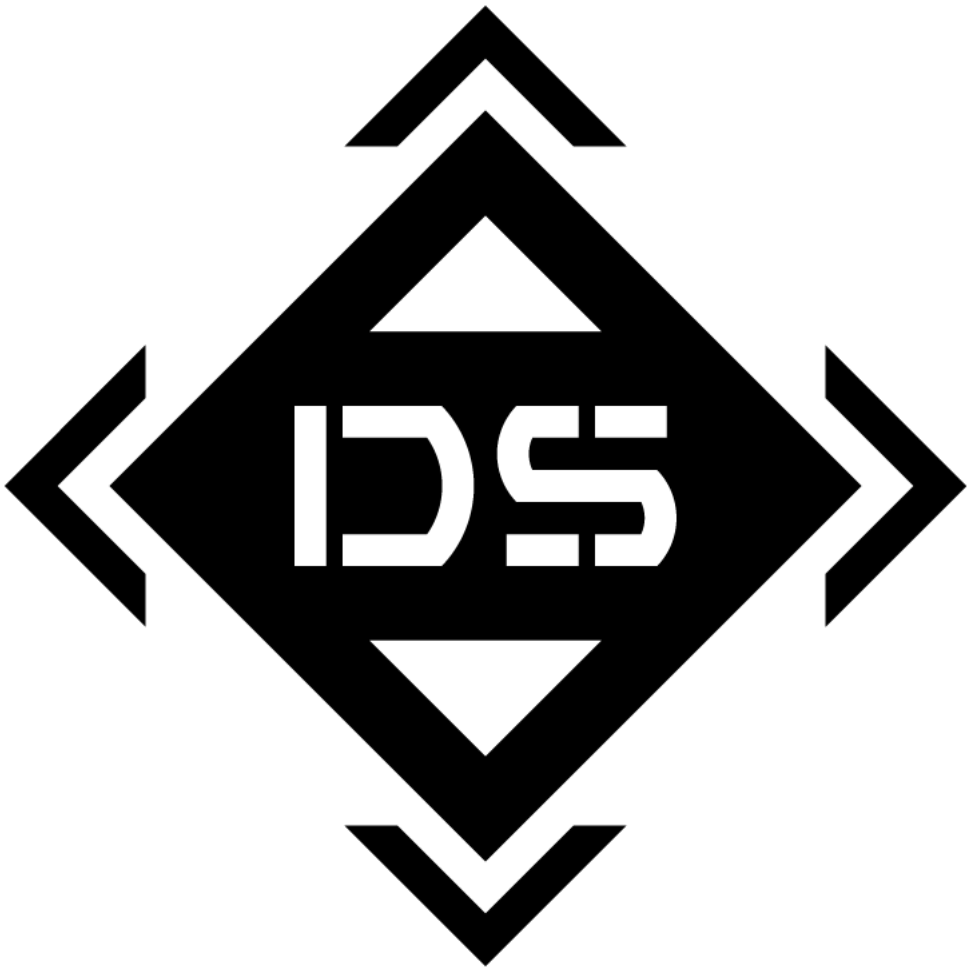


DIGITAL SILENCE

Whitepaper: Bypassing Port Security In 2018 – Defeating MACsec and 802.1x-2010

DEF CON 26 Pre-Release Version

Gabriel Ryan (@s0lst1c3)

August 2018

Last Modified: 7/20/18 6:44:00 AM

Disclaimer and Updates

This whitepaper is an early version designed for pre-release prior to DEF CON 26. All content will be updated by the time of the presentation at DEF CON 26 in August 2018. Final versions of all content will be available at:

- https://digitalsilence.com/blog/

Author Contact Information

Author:	Gabriel Ryan, Co-Founder / Senior Security Assessment Manager Digital Silence Email: gabriel@digitalsilence.com Twitter: @s0lst1c3 LinkedIn: linkedin.com/in/ms08067/ Github: github.com/s0lst1c3 Phone: +1-410-920-8101 SendSafely Secure Link: https://digitalsilence.sendsafely.com/u/gabriel Public Key Fingerprint: 5017 5C8B C1A7 31A0 80E4 9110 17C2 6CC7 BA33 3B45

Abstract

Existing techniques for bypassing wired port security are limited to attacking 802.1x-2004, which does not provide encryption or the ability to perform authentication on a packet-by-packet basis [3][4][6]. The development of 802.1x-2010 mitigates these issues by using MACsec to provide Layer 2 encryption and packet integrity checks to the protocol [7]. Since MACsec encrypts data on a hop-by-hop basis, it successfully protects against the hub, bridge, and injection-based attacks pioneered by Steve Riley, Abb, and Alva Duckwall [7][8].

In addition to the development of 802.1x-2010, improved 802.1x support by peripheral devices such as printers also poses a challenge to attackers. Gone are the days in which bypassing 802.1x was as simple as finding a printer and spoofing a MAC address – hardware manufacturers have gotten smarter.

In this paper, we introduce the Rogue Gateway and Bait n Switch attacks, which together can be used to bypass 802.1x-2010 and MACsec when weak EAP methods are used. Additionally, we introduce the EAP-MD5 Forced Reauthentication attack exploiting a weakness in the initiation of EAP authentication. We discuss how improved 802.1x support by peripheral devices does not necessarily translate to improved port-security due to the widespread use of weak EAP. Finally, we consider how improvements to the Linux kernel ease implementation of bridge-based techniques and demonstrate an alternative to using packet injection and manipulation for network interaction.

We packaged each of these techniques and improvements into an open source tool called silentbridge, which we plan on releasing alongside this paper.

Table of Contents

I. Introduction

In this paper, we provide a brief history of attacks against the 802.1x protocol, as well as descriptions of how the 802.1x and EAP protocols work. We also describe some of the most commonly used EAP methods, highlighting any security issues. We also discuss the historical use of port security exceptions as an attack vector, noting that improved 802.1x support by peripheral devices is changing this. Finally, we discuss our improvements to the bridge-based 802.1x bypass technique introduced by Alva Duckwall [4], along with three attacks that can be used against 802.1x-2004 and 802.1x-2010 using weak forms of EAP.

II. Background and Prior Work

Created in 2001, the original version of the 802.1x standard was designed to provide a rudimentary authentication mechanism for devices connecting to a local area network (LAN) [1]. Three years later, an extension of 802.1x named 802.1x-2004 was released to facilitate the use of 802.1x on wireless networks [2].

In 2005, researcher Steve Riley discovered that 802.1x-2004 could be bypassed by inserting a hub between an authorized device and a switch [3]. The attacker could then attach a rogue device to the hub and sniff packets and inject UDP traffic onto the network. Injecting TCP traffic was not possible due to a race condition that resulted in dropped packets and possible detection [4].

In 2011, a researcher named "Abb" published a tool called Marvin that could be used to bypass 802.1x by introducing a rogue device configured as a bridge directly between an authorized device and the switch [5]. This allowed an attacker to eavesdrop on network traffic without the use of a hub. Later that year, researcher Alva Duckwall improved upon Abb's attack, using source NATing to achieve full network interaction without relying on packet injection [4]. In 2017, Valérian Legrand released a similar tool that featured a modular design written in Python [6].

II.1 MAC Filtering and MAC Authorization Bypass (MAB)

When enterprise organizations using 802.1x need to deploy a device that does not support the protocol, they must either permanently or temporarily disable 802.1x on the port used by the device. Disabling 802.1x on a port and replacing it with a weaker form of access control, such as MAC filtering, introduces a “port-security policy exception.”

Historically, these policy exceptions were prevalent due to widespread lack of 802.1x support by peripheral devices such multifunction printers and IP cameras. Consequently, attackers typically first looked for policy exceptions when attempting to bypass port security, particularly since the bridge-based techniques described in II. Background and Prior Work and IV. Improvements to Classical Bridge-based 802.1x Bypass required considerably more effort. To bypass MAC filtering, the attacker merely located a device that does not use 802.1x, spoofed its MAC address, and connected to the device’s switch port.

II.2 The Current State of Wired Port Security

The largest enterprise networking hardware manufacturers now offer switches that supports 802.1x-2010. This new version of the 802.1x protocol uses MACsec to implement hop-by-hop Layer 2 encryption along with packet-by-packet integrity checks. These additional security features defeat the bridge-based attacks we described in II. Background and Prior Work and IV. Improvements to Classical Bridge-based 802.1x Bypass [7]. However, adoption rates for 802.1x itself remain relatively low, and adoption rates for newer additions to the standard such as 802.1x-2010 are even lower. Regardless, attackers should expect to see increased 802.1x-2010 adoption in the near future, giving rise to a need to develop a method to cope with it.

In addition to the development of 802.1x-2010 and MACsec, improvements in peripheral device security increase the challenges in bypassing wired port security by looking for policy exceptions. At this point, most printer manufacturers offer at least one affordable model supporting 802.1x. As enterprise organizations continue to phase out legacy hardware, they continue to deploy more 802.1x capable peripheral devices into their network environments. This, in turn, decreases the frequency of port security exceptions, reducing the existence of what was once considered low-hanging-fruit for attackers.

In the remainder of this paper, we demonstrate our efforts to address both the introduction of MACsec and increased 802.1x support by peripheral devices. We begin by introducing improvements to Duckwall's bridge-based attacks against 802.1x-2004. We then introduce techniques to bypass 802.1x-2010 when implemented using weak forms of EAP. Finally, we discuss strategies and techniques to compensate for improvements in peripheral devices.

II.3 802.1x Overview

The 802.1x protocol is an authentication framework used to allow or deny access to devices wishing to connect to a local area network (LAN) (either wired or wireless) [1][2][9]. The protocol defines an exchange between the following three parties:

- **supplicant** – the client device wishing to connect the LAN [1][2][9].
- **authenticator** – a network device such as a switch providing access to the LAN [1][2][9].
- **authentication server** – a host that runs software implementing RADIUS or some other Authorization, Authentication, and Accounting (AAA) protocol. Usually the authentication server is a standalone system, although it can be built into the same hardware as the authenticator [1][2][9].

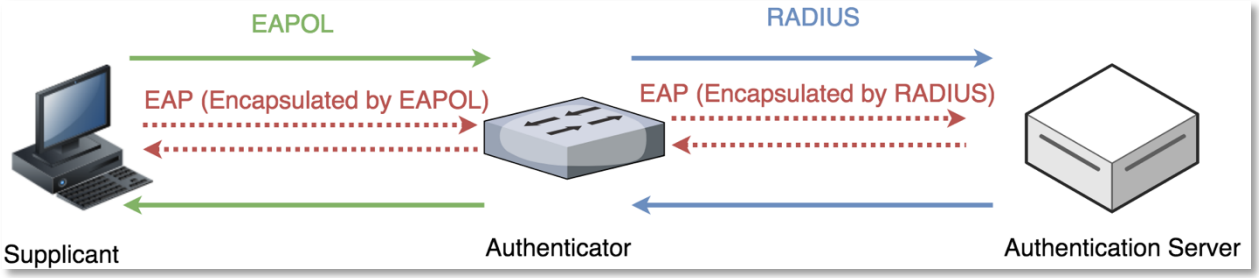


Figure 1 – The EAP authentication process is encapsulated by EAPOL between the supplicant and authenticator, and by RADIUS between the authenticator and authentication server.

The authenticator can be thought of as a gatekeeper that guards access to the LAN. When the supplicant connects to a switch port, it must provide the authenticator with a set of credentials [1][2][9]. The authenticator forwards these credentials to the authentication server, which verifies that the credentials are valid. If the credentials are valid, the authentication server instructs the authenticator to allow the supplicant to access the network. Otherwise, the supplicant is denied access to the network [1][2][9].

The 802.1x authentication process typically follows a four-step sequence:

1. **Initialization** – the supplicant connects to a port on the switch (authenticator). At this time, the switch port is currently disabled. The authenticator detects this new connection and enables the port, but only allows 802.1x traffic to be transmitted. When in this restricted state, the port is "unauthorized" [1][2][9].

2. **Initiation** – either the supplicant or the authenticator can initiate the 802.1x authentication process. In some implementations of 802.1x, the authenticator periodically sends out EAP-Request-Identity frames that prompt the supplicant to begin authenticating [1][2][9]. Alternatively, the authenticator can wait for the supplicant to send an EAPOL-Start frame, to which it will respond with an EAP-Request-Identity frame. In either case, the supplicant replies with an EAP-Response-Identity frame containing an identifier (such as a username). The supplicant receives this frame, encapsulates it in a RADIUS Access-Request frame, and forwards the frame to the authentication server [1][2][9].

