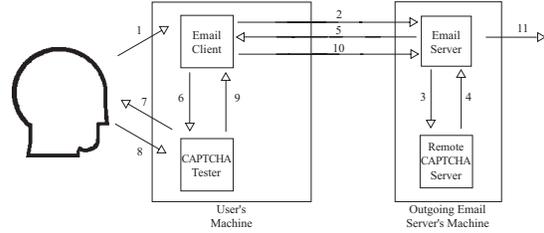

Figure 1. An example scenario

A verbal interpretation of the above scenario follows:
1. The user composes an email using the email client.
2. The email client requests the email server to send the email.
3. Suppose the request does not pass the profiler. The email server asks the CAPTCHA server to generate a CAPTCHA challenge.
4. The CAPTCHA server returns a challenge and its answer.
5. The email server responds to email client with the CAPTCHA challenge.
6. The email client calls the CAPTCHA tester to deliver the challenge.
7. The CAPTCHA tester delivers the challenge.
8. The CAPTCHA tester collects the user's answer.
9. The CAPTCHA tester forwards result to email client.
10. The email client passes the result to email server.
11. The email server verifies the result of the challenge. If the result is correct, the email server sends

the email to the recipient as in the original system; otherwise, the request may be dropped or re-challenged (depending on the system policy).

Figure 2 is an instance of the resulting email client program Pooka with a CAPTCHA challenge.
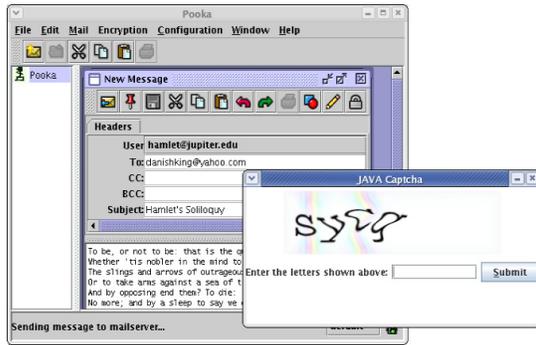

Figure 2. Pooka with CAPTCHA

Analysis. Now we analyze CAMEL in terms of its `security`, `delay time`, `ease-of-deployment`, and `effectiveness`. Some of them are based on the aforementioned concrete implementation.

`Security`. Recall that the adversary may have already compromised a victim legitimate user, meaning that the adversary could have obtained the victim user's secret key or password, and that the adversary can impersonate the victim user in any cryptographic authentication. This justifies why we have to rely on non-cryptographic mechanisms for blocking outgoing malicious impostor emails.

For a given profiler, we denote by β its *false negative rate*, namely the probability of a malicious email being treated as legitimate, and by α its *false positive rate*, namely the probability of a legitimate email being requested to solve a RTT challenge. Assume that the RTT system is secure, meaning that no computer program can correctly solve the RTT challenges with a reasonably high probability. Then, a malicious impostor email is not blocked by CAMEL with probability β, meaning that a malicious impostor email attack succeeds with probability β The above analysis suggests, qualitatively, that as long as the profilers have a low *false negative rate* and the RTT system is secure, CAMEL can block most of the impostor emails. This naturally leads to two interesting questions: What are the average false negative rates in real-life? Can we establish a quantitative understanding on the security achieved by CAMEL?

In order to answer the first question, we conduct case studies based on two real-life datasets that are available to us. The first dataset is based our Department SMTP log for six weeks. It consists of one email server and 285 user accounts, among which only 187 (or 65.6%) had sent more than 20 emails — the minimum number necessary to extract meaningful profilers. We applied Weka's [3] decision tree algorithm to learn profilers for these users. Experimental results indicate the average false negative rate is $\beta = 33.1\%$, and the average false positive rate is $\alpha = 2.8\%$.

The second dataset is that of [2], which contains 58,339 user accounts and 14,172 email servers for the period of three and a half months. Unlike our Department SMTP log, a small percentage (1.7%) of users had enough information to extract their profilers. For those users we apply the same decision tree algorithm to extract their profilers. Experimental results indicate: the average false negative rate is $\beta = 24.1\%$, and the average false positive rate is $\alpha = 5.1\%$.

To answer the second question raised above, we conduct a simulation study based on the aforementioned second dataset because it involves a significant number of email servers and users. In the simulation, we assume that there are users (i.e., their computers) that are initially compromised. To accommodate the worst case scenario, we assume that any compromised user's outgoing emails are all substituted with emails sent by the adversary, and that each non-blocked impostor email has an attachment that is always "double-clicked" by the recipient (i.e., the recipient's machine is immediately compromised). Note that the simulation methodology is consistent with our adversarial model as the adversary does not adopt a fast-spreading strategy. To simplify the simulation, we assume that each malicious impostor email is not blocked with probability β, where β is the average false negative rate of the profilers.

