Typhoid Adware

Daniel Medeiros Nunes de Castro, Eric Lin, John Aycock, and Mea Wang
Department of Computer Science
University of Calgary
2500 University Drive NW
Calgary, Alberta, Canada T2N 1N4
{dmncastr,linyc,aycock,meawang}@ucalgary.ca

March 9, 2010

Abstract

Typical strategy for adware authors is to install their software on as many machines as possible and, for each affected machine, display advertisements to the user(s) of that computer. In this paper we present a different model: *typhoid adware*. Typhoid adware is more covert, displaying advertisements on computers that do *not* have the adware installed. We prove that this is a viable adware model with three proof-of-concept implementations and discuss possible defenses, for which we have two proof-of-concept implementations.

1 Introduction

In the beginning of the 20th century, a cook named Mary Mallon was infected with a highly contagious disease called typhoid fever, but she did not have the symptoms and at first she did not even know she was infected. Later, when informed that she was infecting others with typhoid, she refused to believe health authorities and she ended up infecting an estimated 47 people in total, some of whom died [10, 23]. This true story may seem far removed from the realm of malicious software, but that is not the case – it is a new model for adware.

We can loosely define adware as a program that has a marketing purpose and displays advertisements on the computer screen, possibly along with some other functionality. The advertisements can vary from pre-defined pictures and text to more personalized ones, advertisements customized using information about a user's searches or visited websites [7].

Whether it is because adware is simply annoying or because adware can cause actual harm, with privacy violations, bandwidth usage, and computer resource consumption, adware has become known as a potential threat. As a consequence, recent versions of most antivirus products have included adware

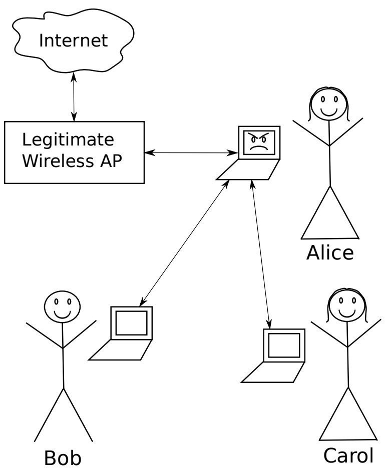

Figure 1: Cappuccino time at the Internet café

detection capability. In some cases, this may fall under the category of so-called "gray area" detection [15] or "potentially unwanted applications" [19].

A user may therefore think that because they have up-to-date antivirus software, they do not need to worry about adware. But what if the menace is not in the user's computer, just close by?

Imagine the situation shown in Figure 1. A café provides wireless Internet access through a wireless access point; normally traffic to and from the Internet passes from customers' laptops through the access point. Now say that Alice's laptop has a new type of adware installed. This adware convinces Bob's and Carol's laptops to communicate through it rather than the legitimate access point, and then automatically inserts advertisements in the content that Bob and Carol see. Alice, meanwhile, sees no advertisements to tip her off; her adware-infected laptop is simply a carrier. For this reason, with reference to the ill-fated Mary Mallon, we call this new type of adware *typhoid adware*. Bob and Carol. for their part, see advertisements but no adware, because the typhoid adware is not present on their machines. In general, the idea is that the typhoid adware chooses victims from its neighbors, intercepting their connection and then altering their network traffic, inserting advertisements into the actual content.

It is easy to dismiss typhoid adware as just a man-in-the-middle attack, but this is shortsighted. First, this application of man-in-the-middle attacks is novel and has some aspects (like the protection of video content) that are specific to this scenario but not to the general man-in-the-middle problem. Second, with Internet access becoming increasingly available in public spaces, threats like typhoid adware taking advantage of the physical proximity of victims are likely to become more prevalent; we hope to stimulate discussion with this work regarding how to deal with these threats proactively. Third, the answer to the technical question “is a typhoid adware attack possible using current hardware?” is not at all obvious, and we had to experiment extensively to come to a reasonable solution. In effect, we have mapped out the attack space so that defenders will have an idea of how this adware threat may manifest itself.

We have developed three proofs of concept to demonstrate that this *is* a viable threat, tested it on wired and wireless networks, and inserted advertisements into both HTML and streaming video. The general idea can be extended for other types of network and applications.

The following sections are organized as follows. We first give the necessary background to explain typhoid adware. Sections 3 and 4 discuss our implementations and experiments, respectively, followed by defenses and related work in Sections 5 and 6. Finally, Section 7 has future work and conclusions.