3. **EAP Negotiation** – The authentication server responds with an EAP-Request frame encapsulated within a RADIUS Access-Challenge. The authenticator strips the RADIUS Access-Challenge frame from this response, and sends the resulting EAP-Request frame to the supplicant [1][2][9]. The EAP-Request frame specifies an EAP method that the supplicant should use to continue the authentication process. The supplicant either begins the EAP authentication process using the recommended EAP method, or responds with a Negative Acknowledgement (NAK) that includes a list of acceptable methods. [1][2][9]

4. **Authentication** – Once the supplicant and authentication server agree on an EAP method, the authentication process begins. The specific details of how the authentication process should proceed depends on the EAP method selected [1][2][9]. No matter what EAP method is used, the authentication process will result in an EAP-Success or EAP-Failure message. In the event of a successful authentication, the port is set to an "authorized state", in which normal traffic is allowed. Otherwise, the port remains in an "unauthorized" state [1][2][9].

II.4 Notable EAP Methods

There are many ways to implement EAP [16]. These different EAP implementations are known as EAP methods [16]. In this section, we review some of the most commonly used EAP methods direct relevant to the material covered in this paper.

II.4.A EAP-MD5

The EAP-MD5 authentication process begins when the authentication server sends an EAP-Request-Identity to the supplicant [16]. The supplicant responds with an EAP-Response-Identity, causing the authentication server to create a randomly generated challenge string. The authentication server sends this challenge string to the supplicant as an MD5-Challenge-Request [16]. The supplicant then concatenates its username, plaintext password, and the challenge string into a single value, and sends the MD5 hash of this value to the authentication server as the MD5-Challenge-Response. Upon receiving the MD5-Challenge-Response, the authentication server repeats the hashing process performed by the supplicant: the authentication server concatenates the username, password, and challenge string into a single value which is input into the MD5 hashing function. This second MD5 hash (created by the authentication server) is compared against the MD5-Challenge-Response (created by the supplicant). If the two hashes are identical, the authentication attempt succeeds. Otherwise, it fails [16].

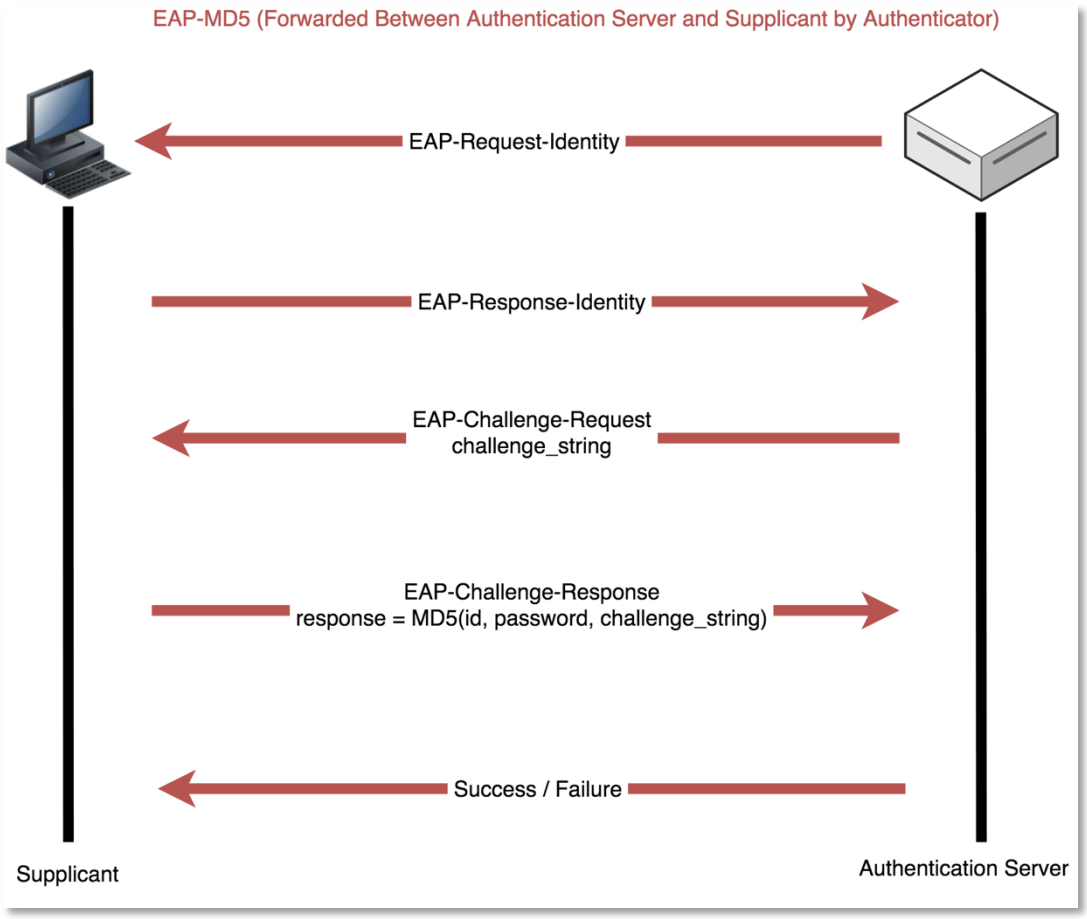


Figure 2 – the EAP-MD5 authentication process

This authentication method, when used alone, is not protected by encryption. As described by Josh Wright and Brad Antoniewicz in their presentation at Schmoocon 2008, an attacker sniffing traffic between the supplicant and the authenticator can capture both the MD5-Challenge-Request and MD5-Challenge-Response [13]. Wright and Antoniewicz describe a dictionary attack to calculate the plaintext password, as illustrated in *Figure 3* below [13].

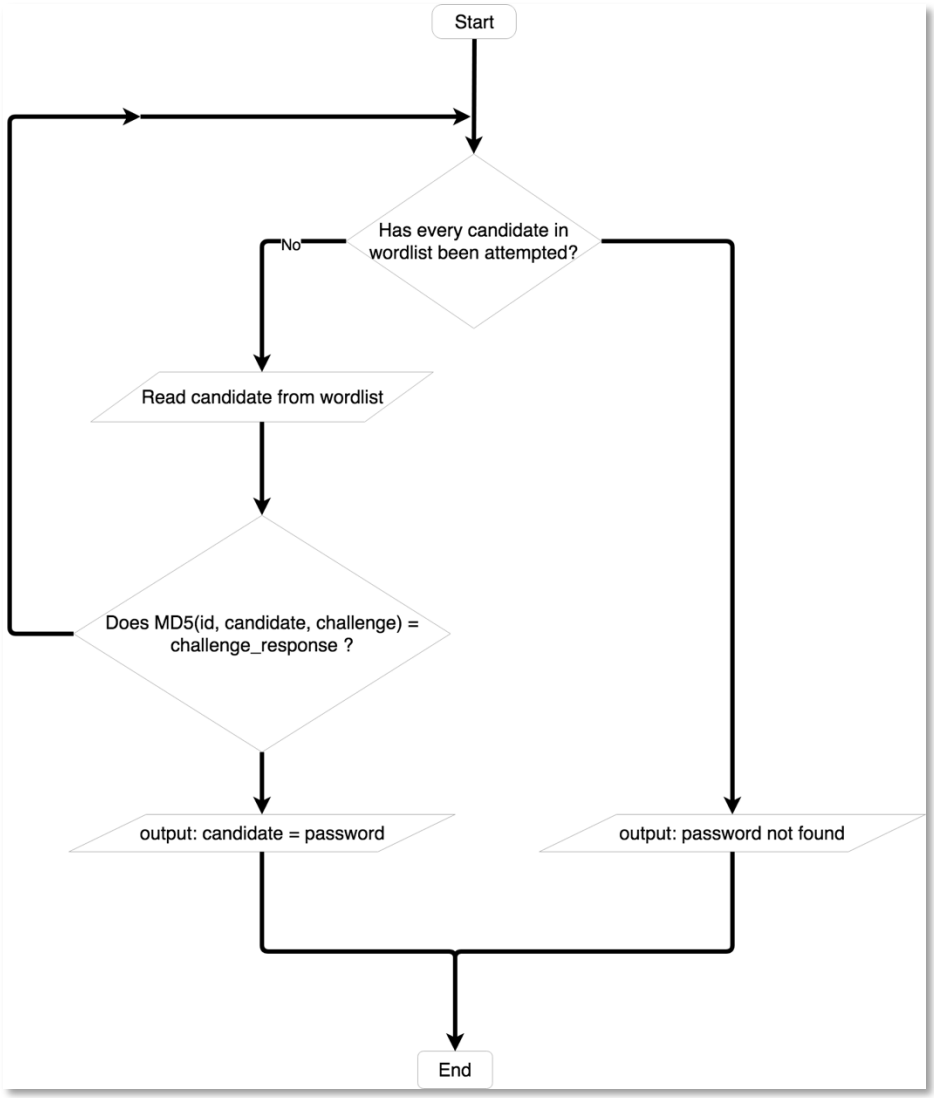


Figure 3 – Wright's and Antoniewicz's dictionary attack against EAP-MD5, expressed as an algorithmic flowchart

Further work by Fanbao Liu and Tao Xie of the National University of Defense Technology in Changsha, China reveals an even more efficient EAP-MD5 cracking technique that uses a length-recovery attack [19].

II.4.B EAP-PEAP / EAP-TTLS

The authentication process consists of two phases: outer authentication and inner authentication. Outer authentication comes first, and begins when the supplicant makes an authentication request to the authentication server via the authenticator [21][29][30]. The authenticator then attempts to prove its identity to the supplicant by responding with an x.509 certificate. If the supplicant accepts the authentication server's certificate, outer authentication succeeds and a secure tunnel is established between the authentication server and supplicant [21][29][30]. We then transition to the inner authentication process through the secure tunnel. The use of a secure tunnel to protect the inner authentication process was developed largely in response to the weaknesses that affect unprotected EAP methods such as EAP-MD5.

Much like EAP itself, there are many different protocols available for use during the inner authentication process [21][29][30]. However, MS-CHAPv2 is the most commonly used authentication protocol for this purpose.

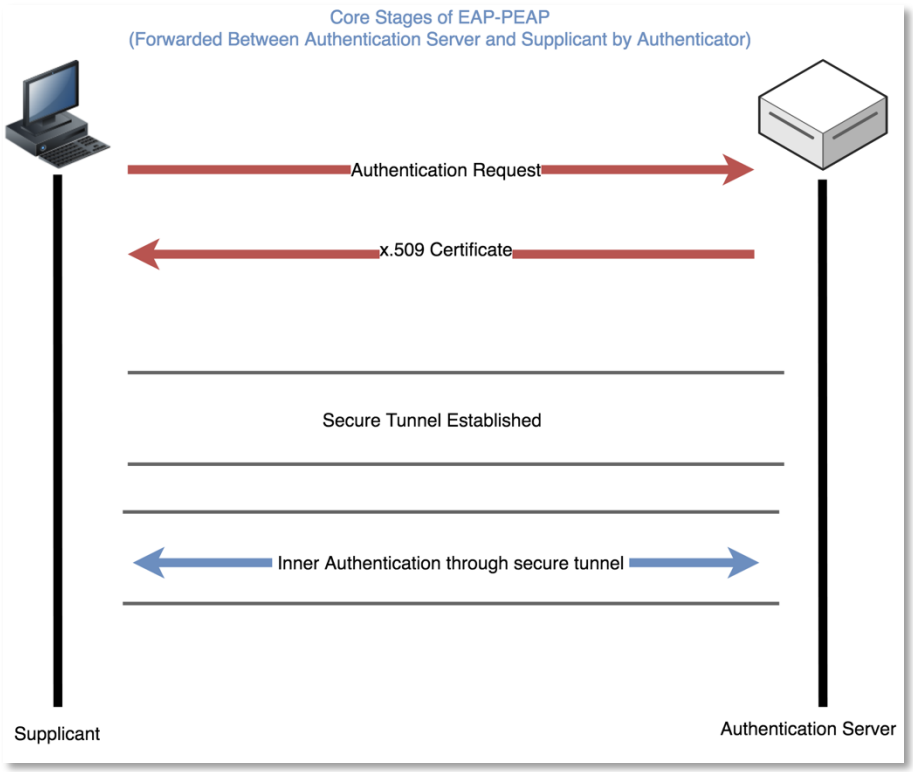


Figure 4 – The EAP-PEAP authentication process

This system is problematic. Although mutual authentication can be enforced using inner-authentication mechanisms such as MS-CHAPv2, the x.509 certificate is the only means through which the supplicant can verify the identity of the authentication server. Absent a guarantee that the supplicant will always reject invalid certificates, the onus is placed on the supplicant (and therefore the user, in many cases) to reject invalid certificates received by the authentication server [13][20].