Figure 3 shows the effect of different deployments of CAMEL averaged over 10 runs: (a) no servers have deployed CAMEL, (b) 25% randomly-chosen servers have deployed CAMEL, (c) 50% randomly-chosen servers have deployed CAMEL, (d) 75% randomly-chosen servers have deployed CAMEL, and (e) all servers have deployed CAMEL. On the other hand, we consider the cases that originally 0.5%, 1%, and 2% randomly-chosen accounts are initially compromised. We plot the number of compromised users over the whole dataset lifetime of 10^7 seconds, or three and a half months.

Figure 4 is a reorganized version of Figure 3. It clearly suggests that, for a given initially compromised user set, the percentage of servers deploying CAMEL plays a crucial role. For instance, in the
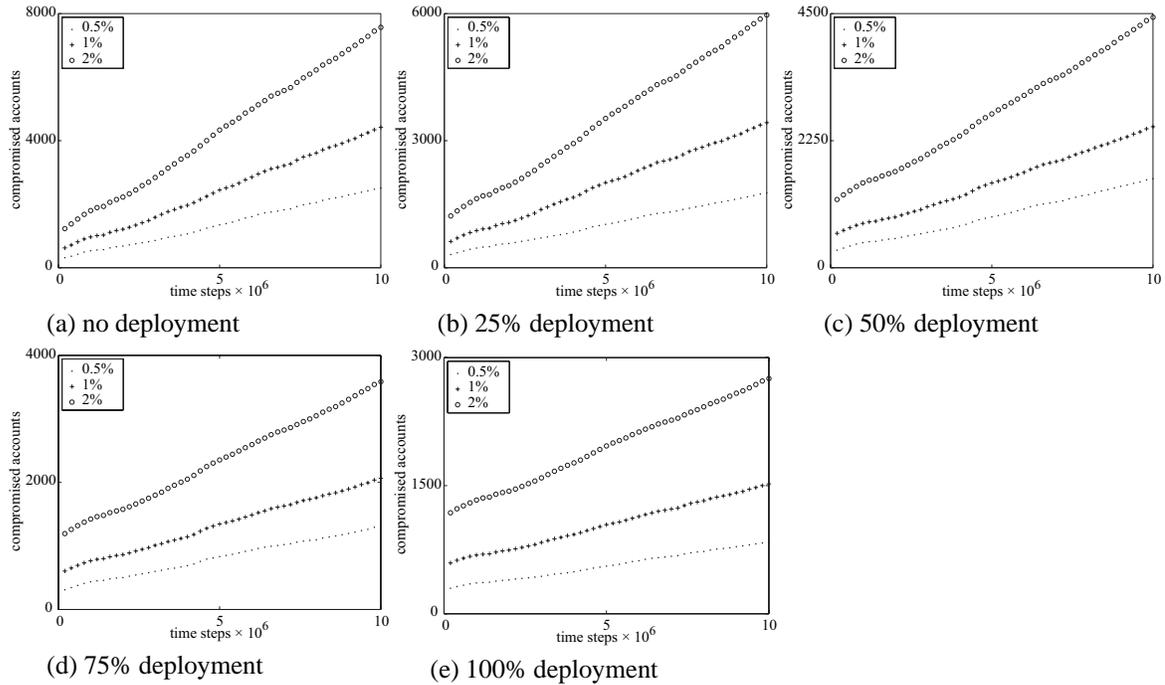
Figure 3. Effect of initially compromised user sets with fixed CAMEL **deployment**
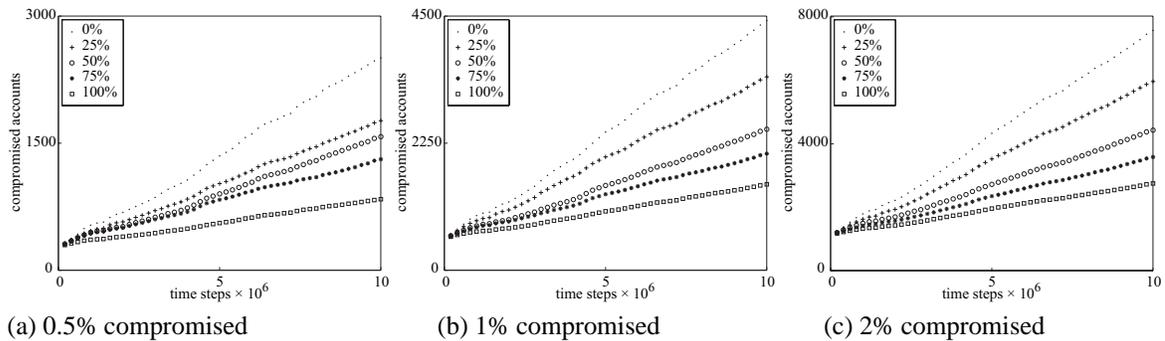

Figure 4. Effect of CAMEL **deployment with fixed initially compromised user sets**

case of 2% users (or 1,168 out of the 58,339) initially compromised, 0% deployment leads to the compromise of another 6,304 users, 50% deployment leads to the compromise of another 3,268 users, and 100% deployment leads to the compromise of another 1,586 users.