2 Background

A full understanding of typhoid adware requires a broad background, from networking to video streaming, which we include here for completeness.

2.1 TCP/IP: IP vs. MAC addresses

In a TCP/IP environment, a host is usually identified by an Internet Protocol (IP) address. This address is a 4-byte number[1] that must be unique in that particular network. However, when computers (or other network devices such as routers and switches) actually exchange data in a local area network (LAN), they use a different address called the Media Access Control (MAC) address. This address is six bytes long and is unique for each network interface, being hardcoded in it while still in the factory.

The TCP/IP stack has a protocol responsible for the conversion from IP to MAC address called ARP: Address Resolution Protocol [14]. Every time that a network interface joins onto a network, it sends a broadcast message to all the members of the LAN that identifies its IP and MAC address. Other devices may save this information into a table, called an ARP table, for future use.

When a device wants to send some data over the network, the operating system needs to decide what MAC address the message will be sent to, based on the IP address of the destination. If the IP address is not in the local network, it will be forwarded to the MAC address assigned to the gateway of that network. To identify the MAC address of the destination (or of the gateway), the operating system first checks the ARP table; if it cannot find it there, an ARP message is broadcast over the network. The reply to this ARP message must contain the MAC address of the host responsible for that IP address. With the MAC address of the destination or the MAC address of the gateway, the data can be sent.

Further description of this process can be found in many sources, such as [17].

[1]We assume IPv4 in this paper unless stated otherwise.

2.2 ARP Spoofing

ARP spoofing is a well-known attack; see [6] and [24], for example.

The process of spoofing, or pretending to be another host, relies on the existence of those ARP tables. In our case we want to pretend to be the gateway, so we can intercept all the traffic from and to a specific host, our victim.

To achieve this, we keep sending to our victim an ARP message announcing that the MAC address of our malicious host is responsible for the IP address defined as the default gateway of the network, gathered during the configuration of the network interface.

On the other hand, the traffic coming from the gateway must also be intercepted, otherwise the victim might receive data from the same IP address but from a different MAC address, indicating that there is something wrong happening. So, we also must send ARP messages to the gateway, announcing our MAC address as responsible for our victim's IP address as well.

Each one of the hosts (the victim and the gateway), with constant updates to their ARP tables, will not request MAC address for each other while the attack is happening. Thus, all the communication between the victim and the external world will pass through our malicious host before going through the actual gateway of the network.

2.3 Flash Video Format

The most popular Flash video format is the FLV format; the description in this section is based on [1]. An understanding of this format is needed to appreciate the challenges we encountered when performing on-the-fly video modification (Section 3.3) as well as the defenses we suggest in Section 5.2.

The FLV format is basically divided into two sections:

FLV header
Until the most recent version, contains nine bytes that identify the format (FLV version) and the content in a general way, i.e., if it is audio, video, or both.

FLV body
The actual content, which is a variable number of pairs (*length*, *FLVTag*).

In the following subsections, we describe the FLV header, the FLV body and the elements known as FLV tags. The format is illustrated in Figure 2.

2.3.1 FLV Header

This part of the file always starts with a 3-byte long constant string, "FLV," to identify the file type. The next byte is the version of the format (0x01), followed by a sequence of flags indicating the existence of audio and/or video and ending with the size of the header (0x09).

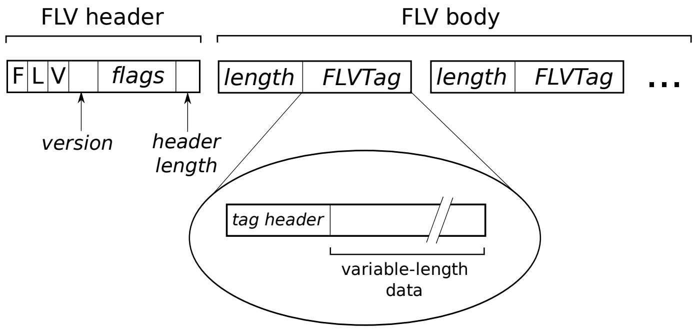

Figure 2: FLV format

2.3.2 FLV Body

The FLV Body consists of a variable number of pairs (*length*, *FLVTag*), where:

Length

A 4-byte long value that represents the size of the *previous* FLVTag, so the very first value for length will always be 0. This value is important as it allows the programmer to implement backward movement on the player.