Remember that EAP is not only used for wired authentication as specified by 802.1x, but for wireless authentication in conjunction with WPA2 [21]. It was this inability to validate the identity of the authentication server that lead to the classic attack against WPA2-EAP wireless networks presented by Brad Antoniewicz and Joshua Wright at Schmoocon in 2008 [13]. When WPA2-EAP is implemented using weak EAP methods such as EAP-PEAP and EAP-TTLS, an attacker can use a rogue access point attack to force the supplicant to authenticate with a rogue authentication server [13][21]. So long as the supplicant accepts the certificate presented by the attacker's authentication server, the supplicant will transmit an EAP challenge and response to the attacker that can be cracked to obtain a plaintext username and password [13][21].

Further increasing the severity of this issue, MS-CHAPv2 is the strongest Inner Authentication protocol available for use with EAP-PEAP and EAP-TTLS. MS-CHAPv2 itself is vulnerable to a cryptographic weakness, first discovered by Moxie Marlinspike and David Hulton in 2012, that allows an attacker to reduce the captured MS-CHAPv2 challenge and response hashes to a single round of DES encryption, which is a mere 56-bits in length [22][23]. These 56-bits are weak enough that they can converted into a password-equivalent NT hash within 24 hours with a 100% success rate using FPGA-based hardware [22][23].

Although the feasibility of similar attacks against wired port security have yet to be explored, this paper demonstrates that such attacks are pivotal in allowing us to bypass 802.1x-2010.

II.4.C EAP-TLS

In 2008, EAP-TLS was introduced by RFC 5216, largely as a mitigation to the aforementioned security issues affecting weak EAP methods such as EAP-PEAP and EAP-TTLS [24]. The strength of EAP-TLS lies in its use of mutual certificate-based authentication during the outer authentication process, preventing attackers from performing the kinds of man-in-the-middle attacks that can be used to attack weaker EAP implementations [24]. Unfortunately, the inconvenience of installing a client certificate on all supplicant devices reduced the overall adoption rate of this technology [25].

III. Research Environment and Architecture

Our lab environment consisted of the following core components:

- **Simulated Network Environment** – the test network against which we performed our attacks.
- **Rogue Device A** – a rogue device configured to use a bridged-based approach for performing 802.1x bypasses.
- **Rogue Device B** – a rogue device equipped with remotely controllable mechanical A/B ethernet splitters.

In the remainder of this paper, we often talk about attacks requiring either a Rogue Device A or Rogue Device B configuration. When we say this, we mean that the attack assumes that the rogue device is configured according to the following descriptions.

III.1 Simulated Network Environment

Our simulated network environment emulates an enterprise internal network protected by 802.1x authentication.

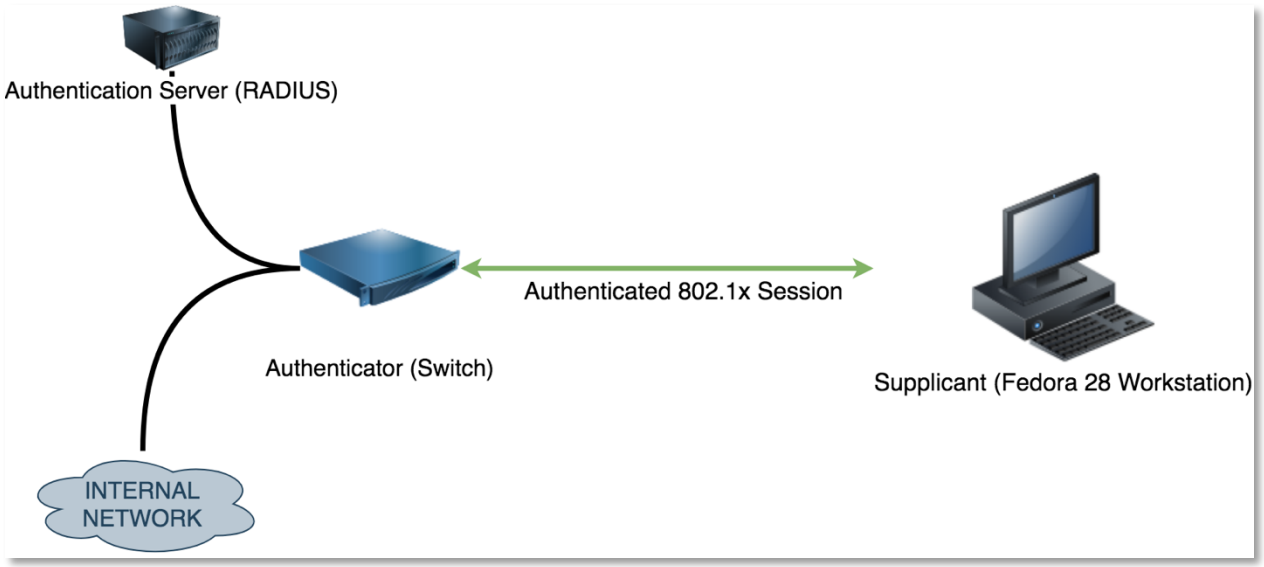


Figure 5 – the simulated network environment

As shown in *Figure 5*, the simulated network environment uses the following components:

- **Supplicant:** a MACsec capable Linux workstation running Fedora 28 equipped with NetworkManager and wpa_supplicant, configured to connect and authenticate automatically with the network.
- **Authenticator:** a MACsec capable Cisco Catalyst 3560-CX switch configured as follows:
    - *GigabitEthernet 0/1 interface* – provides an upstream link to the network gateway
    - *GitabitEthernet 0/2 interface* – provides administrative access to the switch
    - *GigabitEthernet 0/3 interface* – provides a connection to the external RADIUS server
    - *GigabitEthernet 0/5 interface* – standard 802.1x protected port
    - *GigabitEthernet 0/6 interface* – 802.1x protected port with MACsec
- **Authentication Server (RADIUS)** – we used a Raspberry Pi running Freeradius 3.017 as an authentication server for use with the switch.

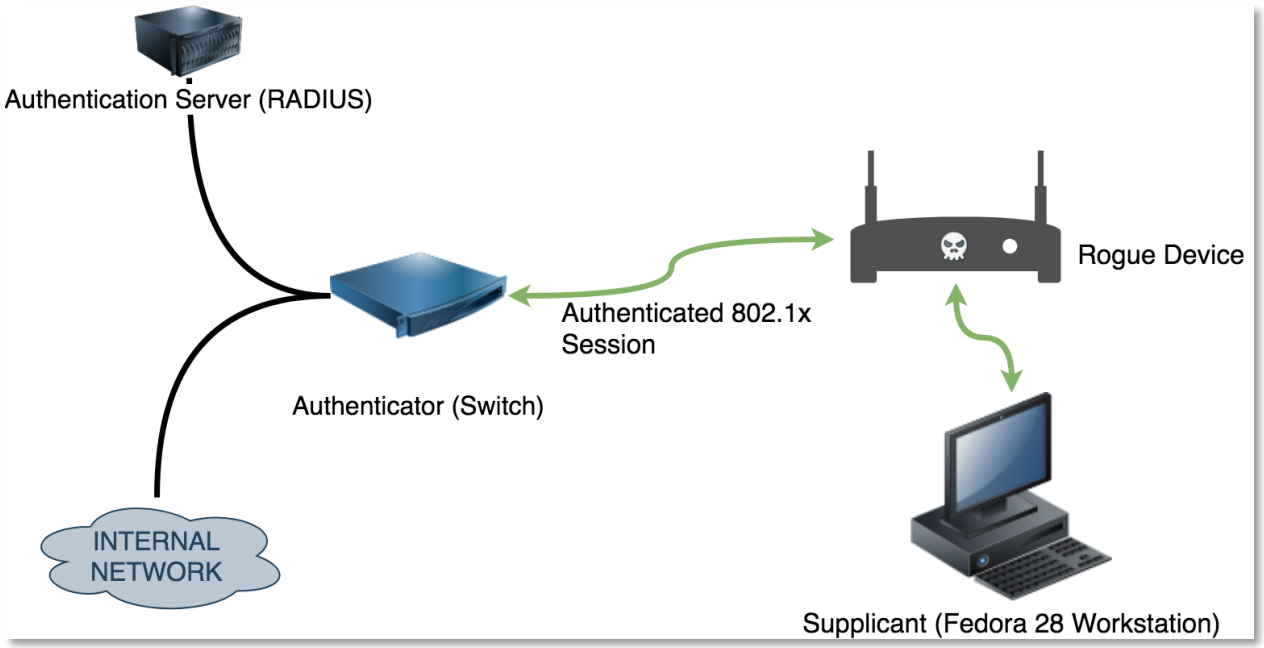


Figure 6 – objective: introducing a rogue device between the authenticator and supplicant

The goal of this experiment was to successfully bypass multiple variations of 802.1x by introducing a rogue device to the network, either by placing it as a bridge between the supplicant and authenticator (see *Figure 6* above) or by connecting it directly to the authenticator itself (see *Figure 7* below).

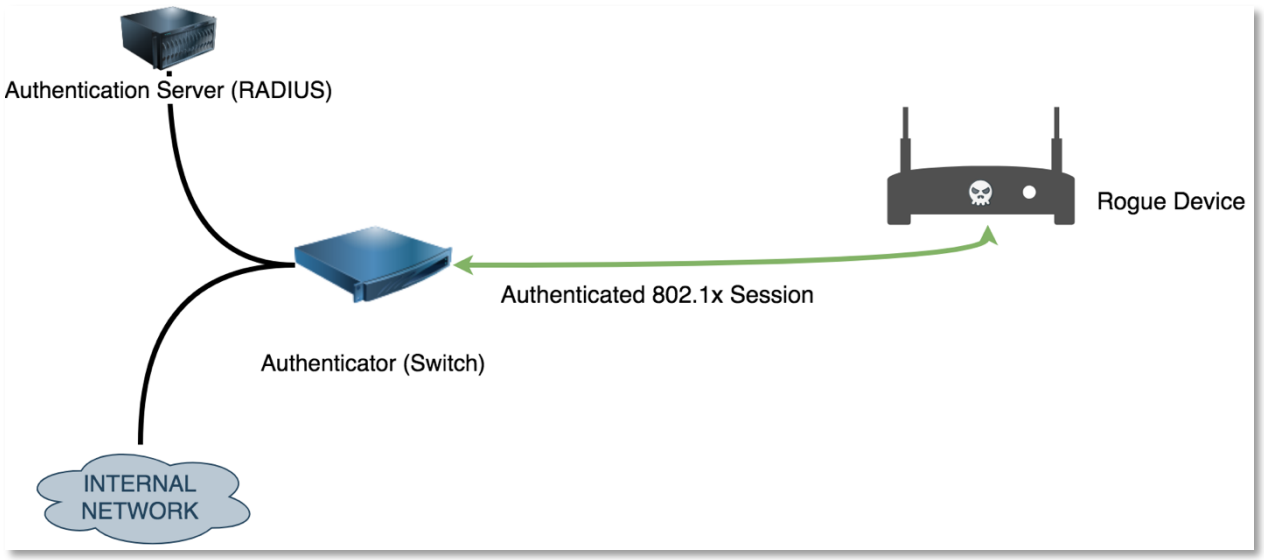


Figure 7 – objective: introducing a rogue device directly to the authenticator

To do this, we constructed two rogue devices: one intended for purely bridge-based bypass methods and the other for mechanically assisted bypass methods.

III.2 Rogue Device A: Pure Bridge-based Design

Rogue Device A followed a pure bridge-based design, as shown in *Figure 8* below. More details about the side channel and transparent bridge can be found in III.4 Establishing a Side Channel and IV. Improvements to Classical Bridge-based 802.1x Bypass.