`Delay time.` Ideally, the deployment of CAMEL should not impose any noticeable delay in the email-sending process on the legitimate users. We define `delay time` as the period from the issuance of a send request to the completion of the SMTP protocol, except the delay on entering the solution to a RTT challenge (which depends on one's typing speed). Further, this should be true even when an outgoing email server is serving a number of requests.

In order to measure the `delay time` imposed

by CAMEL, we conduct experiments of 10000 concurrent outgoing email requests at the rates of 100, 1000, and 5000 requests per minute, respectively. To accommodate the worst case scenario, we assume that every request for sending an email is challenged with a RTT challenge. For each of the three experiments, we obtain the corresponding `delay time` by averaging 10 independent runs We summarize the `delay time` (in ms) in Table 1, which indicates the following: in the original architecture (i.e., with no CAMEL), the delay times with respect to different request arriving rates are about the same; this may be due to the fact that all the requests are issued on the same Ethernet LAN. Moreover, the delays imposed by the concrete implementation of CAMEL (mainly the CAPTCHA system) are truly insignificant, as they are less than 30 ms.

Table 1. Delay imposed by CAMEL

Transaction/minute	w/o CAMEL	w/ CAMEL
100	165.295	186.422
1000	165.295	188.661
5000	165.295	195.270

`Ease-of-deployment.` As showed in our prototype implementation, CAMEL can be easily and incrementally deployed, due to the fact that changes to existing software packages are confined within local organizational networks. Moreover, it is clear that CAMEL can be seamlessly integrated into a larger system framework for countering malicious impostor emails (e.g., one that integrates the solution explored in [4]).

`Effectiveness.` By `effectiveness` we mean "How often is an honest user challenged with a RTT?" It is understood that if an honest user is often requested to solve RTT challenges, the user might simply disable such a mechanism. As a matter of fact, it depends on a user profiler's *false positive rate*, α, namely the probability that an honest user's email is treated as illegitimate. As mentioned above, based on the two real-life datasets, we extracted profilers of false positive rates 2.8% and 5.1%, respectively. Therefore, they may not be seen as disruptive.

4. Related Work

The most closely related work is [4], which introduced the malicious impostor email problem and explored countermeasures against a class of it. The present paper explored a mechanism countering a complementary class of malicious impostor emails at the outgoing servers. Nevertheless, it can be showed that the two mechanisms can be seamlessly integrated into a unified framework.

Space limitation only allows us to mention a few other prior works; we refer the reader to [4] for more details. The spreading of malicious impostor emails could be fundamentally different from the spreading of email worms, particularly because the former are not necessarily fast-spreading. This has actually been confirmed by our preliminary simulation study showed in Section 3. As a consequence, existing epidemic models of, and countermeasures against, email worms (e.g., [5, 8]) are not applicable. The simulation-based study of email worms [9] is different from ours, because the former considers a network consisting of only end users, whereas we consider a network consisting of both end users and email servers.

5. Conclusion

We explored a CAMEL mechanism against malicious impostor emails enforced by outgoing email servers. Experiments show that it is efficient, and simulations suggest that its wide deployment significantly improves the overall system security. Simulation results also suggest the need for a new theoretic model for explaining the spreading behavior of malicious impostor emails.

References

[1] `http://www.viruslist.com/en/ viruses/encyclopedia?chapter= 152540408#email.`

[2] `http://www.itp.uni-bremen.de/ complex/email_net.html.`

[3] Weka. `http://www.cs.waikato.ac. nz/ml/weka/.`

[4] E. Kartaltepe and S. Xu. On Automatically Detecting Malicious Imposter Emails. Proceedings of the 2005 International Workshop for Applied PKI (IWAP'05).

[5] J. Kephart and S. White. Directed-Graph Epidemiological Models of Computer Viruses. Proceedings of IEEE Symposium on Security and Privacy 1991.

[6] M. Naor. Verification of a human in the loop or identification via the turing test. `http://www.wisdom.weizmann. ac.il/~naor/onpub.html.`

[7] A. Petersen Pooka: A Java Email Client. `http://suberic.net/pooka`

[8] M. Williamson. Design, Implementation and Test of an Email Virus Throttle. Proceedings of Annual Computer Security Application Conference (ACSAC) 2003.

[9] C. Zou, D. Towsley, and W. Gong. Email Worm Modeling and Defense. Proceedings of International Conference on Computer Communications and Networks (ICCCN'04).