FLVTag

A sequence of bytes of variable size that contains a tag header and the actual data.

2.3.3 FLV Tags

Like the file, each FLV tag also contains a header and a body of data.

The tag header gives information like the type of data (audio, video, or script data), the size of the data, and its timestamp.

If the data contains audio or video, the first byte of the tag body is used to specify the sound format or codec (depending on the tag type) used to produce the data.

There is also important information that is stored in the first byte of a video tag: the frame type. One specific type of frame is what is called a “keyframe,” or “a seekable frame,” according to the specification. Key frames are used to reverse and fast forward and are also used as references to the other frames after them. Basically the keyframe contains a full image and the other frames only have the data that changed from the previous one.

The first three FLV tags are a metadata tag, a script tag that stores information about the video, followed by one video tag that is marked as a keyframe (the first keyframe in the file) and one audio tag. Following tags have no specific order and may be any combination of audio, video, or “script” tags.

3 Typhoid Adware Implementations

In order to demonstrate and evaluate typhoid adware, we have developed three proof-of-concept implementations that are able to modify real world web content.

Our work was divided into two main tasks: first, intercepting and hijacking the connection and, second, the actual modification of the content. As all targeted communications happen via TCP/IP, we used ARP spoofing to intercept and hijack connections – this was done using the *arpspoof* program from the package *dsniff* [18]. Following successful ARP spoofing, our malicious host was pretending to be the gateway.

Our focus is a web application, so we used a simple implementation of a HTTP proxy written in Python, TinyProxy [8], where we introduced the content modification feature of typhoid adware. We configured our malicious host as a proxy by redirecting all the HTTP traffic (traffic using port 80/TCP) to a local port, used by our modified proxy server. This redirection was done by defining rules for Network Address Translation (NAT) in our malicious node, easily implemented using Netfilter [21] and the *iptables* package on Linux kernels 2.4 and 2.6.

We then implemented three different types of content modification, for HTML, video, and streaming video. These are described in the remainder of this section.

3.1 Modifying HTML

Our first and most naïve implementation was simple HTML modification. As a web page is basically a text file, inserting or substituting content is an elementary task, but it proved to be useful for our first tests.

For this HTML content modification we have tested the following approaches:

1. Inserting HTML code: general HTML code was inserted in pre-defined positions of the file (for instance, after the `<BODY>` tag). Those could be `<DIV>` tags, that allow the use of layers, so we could have banners over the original text or images.

2. String substitution: this varies from substituting specific words for others of opposite meaning (e.g., "must" for "should not"), which is amusing but not particularly useful, to substituting URLs or images for others that could show advertisements or send the user to some company website.

3. Inserting JavaScript code, usually in the `<HEAD>` section. This allows us to insert popup windows or "floating" text and banners.

Some webservers have a feature to save bandwidth that compresses HTML files before sending them, usually using GZip format and easily identified by the HTTP Header "Content-type." In order to modify such type of content, we had to decompress the file, modify it, compress it again and only then send it to the client. As HTML files are usually small (a few kilobytes), caching them before sending was not considered an issue.

3.2 Modifying Video

The second implementation used our typhoid adware proxy to cache *all* the FLV video content requested by a victim, saving it in a file and modifying the video in its entirety before sending it to the victim. The video modifications were done using FFmpeg [22], well-known open source software that is able to work with several video and audio formats.

Our goal was inserting a picture or text (our advertisement) into the video, and to do so, we used a feature of FFmpeg called *vhook*. *Vhook* is currently deprecated and was even removed from earlier versions of FFmpeg, but it served well for the purposes of a proof-of-concept. We also needed to compile FFmpeg with some features that are not in the default installation:

- *libx264:* a free implementation of a H.264 encoder. We have observed that this video format is prevalent among the videos available on YouTube. Even though it was not a requirement for our proof-of-concept, we decided to include this codec because: the modified video using the default options of FFmpeg was sometimes more than five times the original size;[2] the time of decoding/encoding was improved, as using the same codec was more efficient.

- *libmp3lame* and *libfaac*: free implementations of MP3 and AAC codecs, respectively. Included in our compiled version of FFmpeg for the same reasons as above.

The strategy of caching the whole video in order to modify it and only then sending it to the victim was a relatively easy task, but it would be only useful for small videos. As the videos got longer, so did the time to cache it and the user, our victim, would tend to give up on that particular video. Consequently all of the typhoid adware's processing time would be wasted, as well as the advertisement. Thus, the next step was making the streaming video content modification on the fly.