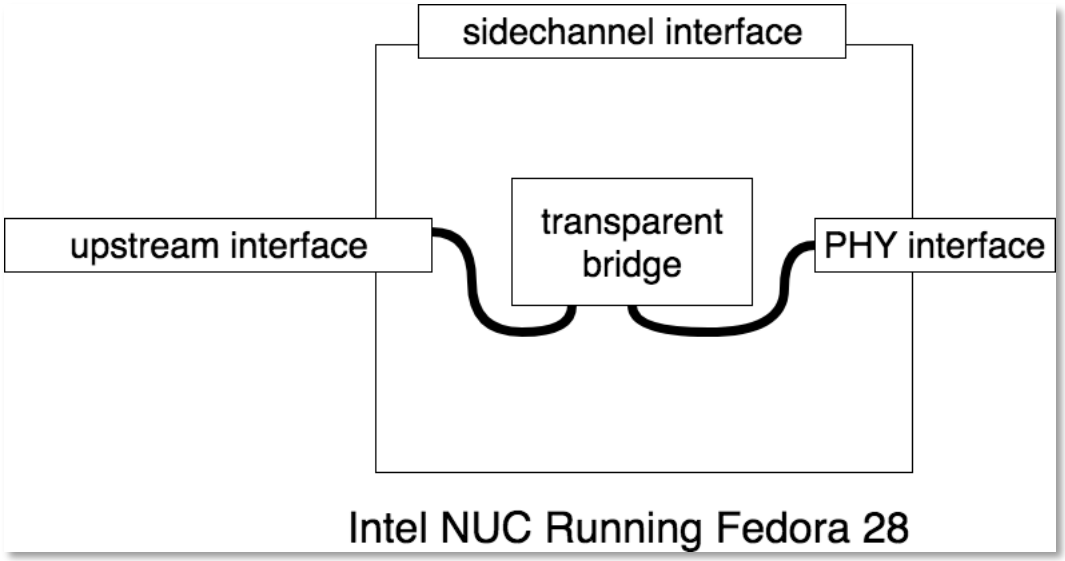


Figure 8 – Rogue Device A

The device consisted of an Intel NUC micro-computer running Fedora 28 and equipped with the following network interface cards:

- **upstream** – the upstream interface to connect the rogue device with the authenticator (or switch). We used a single Ugreen USB 2.0 to RJ45 Network Adapter for this purpose.
- **PHY** – the PHY interface to connect the rogue device with the authenticator (or switch). We used a single Ugreen USB 2.0 to RJ45 Network Adapter for this purpose.
- **sidechannel** – the sidechannel interface consisted of a single USB LTE modem used to provide a backdoor into the device.

Additionally, the device was equipped with the silentbridge software we wrote as part of this research project.

III.3 Rogue Device B: Mechanically Assisted Bypass

Rogue Device B builds off the design of Rogue Device A, keeping all of the key design elements of the first device while adding two physical A/B Ethernet splitters to bypass the device entirely. When the splitters are in position A, they connect directly to each another using an ethernet patch cable. This causes the device acts as an ethernet extender, bypassing the network interfaces of the rogue device entirely.

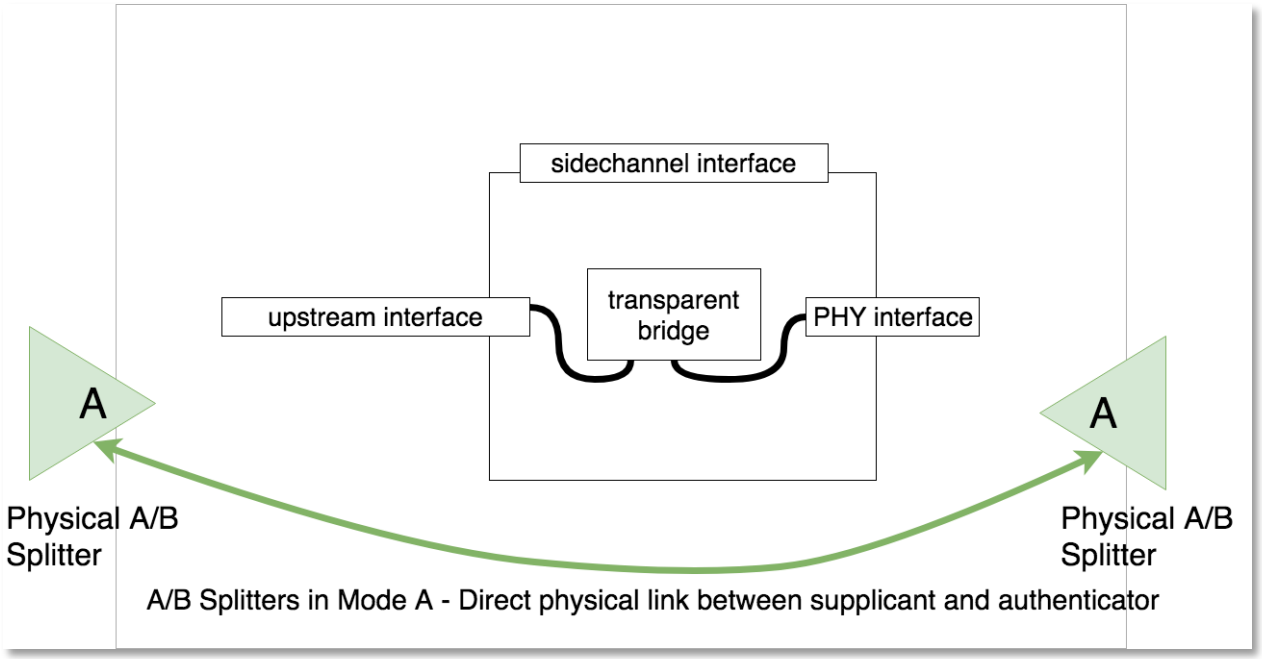


Figure 9 – Rogue Device B

When the splitters are in position B, ethernet traffic passes directly to the upstream and PHY interfaces of the rogue device. Specifically, placing both splitters in position B connects the upstream interface to the authenticator, and connects the PHY interface to the supplicant.

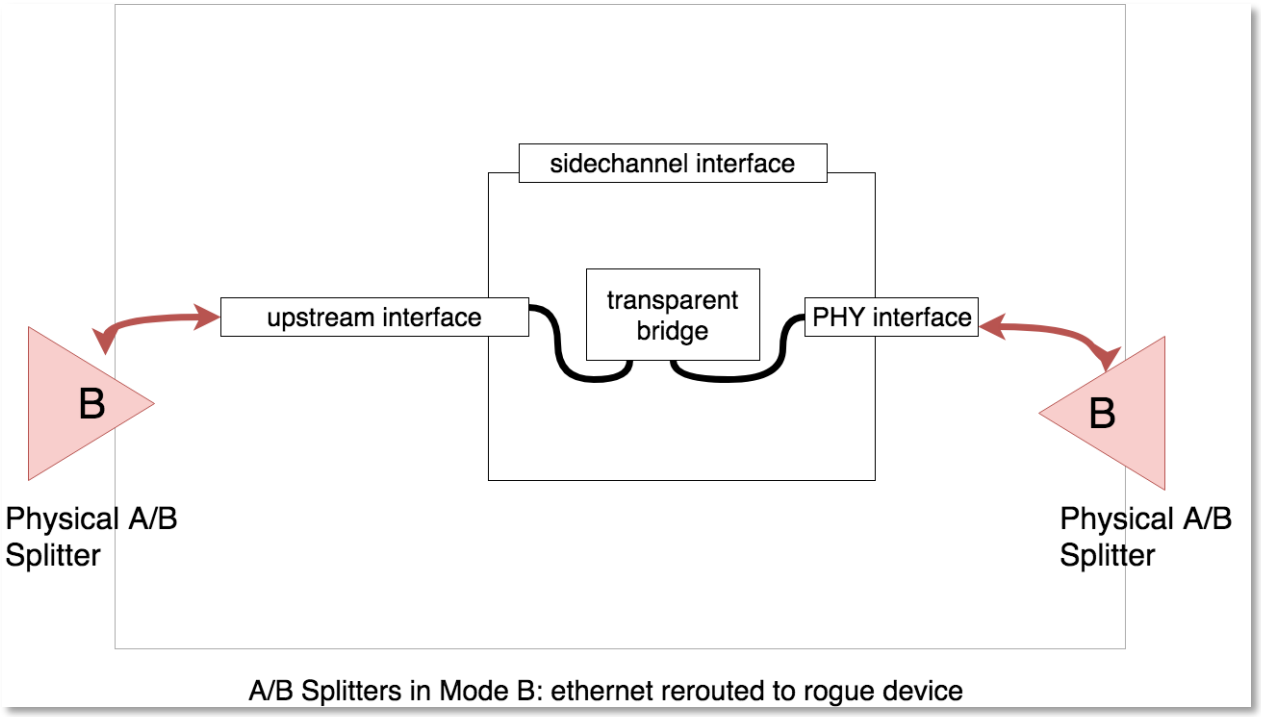


Figure 10 – Rogue Device B

Both A/B splitters can be operated independently of one another. To implement this functionality, we modified a pair of MT-VIKI FBA_MT-RJ45-2M RJ-45 ethernet splitters to each be controlled by a pair of 12mm, 24V solenoids. The solenoids were controlled by an Arduino compatible microcontroller connected to the rogue device over serial connection.

We recognize that this may not be the most efficient way of controlling the device, but designing an Ethernet relay free of impedance issues was beyond the scope of this research.

III.4 Establishing a Side Channel

Establishing a side channel to remotely access the device is required to perform the Rogue Gateway and Bait n Switch attacks we describe, and gives us a way of controlling the rogue device even when it is not connected to the target network.

We equipped both rogue devices with Linux-compatible LTE modems configured to obtain an IP address on boot. We then configured the devices to allow remote access through the LTE modem using a reverse SSH tunnel to an SSH redirector, as shown in *Figure 11* below.

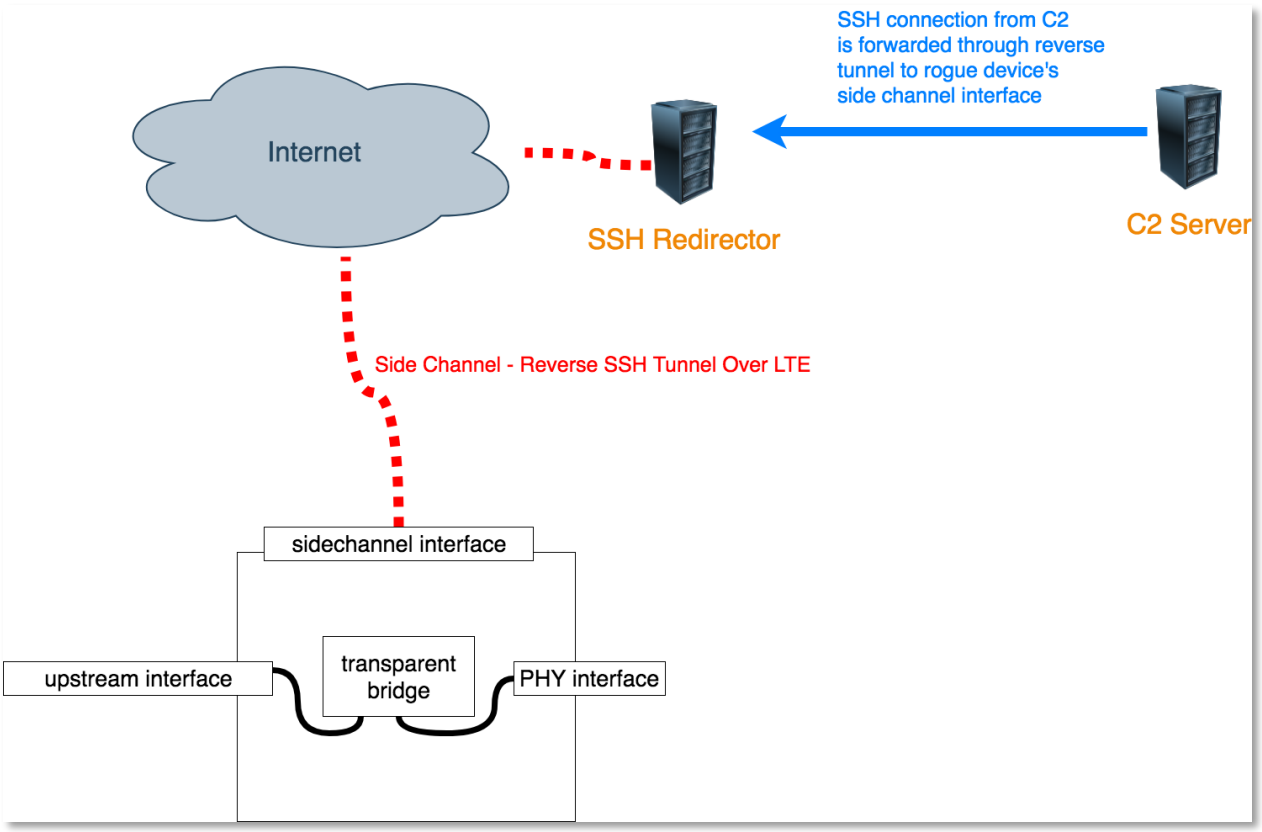


Figure 11 – Establishing a side channel using a reverse SSH tunnel over LTE

In this configuration, the rogue device initiates a reverse SSH tunnel from the sidechannel interface to the redirect on boot. The redirector then forwards incoming SSH connections through the reverse tunnel to the rogue device. We followed the configuration described by Stanislav Sinyagin in his blog posts *Call Home SSH Scripts* and *Improved Call Home SSH Scripts* [26][27].

III.5 Putting It All Together

The full lab setup is shown in the diagrams below. The first configuration uses Rogue Device A and is shown in *Figure 12* below.

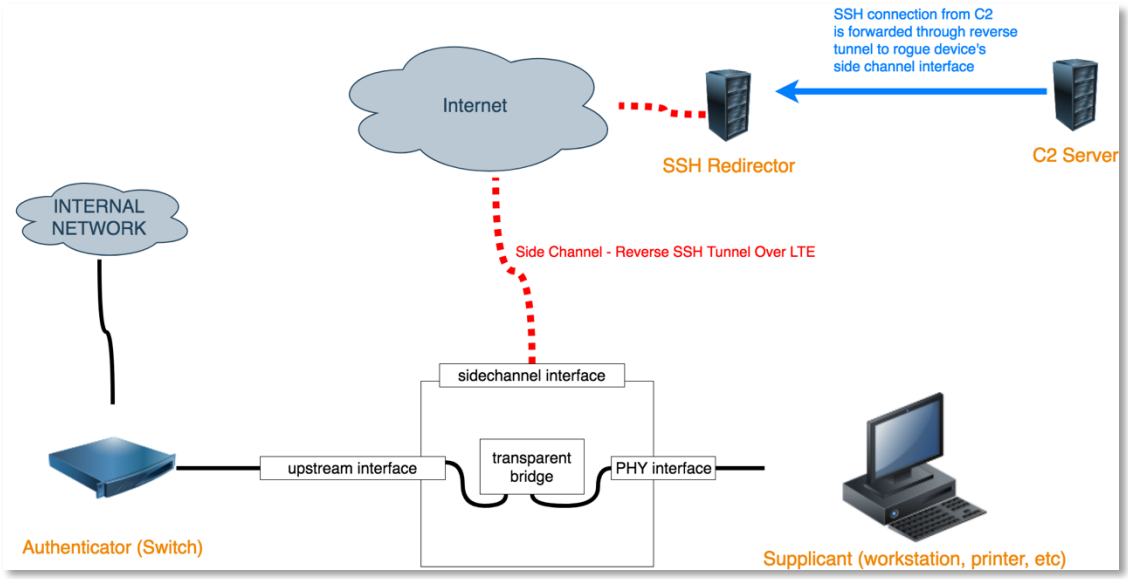


Figure 12 – the complete lab environment (using Rogue Device A)

The second configuration uses Rogue Device B and is shown in *Figure 13* below.

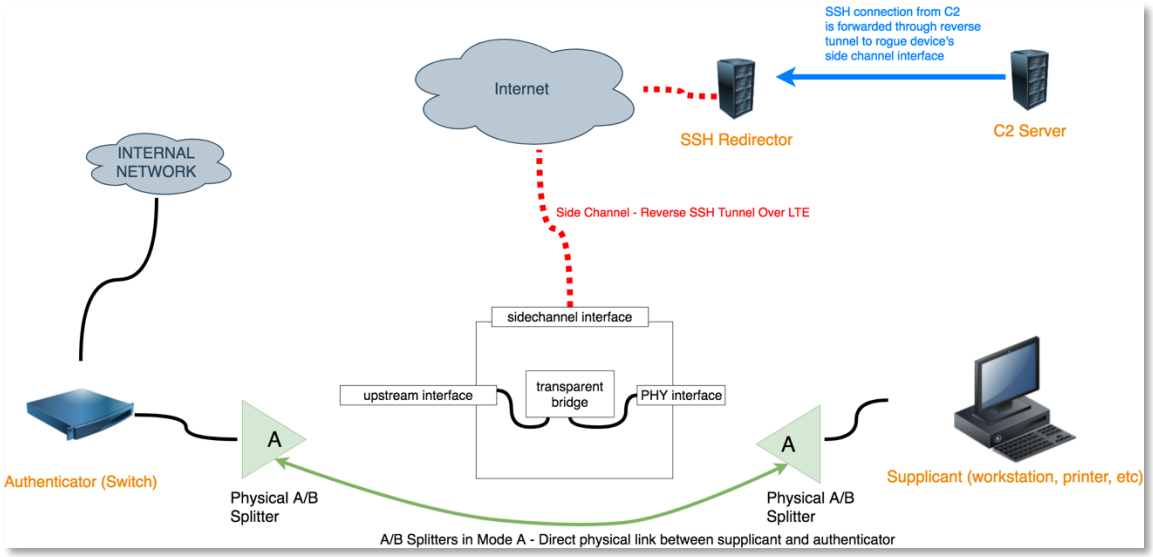


Figure 13 – the complete lab environment (using Rogue Device B)

IV. Improvements to Classical Bridge-based 802.1x Bypass

One of the first steps we took in exploring this topic was attempting to recreate the classical bridge-based 802.1x developed by Alva Duckwall [4]. As we mentioned in II.2 The Current State of Wired Port Security, this attack uses a transparent bridge to silently introduce a rogue device between the authenticator and the supplicant [4]. The ability to interact with the network is granted by using iptables to source NAT (SNAT) traffic originating from the device [4]. To reduce risk of discovery, iptables is used to prevent the rogue device from using a source port that is already in use by the supplicant [4]. Additionally, a hidden SSH service is created on the rogue device by using iptables to forward traffic destined for the supplicant's IP address on a specific port to the rogue device on port 22 [4].

In this section we'll discuss the improvements we made to this original attack, all of which were developed during the process of recreating it.

Leveraging Native EAPOL Forwarding

One of the most immediate drawbacks to the traditional bridge-based approach is that the Linux kernel will not forward EAPOL packets over the bridge, presumably for security reasons [4]. Existing tools for performing bridge-based 802.1x bypasses deal with this problem in one of two ways: patching the Linux kernel, or using high-level libraries such as Scapy [4][6]. Relying on kernel patches can be unwieldy, and relying on high level scripting languages such as Python can slow the bridge under heavy loads. To make matters worse, none of the publicly available kernel patches work with the latest versions of Linux [17][18].

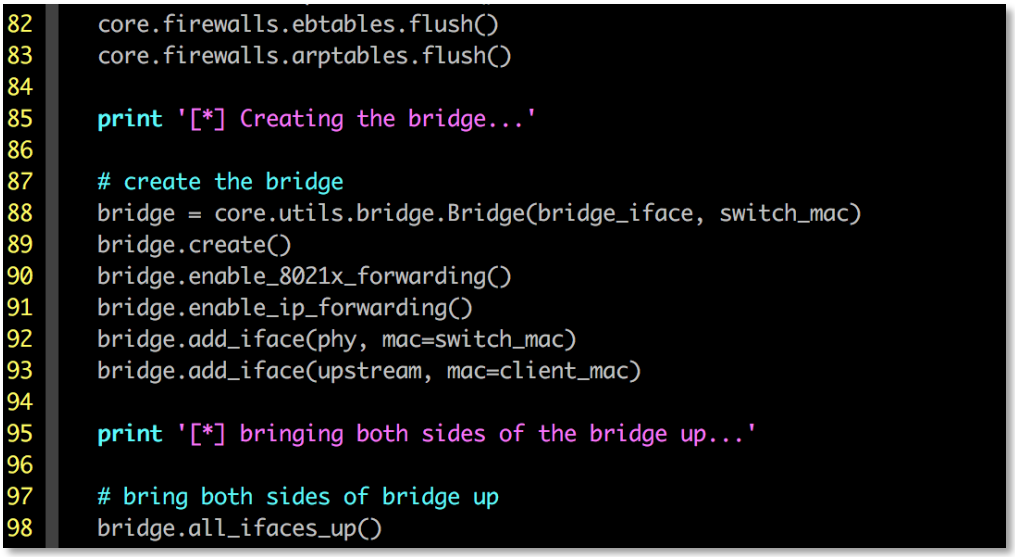
```
65          os.system('ifconfig %s down' % self.name)
66
67      def enable_8021x_forwarding(self):
68
69          os.system('echo 8 > /sys/class/net/%s/bridge/group_fwd_mask' % self.name)
70
71      def enable_ip_forwarding(self):
72
73          os.system('echo 1 > /proc/sys/net/ipv4/ip_forward')
```

Figure 14 – 802.1x forwarding can be enabled using the proc filesystem

Fortunately, the situation has dramatically improved since Duckwall's script was released: as of 2012, the Linux kernel no longer must be patched in order to bridge EAPOL packets [11]. Instead, users can enable this feature using the proc filesystem [11]. We updated our implementation of Duckwall's classical 802.1x bypass to reflect this, ensuring long-lasting reliability regardless of the kernel version in use.

Bypassing Sticky MAC

A second minor improvement that we made was to ensure that our classical 802.1x bypass accounted for the widespread use of Sticky MAC by modern authenticators. Most 802.1x capable switches created in recent years support a feature known as Stick MAC, which dynamically associates the MAC address of the supplicant to the switch port once the supplicant has successfully authenticated [28][29]. If another MAC address is detected on the switch port, a port security violation occurs and the port is blocked [28][29]. To keep this from occurring, our implementation sets the bridge and PHY interfaces to the MAC address of the authenticator and sets the upstream interface to the MAC address of the supplicant (see *Figure 15*).



```
82    core.firewalls.ebtables.flush()
83    core.firewalls.arptables.flush()
84
85    print '[*] Creating the bridge...'
86
87    # create the bridge
88    bridge = core.utils.bridge.Bridge(bridge_iface, switch_mac)
89    bridge.create()
90    bridge.enable_8021x_forwarding()
91    bridge.enable_ip_forwarding()
92    bridge.add_iface(phy, mac=switch_mac)
93    bridge.add_iface(upstream, mac=client_mac)
94
95    print '[*] bringing both sides of the bridge up...'
96
97    # bring both sides of bridge up
98    bridge.all_ifaces_up()
```

Figure 15 – bypassing Sticky MAC

Support For Side Channel Interaction

Perhaps the most significant improvement we made to the classical 802.1x bypass was to add support for remote access via a side channel, as described in III.4 Establishing a Side Channel. In Duckwall's classical 802.1x bypass, all outbound ARP and IP traffic is initially blocked while the transparent bridge is being initialized.

```
49  #start dark
50  arptables -A OUTPUT -j DROP
51  iptables -A OUTPUT -j DROP
52
53
54  # bring up the bridge with our bridge IP
55  ifconfig $BRINT $BRIP up promisc
56
57  # creat to source NAT the $COMPMAC
58  # for traffic leaving the device
59  # from the bridge mac address
60  ebtables -t nat -A POSTROUTING -s $SWMAC -o $SWINT -j snat --to-src $COMPMAC
61  ebtables -t nat -A POSTROUTING -s $SWMAC -o $BRINT -j snat --to-src $COMPMAC
62
```

Figure 16 – The original 802.1x bypass by Duckwall, shown above, blocks all outbound ARP and IP traffic while the bridge is being initialized (script hosted by Mubix on Github.com) [18]