3.3 Modifying Streaming Video

In order to dynamically modify the video, we had to implement a cache system to address these challenges:

- The packets sent to the browser (that are intercepted by our proxy) are small enough not to have an entire FLV tag in them and we need to guarantee that the FLV tag is complete in order to make the modification.

- Not only must an FLV tag be complete, but we found that it is necessary that the part of the file we are modifying must start with a tag marked as a keyframe, otherwise the image cannot be recovered.

[2]As we show later, using this codec, the size of the resulting file was from 6% to 130% bigger than the original one, depending on the video converted.

- Our proof-of-concept first saves the data in a file and then calls FFmpeg. This involves lots of I/O operations and system calls that are both time-consuming and computationally expensive, so caching gives us control that allows us to improve performance.

Our proxy is meant to modify videos that are distributed in a FLV format, which we can identify by checking the header of the HTTP response for "Content-Type: video/x-flv." Once our proxy identifies when a video is being transmitted, it passes it to our code for inserting the advertisement into the video, otherwise it just forwards the response directly to the user.

It is also important that, when the content is a video, it modifies the "Content-Length" field in the HTTP header. We cannot predict the final size of the video, because we are modifying it on the fly. However, according to our observations, the final video size ranges from 6% to 130% bigger than the original size. If we do not change the Content-Length, the browser will close the connection after the specified number of bytes are received and the user will not receive the full video; it will "freeze" somewhere in the middle. We have observed that if the given length is *bigger* than the actual video, the transmission still works fine, so we defined the Content-Length as three times the original size. In this case, the connection is closed by the proxy, but the user still receives the whole video and no error message is provided.

We developed code in Python that was responsible for caching according to a predetermined cache size, controlling when FLV tags were completely received, and returning modified content to the proxy. The algorithm for this code is shown in Figure 3. This `read` function is called by the proxy every time some content is received, and the data returned from the function is sent to the client.

4 Experiments

Our experiments have been conducted using the third proof-of-concept, modification of streaming video. Of the three implementations, streaming video modification is the most difficult and the most compute intensive; if it is feasible, then the typhoid adware model is feasible. Modifying streaming video is also arguably the most interesting from an adware creator's point of view.

4.1 Experimental Setup

For our experiments we used two laptops running Linux. In our victim computer,[3] we have used the default web browser (Firefox) and additionally we have installed tcpdump, in order to gather data for experiments. Our typhoid adware computer[4] required Python v.2.6.2 and FFmpeg. We have used release 17758 of FFmpeg from its SVN repository because this is the last version with

[3] 1.5 GHz Intel Celeron, 2 Gb RAM, Ubuntu Linux 8.04, 100 Mbps Ethernet, 802.11g wireless.

[4] 1.8 GHz Intel Core2 Duo, 3 Gb RAM, Ubuntu Linux 9.04, 100 Mbps Ethernet, 802.11b/g wireless.

```
// Constants
N = approximate number of seconds
CACHE_SIZE = N * 32 Kb

// Global variables
tags: list of FLV_tags
buffer: array of bytes
last_keyframe: integer

function read(data: array of bytes)
    output: array of bytes

    buffer = buffer + data
    (buffer, tags) = split_tags(buffer, tags)

    if length(tags not sent) ≥ CACHE_SIZE
        last_keyframe = last tag that was not sent
                        and is marked as a keyframe
    endif

    tags_to_send = tags until the last_keyframe-1
    remove tags_to_send from tags
    save tags_to_send into temp_file

    call FFmpeg(temp_file, advertisement),
         saving to output_file

    read output_file into output
    remove header from output
    return output
```

Figure 3: Function for modifying streaming video

Table 1: Time (seconds) to send the video – wired

Cache Size	Mean	Std. Dev	Median
0 Kb (proxy only)	7.97	3.14	8.69
32 Kb	16.56	0.90	16.71
64 Kb	16.41	0.53	16.43
160 Kb	15.88	0.34	15.90
320 Kb	16.56	0.45	16.61

support for vhook. It was compiled with options –enable-libx264, –enable-gpl, –enable-libmp3lame, and –enable-libfaac.