Although these restrictions are eventually lifted when the transparent bridge setup is complete, they are still enough to cut off access through the side channel device, consequently causing loss of access to the rogue device. In order to maintain access to the device while the bridge is being initialized, we added a firewall exception that allows outbound traffic from our sidechannel interface only.

```
100     print '[*] Initiate radio silence...'
101
102     # start dark - but make an exception for our side channel
103     core.firewalls.iptables.allow_outbound(sidechannel, port=egress_port)
104     core.firewalls.arptables.allow_outbound(sidechannel)
105     core.firewalls.iptables.drop_all()
106     core.firewalls.arptables.drop_all()
107
108     time.sleep(2)
109
110     print '[*] Bringing the bridge up with a non-routable IP...'
111
```

Figure 17 – creating an exception for side channel traffic

By default, our implementation allows outbound traffic to port 22 from the sidechannel interface, although we also provided users with the ability to specify an alternative port using a command line flag.

```
┌─[solstice@ops11-2] - [~/silentbridge] - [7528]
└─[$] ./silentbridge --create-bridge --upstream eno2 --phy eno1 --egress-port 443
```

Figure 18 – specifying an alternative egress port

Conclusion

Our improved classical bypass worked as expected when used against 802.1x-2004, as shown in the screenshot below. The attack can be performed with both Rogue Device A (basic implementation) and with Rogue Device B (mechanical splitters). The addition of mechanical splitters only enhance this technique, and is not an essential component of the attack itself.

```
┌[root@localhost] [/dev/pts/4]
└[/home/solstice]> cd silentbridge
┌[root@localhost] [/dev/pts/4] [master ⚡]
└[/home/solstice/silentbridge]> python main.py --create-bridge  --phy eno1 --client-mac 38:60:77:d0:ef
:dd --upstream enp0s20f0u4 --switch-mac 00:42:5a:87:57:85
net.ipv6.conf.all.disable_ipv6 = 1
net.ipv6.conf.default.disable_ipv6 = 1
[*] Creating the bridge...
[*] bringing both sides of the bridge up...
[*] Initiate radio silence...
[*] Bringing the bridge up with a non-routable IP...
┌[root@localhost] [/dev/pts/4] [master ⚡]
└[/home/solstice/silentbridge]>
```

Figure 19 – performing a bridge-based 802.1x bypass

Reproduction Command:
./silentbridge --create-bridge --upstream UPSTREAM_IFACE_ --phy PHY_IFACE_ --client-mac SUPPLICANT_MAC_ADDRESS --switch-mac SWITCH_MAC_ADDRESS

V. Bate n Switch Attack: An Alternative To Packet Injection

Traditional 802.1x bypass techniques tend to focus on ways of interacting with a protected wired network without actually authenticating. Although this can be accomplished using packet injection when the protected network uses 802.1x-2004 or earlier, a simpler approach is sometimes better. One situation in which this is particularly true is when 802.1.x-2010 is used to protect the wired network, as MACsec effectively denies us the opportunity to use packet injection.

The Bait N Switch attack is a means of using stolen credentials to authenticate directly to a protected wired network without tripping port security. This allows the attacker to interact with the network without relying on packet injection.

V.1 Bridge-Based Approach

In the first variation of the Bait n Switch attack, we use the Rogue Device A configuration described in III.2 Rogue Device A: Pure Bridge-based Design. We begin the attack by performing the Classical Bridge-based 802.1x bypass described in IV. Improvements to Classical Bridge-based 802.1x Bypass, using the rogue device to establish a transparent bridge between the supplicant and authenticator as shown in *Figure 20* below.

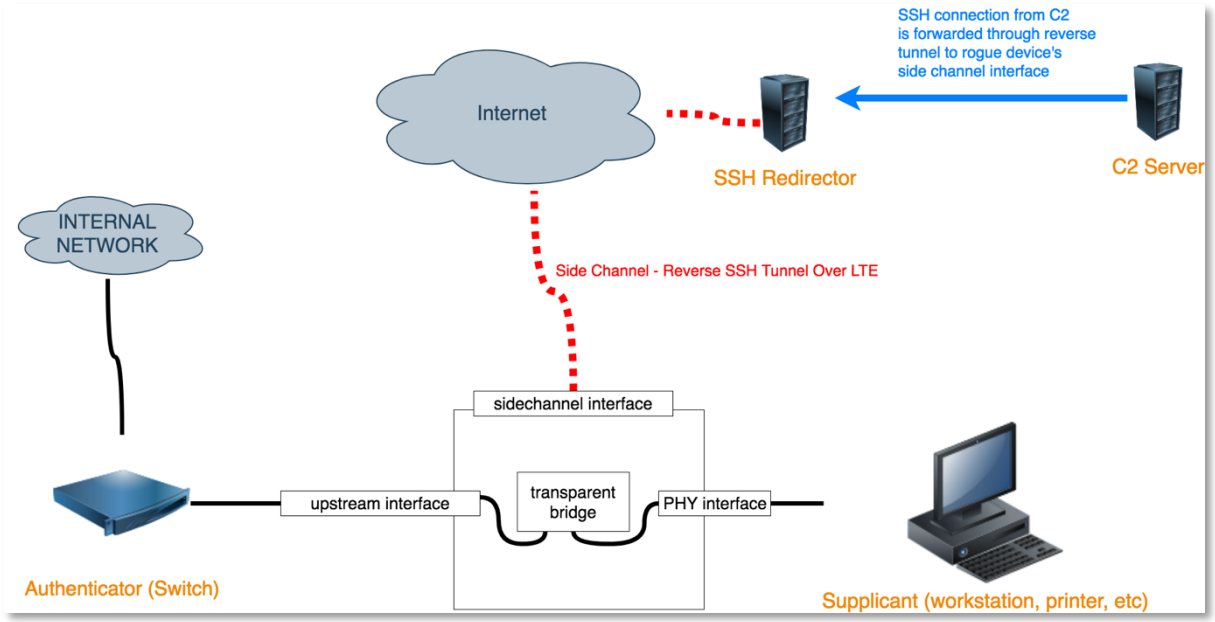


Figure 20 – establishing a bridge-based 802.1x bypass in preparation for a Bait n Switch attack

We then disconnect the supplicant from the network by bringing our PHY and bridge interfaces down. Next, we set the MAC address of our upstream interface to the MAC address of the supplicant and use the upstream interface to authenticate with the authenticator using stolen RADIUS credentials. Finally, we give our upstream interface a static IP address that matches the one previously assigned to the client device.

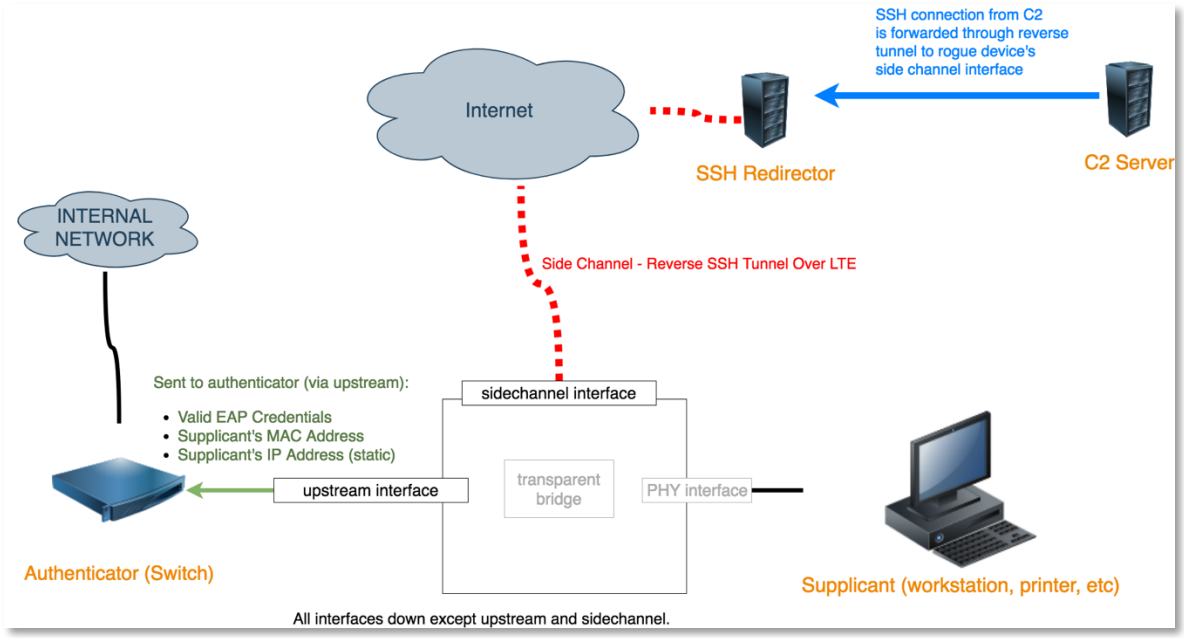


Figure 21 – performing a Bait n Switch attack

In essence, the Bait n Switch attack silently swaps the authorized device with the attacker's rogue device. The attack is simple, can be used to achieve full network interaction, and is reasonably stealthy so long as performed during off-hours when the affected supplicant is unlikely to be in use.

V.2 Using Mechanical A/B Splitters

Using the Rogue Device B configuration described in III.3 Rogue Device B: Mechanically Assisted Bypass, we can use the Bait n Switch attack to authenticate with networks protected by 802.1x-2010 and MACsec. When combined with the Rogue Gateway Attack described in VI. Defeating MACsec Using Rogue Gateway Attacks, this technique can bypass 802.1x-2010 in cases where weak EAP implementations are used.

To begin the attack, we first introduce our rogue device to the network as shown in *Figure 22* below. As we do this, we make sure that both A/B splitters are in position A, which preserves the direct physical link between the authenticator and supplicant.

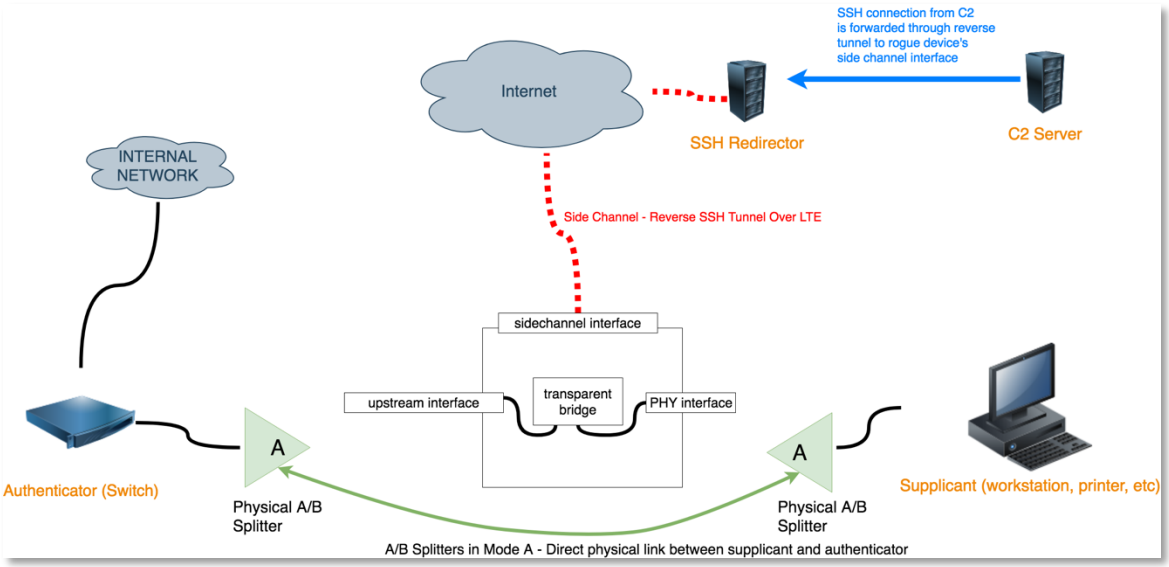


Figure 22 – preparing to perform Bait n Switch using Rogue Device B configuration

We then disconnect the supplicant from the network by bringing our PHY and bridge interfaces down and placing the splitters in the B position, which reroutes the physical link to the rogue device. Next, we set the MAC address of our upstream interface to the MAC address of the supplicant and use it to authenticate with the authenticator using stolen RADIUS credentials. Finally, we give our upstream interface a static IP address that matches the one previously assigned to the supplicant.

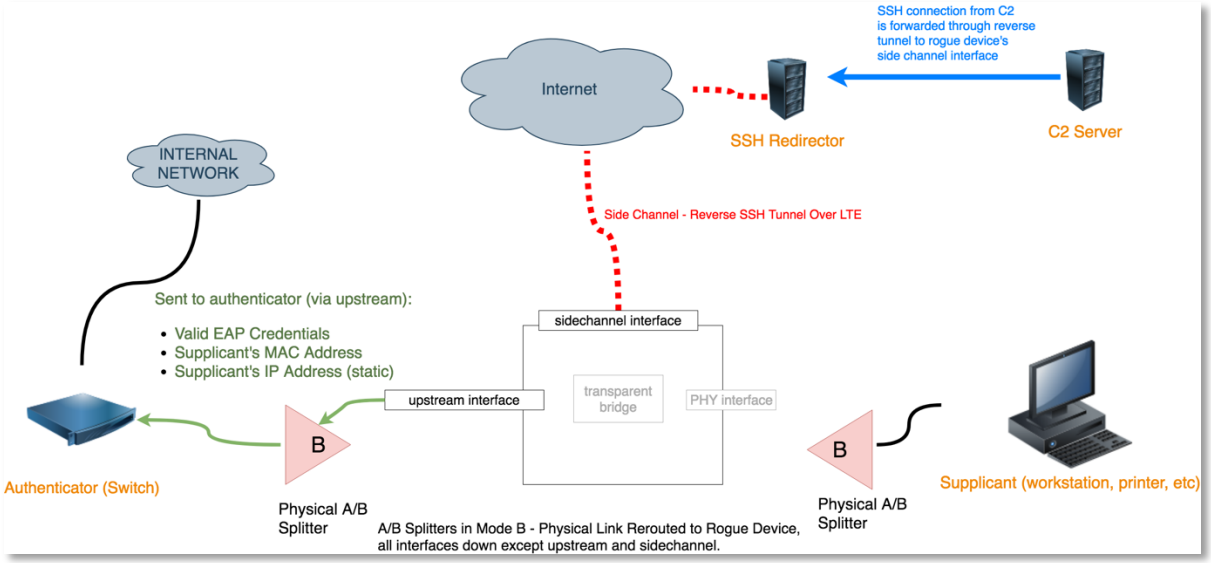


Figure 23 – performing the Bait n Switch using Rogue Device B configuration

VI. Defeating MACsec Using Rogue Gateway Attacks

The 802.1x bypass techniques pioneered by Riley, “Abb,” Duckwall, and later improved upon by Legrand all take advantage of the same fundamental security issues that affect 802.1x-2004: the standard does not provide encryption or the ability to perform authentication on a packet-by-packet basis [3][4][6][7].

To address these issues, the IEEE developed a new standard, 802.1x-2010, which uses MACsec to provide Layer 2 encryption and packet-by-packet integrity checks [7]. MACsec provides encryption on a hop-by-hop basis, which successfully mitigates the bridge-based attacks that we discussed in II. Background and Prior Work and IV. Improvements to Classical Bridge-based 802.1x Bypass while also providing network administrators with a means to inspect data in transit [7][8].

MACsec and 802.1x-2010 work in three phases: authentication and master key distribution, session key agreement, and session secure [7][8][9]. Authentication is intended to be performed using EAP, although 802.1x-2010 allows for a Pre-Shared Key (PSK) to be used as well, either as a fallback or as a direct replacement for EAP.

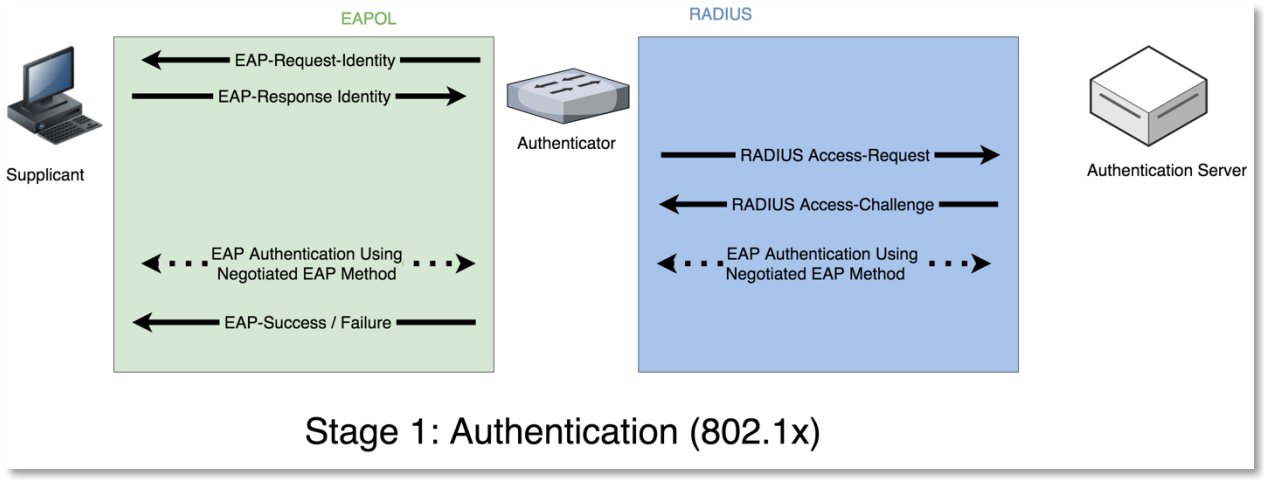


Figure 24

When EAP is used as the authentication mechanism, the entire process is reminiscent of WPA2-EAP, in that it involves a supplicant (client device), authenticator (switch), and authentication server. When a device is first connected to a protected switch port, the authenticator initiates the EAP authentication process by sending an EAP-Request-Identity frame to the supplicant [7][8][9]. The supplicant then responds with its identity, which is forwarded by the authenticator to the authentication server as a RADIUS Access-Request message.

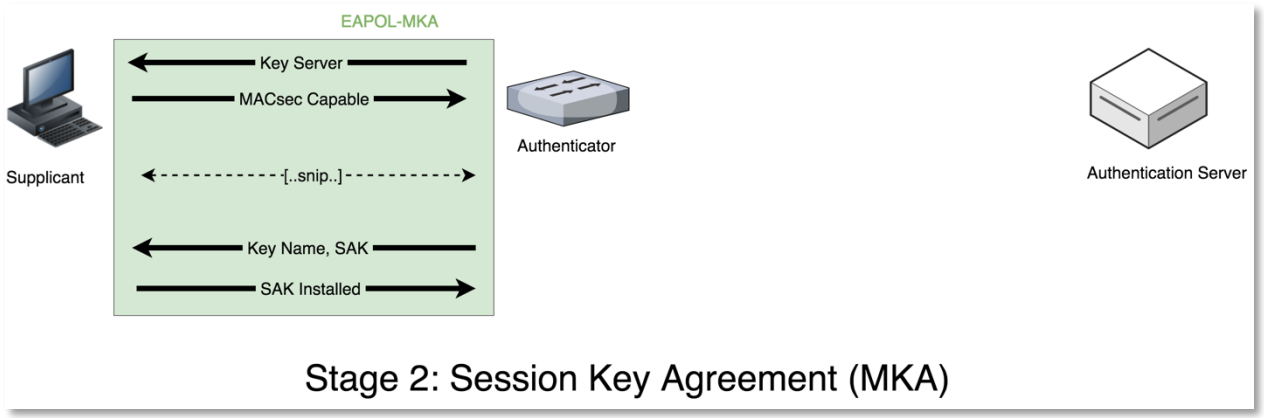


Figure 25

At this point, the supplicant and authentication server negotiate an EAP type that supports Master Session Key (MSK) derivation, and the supplicant attempts to authenticate using the agreed-upon EAP type [7][8][9]. If authentication

succeeds, the supplicant and switch perform the session key agreement and then enter the Session Secure state, where MACsec's Layer 2 encryption becomes active.

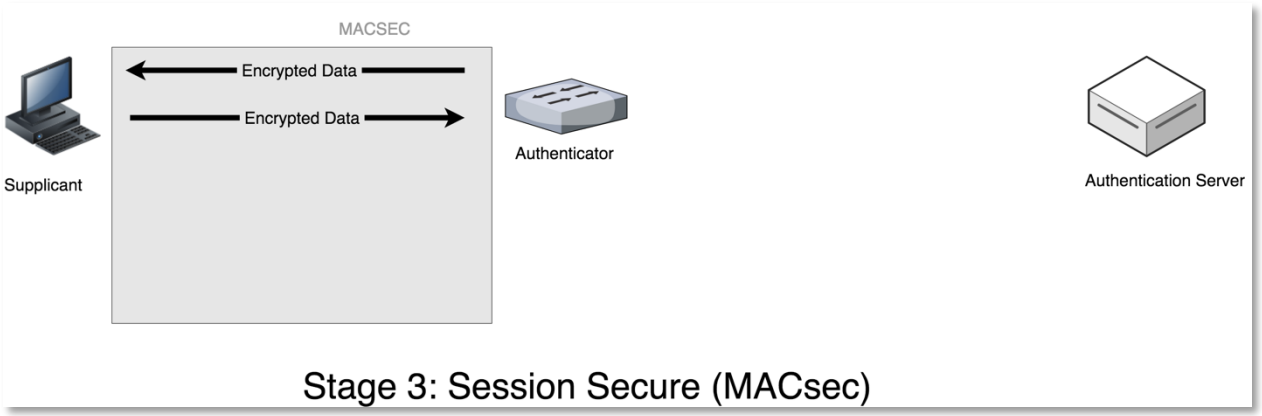


Figure 26

With a couple of exceptions, which we will examine shortly, the hop-by-hop encryption provided by MACsec prevents attackers from bypassing 802.1x-2010 by bridging two network interfaces together as is possible with 802.1x-2004 [10]. Possible exceptions are provider bridges (PBs) and backbone bridges (PBBs), which warrant further investigation despite being outside the scope of this discussion [10]. However, it is possible for an attacker to introduce a rogue device to a network protected by 802.1x-2010 using more rudimentary methods.

VI.1 Defeating MACsec Using Rogue Gateway Attacks

Significantly, 802.1x-2010 still uses EAP to authenticate new devices to the network [7]. As we described briefly in II. Background and Prior Work, there are many ways of implementing EAP, and most of them suffer from some sort of security issue. The 802.1x-2010 standard allows any EAP method so long as it meets the following mandatory requirements:

- Supports mutual authentication between client and server
- Supports derivation of keys that are at least 128 bits in length
- Generates an MSK of at least 64 octets

Most of the commonly seen weak EAP methods, including EAP-PEAP and EAP-TTLS, meet these requirements. It is up to individual vendors to decide whether or not these methods should be supported, and up to system administrators to choose EAP methods that can withstand man-in-the-middle attacks.

The implication of support for weak EAP methods is that security of 802.1x-2010 deployments are still only strong as the EAP methods used. Unless the target deploys strong forms of EAP such as EAP-TLS or EAP-PEAP with globally enforced rejection of invalid certificates, an attacker can simply repurpose existing principles for attacking these authentication protocols as a means of bypassing port security.

Consider a scenario in which EAP-TTLS provides authentication to a network secured using 802.1x-2010. The attacker could introduce a Rogue Device B between the supplicant and authenticator as shown in *Figure 27* below.