Our wired environment had those laptops linked to an in-lab 10/100 Mbps hub connected in turn to a 10/100 Mbps switch and eventually to the Internet. In our wireless environment the laptops were connected to a wireless access point (802.11g, 100 Mbps Ethernet) that works also as a router, connected to the Internet by a cable connection.

As the streaming video source we have chosen YouTube, one of the most popular video streaming providers, thus a website with high probability of being accessed by a user. To show a video, YouTube uses a Flash player embedded in the webpage which makes a request for the video that is distributed in FLV format.

4.2 Typhoid Adware Processing Time

In order to evaluate the impact on the typhoid adware computer, the machine that executes the attack, we measured the time to send the video content. For reference, the YouTube video we used was 46 seconds and 1,853,392 bytes long (in unmodified form).

First, we ran the typhoid adware just as a proxy, i.e., without modifying the content, then we modified the content by caching in 32 Kb, 64 Kb, 160 Kb, and 320 Kb chunks. Those cache sizes were chosen because we have observed that in a FLV file, one second is approximately 32 Kb, so we were caching around 1, 2, 5 and 10 seconds of video respectively. Each test was repeated 10 times to compensate for network timing variations.

The times were calculated from the moment the typhoid adware proxy receives the first packet of video content until it sends the last packet.

Table 1 shows the results in a wired environment and Table 2 shows the results in a wireless environment. We ran tests in both environments to evaluate how the media would impact on the times.

We observed that, in the wireless environment, the time to send the file is a little bit longer, even when we are not caching and just forwarding content. It shows that we do have a slight delay in a wireless environment, but that this does not affect the performance of the typhoid adware.

Table 2: Time (seconds) to send the video – wireless

Cache Size	Mean	Std. Dev	Median
0 Kb (proxy only)	10.03	0.72	9.97
32 Kb	17.50	2.19	17.08
64 Kb	16.56	1.50	16.14
160 Kb	18.08	2.24	17.53
320 Kb	19.13	1.99	19.26

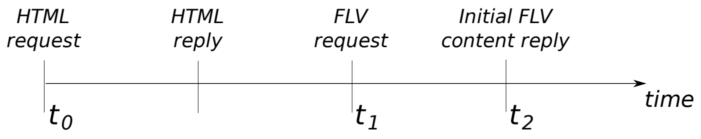

Figure 4: Timeline of network events

4.3 Latency

Using the same tests, we measured the latency on the client side, so we could evaluate how the delay of having the content changed impacted the user, who is expecting to see the video start in a reasonable time.

In order to calculate the time, we used *tcpdump* on the victim computer to record the network traffic during the aforementioned tests. We used three timing points, illustrated in Figure 4:

1. t_0: the time when the HTML request for the web page that contains the video is sent

2. t_1: the time when the HTML request for the actual video is sent (FLV request)

3. t_2: time when the client starts receiving FLV content

For this evaluation, we added one extra set of tests (also run 10 times in both wired and wireless environments), for video requests when typhoid adware was not involved at all, not even as a regular proxy, to determine a baseline for the times. Tables 3 and 4 show the results for a wired and a wireless environment, respectively.

Again, as expected, the environment impacts the time to start receiving content. But the times for both environments are similar. Also, the times reflect the cache size: bigger caches increase the delay. Especially for smaller cache sizes, the latency introduced by typhoid adware is slight from the user's point of view; they would be unlikely to notice the extra delay.

4.4 File size

We have also measured the size of the final converted video file when the file is converted as a whole, and also when we convert parts of the file, as happens

Table 3: Time (seconds) to receive video – wired

Cache Size	$t_1 - t_0$		$t_2 - t_1$	
	Mean	Std. Dev	Mean	Std. Dev
Direct (no proxy)	2.37	0.13	0.06	0.02
0 Kb (proxy only)	2.60	0.13	0.15	0.05
32 Kb	2.69	0.15	1.06	0.12
64 Kb	2.65	0.20	1.01	0.04
160 Kb	2.54	0.12	2.51	0.11
320 Kb	2.69	0.14	3.45	0.10

Table 4: Time (seconds) to receive video – wireless

Cache Size	$t_1 - t_0$		$t_2 - t_1$	
	Mean	Std. Dev	Mean	Std. Dev
Direct (no proxy)	3.23	1.10	0.08	0.00
0 Kb (proxy only)	3.18	0.14	0.30	0.04
32 Kb	3.86	1.15	1.36	0.02
64 Kb	3.30	0.10	1.44	0.11
160 Kb	4.17	1.13	2.94	0.05
320 Kb	4.89	1.68	3.91	0.03

when we do the on-the-fly modification.

We wanted to know how the file size would increase for three reasons. First, it can be used as a parameter to how much we are going to increase the Content-Length field in the HTTP header. Second, we would like to know if the cache size influences the final file size. Third, it would show what we might lose in terms of bandwidth during an attack of typhoid adware.

Using the default options of FFmpeg included in the standard Linux distribution gave us a final file size that was sometimes five times bigger than the original one. However, when FFmpeg was compiled with support for H.264, MP3, and AAC (the first a video codec and the other two audio codecs), we found that the final file was only from 6% to 130% bigger than the original file. A previous study [2] showed that YouTube content is biased towards short videos, meaning that even a 130% increase will not have a tremendous impact.

Also, we noticed that the cache size has little influence on the final size of the file. We conjecture that the final file size is more a consequence of the compression rate for each file. Figure 5 shows the how the file size increased with the cache size for seven videos of varying initial sizes ("Video1" is the video used for the other tests in this section).

4.5 Glitches

One other observation was the occurrence of "glitches." When caching less than 64 Kb (approximately two seconds) we could observe sound and image glitches

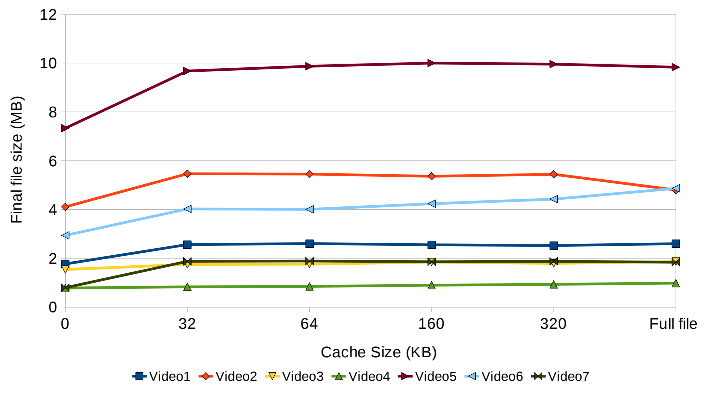

Figure 5: File size after video modification

during the whole video, which could cause a user to become suspicious that the content was being modified. However, as more was cached, fewer glitches were noticed.

In order to choose a cache size that causes the least impact to the client, we need to consider both the latency and the amount of glitches. Around 5 seconds of caching (160 Kb) proved to be a good balance in our experiments.

4.6 Is there Life Outside YouTube?

While we have run our tests accessing YouTube video, we have also tested if our typhoid adware could intercept and modify content from other sources. Although space limitations prevent us from giving full details, typhoid adware successfully intercepted and modified content from different sources such as the Brazilian website UOL, the Canadian CBC, and videos from CNN. Clearly our video modification approach generalizes to other video sources.

5 Defenses

Some measures can be implemented to prevent – or at least minimize – the risk of typhoid adware.

5.1 ARP Spoofing

As ARP spoofing is a well known problem, there are already several proposed solutions to it, like heuristic analysis of ARP packets [5]. Static IP-to-MAC mapping tables would avoid the ARP spoofing problem completely by not needing ARP at all, but *only* relying on hardcoded mappings is clearly infeasible for any kind of dynamic environment; most users would require at least a partially dynamic solution, if only to go from one Internet café to the next.

One could also argue that the simple adoption of IPv6 would solve this problem, as ARP is not used in IPv6. However, it has the Neighbor Discovery Protocol that is similar to ARP and is also vulnerable to traffic redirection attacks, as mentioned in its specification [13].

The general idea is that, in order to avoid ARP spoofing, the best solution is still detection and eventual removal of malicious nodes from the network, but this is usually implemented by the network administrator and, in case of poor or overworked administration, users would be vulnerable. However, many current anti-virus packages come with firewall software, and we think that ARP spoofing detection is a feature that could reasonably be included into such tools.

In the short term, we revisit static IP-to-MAC mapping tables. This idea is usually discarded under claims that it is not feasible in large networks, demanding a lot of work from the network administration staff in order to maintain those tables and is highly susceptible to errors. However, recall our original scenario of an Internet café: we have a simple network topology and low (no?) administration, with IP addresses being assigned by a DHCP server, where the DHCP server may be a feature supplied by the access point itself.

We propose the introduction of an "Internet Café" setting for network configuration. The DHCP protocol specifies that, once an IP address is assigned, the DHCP server sends an Acknowledge message, which may contain the router (or default gateway) information for the client's network, more likely the actual access point's address. Using that information, our special setting would gather the MAC address of that router and automatically set it in the static IP-to-MAC mapping table at the client's machine. By doing this, even if a malicious node is able to send fake ARP messages to the router, the ARP spoofing process would fail as the potential victim would not accept the malicious MAC address as being the router's.

As a proof-of-concept, we have implemented a Internet Café setting for the Linux environment (Ubuntu 9.04) consisting of some shell and Python scripts that are executed during the "pre-up" and "up" events that occur during the process of activating a network interface. Our first idea was that we would intercept the DHCP traffic just before the interface was up and, after that, we would read the messages and gather the information we needed.[5]

We have this implementation successfully working in an environment with a combined access point/router/DHCP server, and it proved effective against ARP spoofing attacks (and thus typhoid adware).

However, we realized that another implementation was also possible. A DHCP client, or any other method of setting the IP configuration, will configure the routing information for an interface as soon as the interface is up. We just need to gather the MAC address for that router and set it statically on the MAC-to-IP mapping table. This approach also proved effective against ARP spoofing attacks in our tests, and it was even easier to implement and deploy.

[5]This implementation required a bug fix to the "Network Manager" component of our Linux distribution. This component is responsible for bringing interfaces up and down, and calling the required scripts for configuration. Prior to the bug fix, it was not triggering the "pre-up" event like it was supposed to.

Unlike the first implementation, this approach worked in different environments, both a combined access point/router/DHCP server as well as an environment where the DHCP server was located on a different machine than the router.

Of course, one can argue that this idea could be easily broken by introducing a malicious DHCP server in the network, which would give fake routing information to the client and allow the interception and modification of packets. This is another known technique for man-in-the-middle attacks and it could also be used by typhoid adware. There are several approaches to detect rogue DHCP servers in a network, like exchanging DHCP Inform messages to authenticate the valid ones [12], but this would rely on extra network administration tasks. Another simple technique is sending a DHCP query – if we receive different responses then it would be a good indication that there is a rogue DHCP server in the network. While a network administrator could use this technique for detection, an Internet Café setting could use this information to inform the user that that network is compromised.

5.2 Content Modification

In order to avoid content modification, we suggest some strategies to be implemented both in the video file and in the video player. We note that other formats, like Matroska [11], have support for these types of strategy, so they are clearly implementable.

5.2.1 Encryption

Sending the video and audio content encrypted, using SSL, for instance, would increase the difficulty of modifying the content, however this would decrease performance.

5.2.2 Checksum List

The very first part of a FLV file is a metadata section that can include almost any kind of information. Thus, this could be used to store an array of checksums, calculated for each FLV tag that contains video or audio data. The FLV player would then be able to verify the checksums as content is received.

This strategy is a suitable defense against "on-the-fly" modifications, but if the video is cached as a whole, the checksums can be recalculated by typhoid adware.

5.2.3 Signed Checksum List

An extension of the previous idea is signing the FLV file, also using the metadata section of the FLV file. The content producer could include a digitally signed checksum list that would be tested by the FLV player, in order to detect whole-video modification where the checksums had been updated correctly.

5.3 Timing Anomalies

One approach would be to try and detect timing differences on the user machine that may indicate the presence of typhoid adware. However, given that our experimental timings did not reveal a delay that could not naturally occur in the network, we are somewhat skeptical that this defense would work.

6 Related Work

There are several loosely-related types of attack. Content modification, or content pollution, has been studied in peer-to-peer environments, e.g., [4]. Proximity-based worm attacks via Bluetooth have also been studied [20].

In general security terms, typhoid adware performs a man-in-the-middle attack, although we are not aware of one being applied to adware in the way we describe. Closely related to our first implementation is [25], although again the adware potential is unexplored. Also related is work on hijacking wireless connections [9], but their focus is mostly on content injection into a victim's browser cache, a departure from the typhoid adware model that leaves no trace. However, their hijacking technique could be used to implement typhoid adware over a wireless network.

Patents exist that cover legitimate, targeted insertion of advertisements into various media including video (e.g., [3]) but these systems may preprocess the videos using lots of computing power, and of course need not redirect connections.

Finally, the problem of determining whether HTML content has been modified *en route* has been considered by [16]. Their detection method, like some we suggest in Section 5, relies on additions to the content that the client uses to verify it.

7 Future Work and Conclusions

While there are further enhancements that could be made to typhoid adware, such as handling file formats other than FLV, our proof-of-concept implementations have sufficiently demonstrated the idea, and further research will be directed towards defenses.

In this paper we have presented typhoid adware, a new approach to spreading advertisements. The technique is more covert than current adware, because the computer containing adware shows no advertisements to reveal its presence, and computers that do see advertisements contain no adware to detect.

Typhoid adware can be implemented using well known techniques such as ARP spoofing and proxies, and leveraging already-existing tools. It was successfully demonstrated in both wired and wireless networks, modifying a variety of content including streaming video. Even in the most overhead-intensive case, streaming video, the victim still receives the content in a reasonable time. In terms of defense, we have implemented two approaches and suggested some other, content-based ones. Typhoid adware is a viable future threat, especially for network environments that are not well monitored, like the increasingly ubiquitous Internet café.

Acknowledgments

The third author's work is supported in part by a grant from the Natural Sciences and Engineering Research Council of Canada. Thanks to Anil Somayaji for discussion about timing anomalies.

References

[1] Adobe Systems Incorporated. *Video File Format Specification Version 10*, November 2008.

[2] X. Cheng, C. Dale, and J. Liu. Statistics and social network of YouTube videos. In *16th International Workshop on Quality of Service*, pages 229–238, 2008.

[3] H. V. Cottingham. Internet service provider advertising system. United States Patent #6,339,761, 15 January 2002.

[4] P. Dhungel, X. Hei, K. W. Ross, and N. Saxena. The pollution attack in P2P live video streaming: Measurement results and defenses. In *2007 Workshop on Peer-to-peer Streaming and IP-TV*, pages 323–328, 2007.

[5] A. Di Pasquale. ArpON. Last retrieved 8 June 2009.

[6] J. Erickson. *Hacking: The Art of Exploitation.* No Starch Press, 2003.

[7] Federal Trade Commission. Monitoring software on your PC: Spyware, adware, and other software. Staff report, 2005.

[8] S. Hisao. Tiny proxy, November 2006. version 0.2.1.

[9] M. Kershaw. Wireless security isn't dead, attacking clients with MSF. Black Hat DC, 2010.

[10] J. W. Leavitt. *Typhoid Mary: Captive to the Public's Health.* Beacon Press, 1996.

[11] Matroška. Last retrieved 10 June 2009.

[12] Microsoft TechNet. Preventing rogue DHCP servers. Last retrieved 4 September 2009.

[13] T. Narten, E. Nordmark, W. Simpson, and H. Soliman. Neighbor Discovery for IP version 6 (IPv6). RFC 4861, 2007.

[14] D. Plummer. Ethernet Address Resolution Protocol. RFC 826, 1982.

[15] J. Purisma. To do or not to do: Anti-virus accessories. In *Virus Bulletin Conference*, pages 125–130, 2003.

[16] C. Reis, S. D. Gribble, T. Kohno, and N. C. Weaver. Detecting in-flight page changes with web tripwires. In *5th USENIX Symposium on Networked Systems Design and Implementation*, pages 31–44, 2008.

[17] T. Socolofsky and C. Kale. A TCP/IP tutorial. RFC 1180, 1991.

[18] D. Song. dsniff. version 2.4.

[19] Sophos. Potentially unwanted application (PUA). Glossary of terms. Last retrieved 4 June 2009.

[20] J. Su, K. K. W. Chan, A. G. Miklas, K. Po, A. Akhavan, S. Sariou, E. de Lara, and A. Goel. A preliminary investigation of worm infections in a Bluetooth environment. In *4th ACM Workshop on Recurring Malcode*, pages 9–16, 2006.

[21] The netfilter.org Project. Netfilter/iptables.

[22] S. Tomar. Converting video formats with FFmpeg. *Linux J.*, 2006(146):10, 2006.

[23] P. Wald. *Contagious: Cultures, Carriers, and the Outbreak Narrative.* Duke University Press, 2008.

[24] S. Young and D. Aitel. *The Hacker's Handbook.* Auerbach, 2004.

[25] B. Zdrnja. Malicious JavaScript insertion through ARP poisoning attacks. *IEEE Security and Privacy*, 7(3):72–74, 2009.