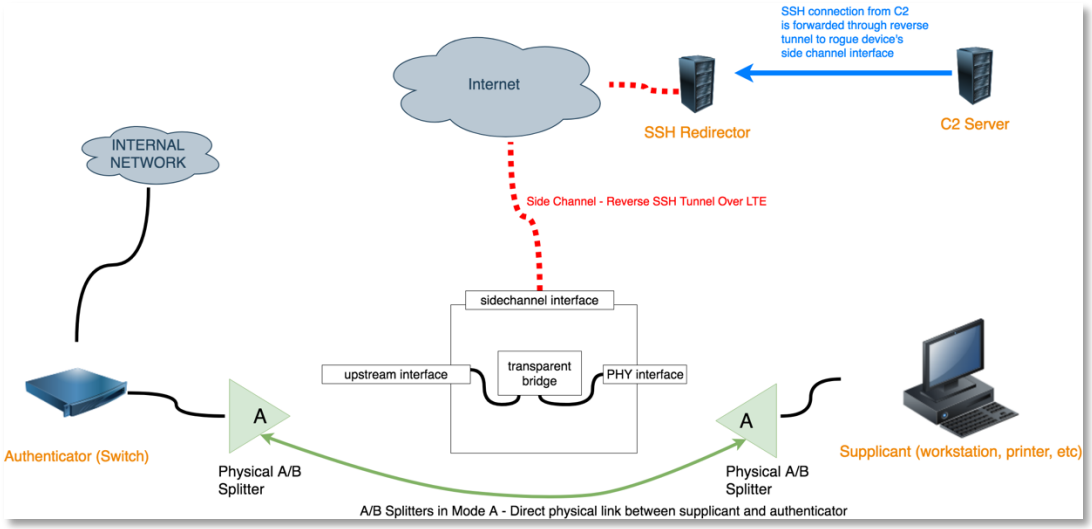


Figure 27

As described in III.3 Rogue Device B: Mechanically Assisted Bypass, this rogue device configuration makes use of two mechanically controlled A/B ethernet splitters: when the splitters are in the "A" position, the supplicant is allowed to communicate directly with the authenticator as shown in *Figure 27* above. To initiate the attack, the attacker brings the upstream interface down and flips the splitters to the "B" position as shown in *Figure 28* below. This provides direct connectivity between the rogue device and the supplicant. The attacker then starts hostapd as a rogue RADIUS server, configuring it to listen on the rogue device's PHY interface.

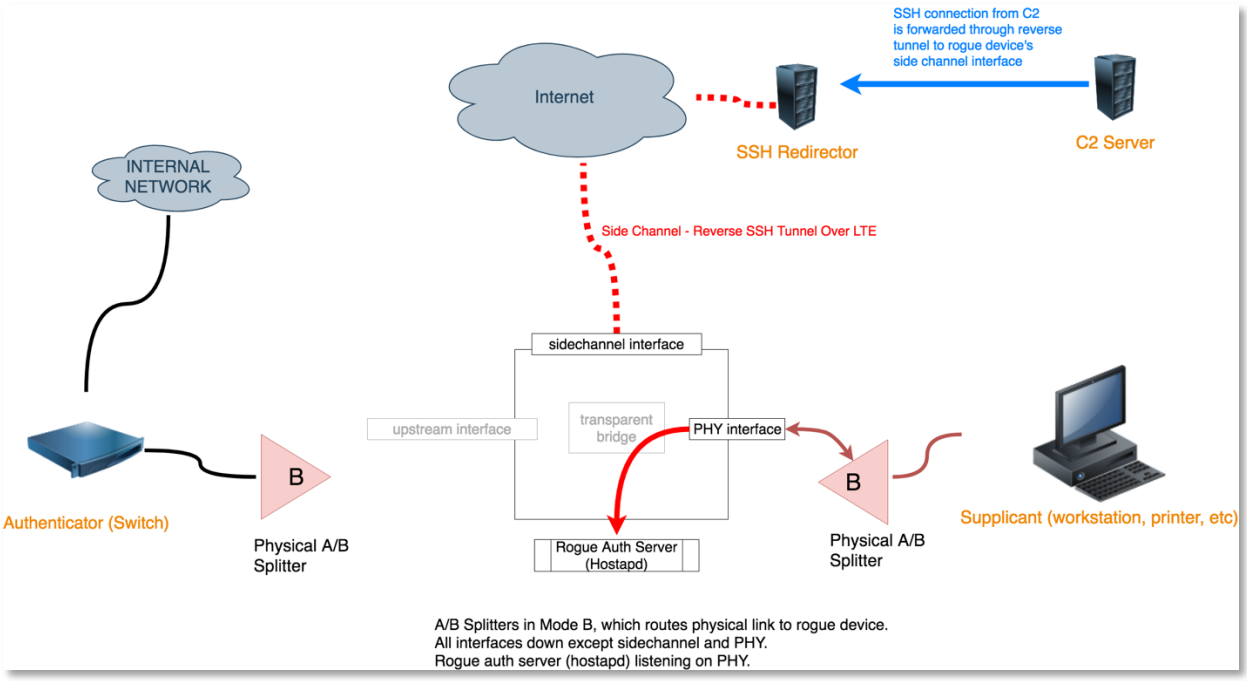


Figure 28

The attacker then sends a spoofed EAPOL-Start frame to hostapd, causing hostapd to send an EAP-Request-Identity frame to the supplicant. In response, the supplicant attempts to authenticate with the rogue device. As long as the supplicant accepts the rogue device's x.509 certificate, the attacker will capture an MS-CHAPv2 challenge and response from the supplicant which can be cracked to obtain plaintext credentials [13].

Once the attacker cracks the captured hashes, a Bait n Switch attack connects the rogue device to the network as shown in *Figure 29* below.

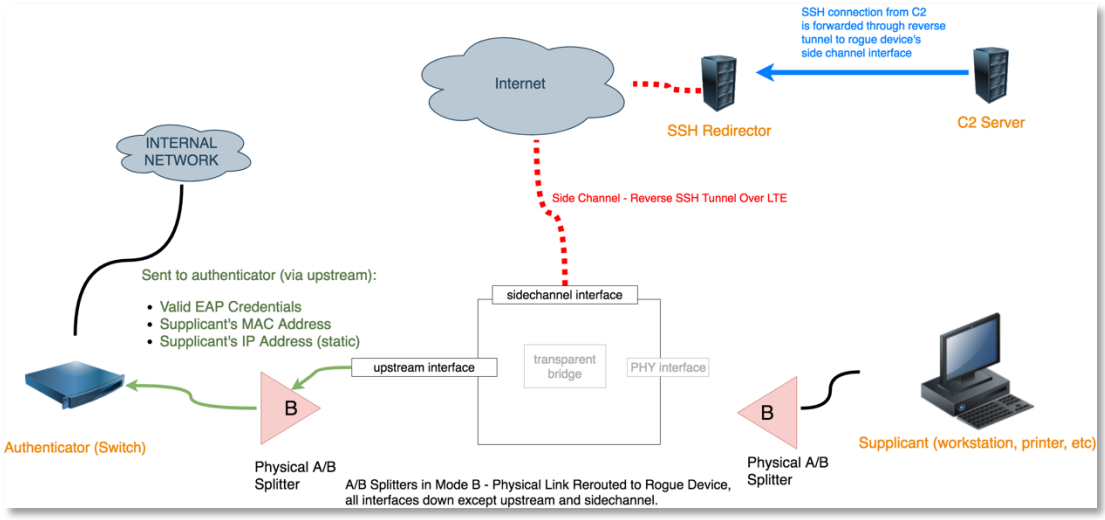


Figure 29

VII. Dealing with Improvements to Peripheral Device Security

Previously, we described how improved 802.1x support by peripheral devices such as multifunction printers has made it increasingly difficult to bypass wired port security by looking for policy exceptions. While improved adoption of 802.1x is a step in the right direction, it does not necessarily translate to strong port security for peripheral devices. 802.1x authentication relies upon EAP, and most forms of EAP have known security issues that have existed for over a decade [13].

Adoption rates for secure forms of EAP, such as EAP-TLS or EAP-PEAP with forced rejection of untrusted certificates, are relatively poor due to the complexity involved with deploying these technologies at scale [12]. For peripheral devices such as printers, the adoption of secure EAP is even worse, considering that very few cost-effective peripheral devices can be configured using Group Policy.

Thus, while port-security exceptions for peripheral devices may not be as prevalent as they used to be, peripheral devices themselves are still highly viable entry points because they are less likely to be configured using strong forms of EAP. What is missing are techniques for attacking weak forms of EAP within a wired network.

In this section, we introduce two attacks against EAP-MD5 and EAP-PEAP on wired networks, since these are the two forms of weak EAP most commonly used by peripheral devices and even some workstations. These attacks allow us to continue to use peripheral devices as entry points to networks protected by 802.1x-2004.

VII.1 EAP-MD5 Forced Reauthentication Attack

EAP-MD5 is one of the most widely used forms of EAP used to protect peripheral devices such as multifunction printers and cheap IP phones. Despite its many flaws, it is one of the easiest forms of EAP to setup and configure, which makes it particularly well suited for this purpose.

VII.1.A Passive Attack Against EAP-MD5

Consider how we can use the existing attacks against EAP-MD5 described in II.4.A EAP-MD5 in conjunction with the classical 802.1x bypass described in II. Background and Prior Work and IV. Improvements to Classical Bridge-based 802.1x Bypass to attack peripheral devices. We can use bridges and MAC spoofing to place a rogue device between the supplicant and the authenticator, and use the rogue device to sniff EAPOL packets as the supplicant authenticates with the network. We also have proposed the Bait n Switch method described in V. Bate n Switch Attack: An Alternative To Packet Injection as a means of authenticating with a network protected by 80.1x.

Using these attacks allows us to bypass 802.1x by locating a peripheral device (supplicant) that is configured to use EAP-MD5, installing a rogue device between the supplicant and the switch (authenticator), and waiting for the peripheral device to reauthenticate with the network. We then can sniff the EAP-MD5 challenge and response, use a dictionary attack to obtain the plaintext username and password, and finish by using a Bait n Switch attack to authenticate with the network.

There is one major drawback to this approach: we must wait for the supplicant to reauthenticate with the switch. Realistically, this will not occur unless the device is unplugged, turned off, or becomes inactive for an extended period of time [16].

For our attack to be truly useful, we need to be able to force the device to reauthenticate. The most obvious way of doing this is to briefly disconnect the supplicant from the authenticator. However, for this to work we would have to physically disconnect the Ethernet cable from the bridge, as merely disabling a network interface briefly will not trigger reauthentication [16]. We can do this remotely if we're using the Rogue Device B configuration for our rogue

device, but it would be better to have a solution that can be implemented using Rogue Device A, which has less overhead.

A better approach is to perform an attack that takes advantage of the way the EAP authentication process is initiated.

VII.1.B EAP-MD5 Forced Reauthentication Attack

The first two steps of the EAP authentication process are:

> **Step 1** – (optional) The supplicant sends the authenticator an EAPOL-Start frame [1][2][9].
> **Step 2** – The authenticator sends the supplicant an EAP-Request-Identity [1][2][9].

In order to provide a means for the authenticator to force the supplicant to reauthenticate, Step 1 is considered optional. EAP authentication can be initiated using Step 1 (EAPOL-Start) or Step 2 (EAP-Request-Identity) [1][2][9].

This approach, however, fails to provide the supplicant a means of verifying whether incoming EAP-Request-Identity frames have been sent in response to an EAPOL-Start frame, or whether reauthentication has been initiated independently by the authenticator itself. Regardless of whether the authentication process starts at Step 1 or Step 2, the structure of the EAP-Request-Identity frame remains the same.

This means that we can force reauthentication by sending a forged EAPOL-start frame to the authenticator as if it came from the supplicant. This will cause the authenticator to believe that the supplicant is initiating authentication, which will cause the authenticator to send the supplicant an EAP-Request-Identity frame. The supplicant then receives the EAP-Request-Identity frame from the authenticator and responds with the next phase of the authentication process. The authentication process then proceeds as normal, as both the supplicant and the authenticator believe that the other party has initiated the transaction.

```
3 def force_reauthentication(iface, client_mac):
4
5     # send an EAPOL-Start broadcast from client's mac
6     sendp(Ether(src=client_mac, dst="01:80:c2:00:00:03")/EAPOL(type=1), iface=iface)
```

Figure 30 – using Scapy to send EAPOL-Start frames

Using this technique, we can upgrade our Passive Attack Against EAP-MD5 (see VII.1.A Passive Attack Against EAP-MD5) into an active one. We begin by introducing the rogue device to the network between the authenticator and supplicant, establishing a transparent bridge to passively sniff traffic. We then force reauthentication, sniffing the resulting EAP-MD5 challenge and response. We then use a dictionary attack to obtain plaintext credentials and connect to the network using a Bait n Switch.

VII.1.C Proposed Mitigation to EAP Forced Reauthentication Attacks

A safer way of initiating the EAP authentication process would be to include a safety-bit in the EAP-Request-Identity frame that is set to 1 when the frame was sent in response to an EAPOL-Start frame. When the supplicant receives an EAP-Request-Identity frame, it should check the value of the safety-bit. If the safety-bit is set to 1, and the supplicant did not recently issue an EAPOL-Start frame, the authentication process should be aborted and an alert sent to the authenticator.

VII.2 Leveraging Rogue Gateway Attacks Against Peripheral Devices

The other forms of weak EAP commonly used by peripheral devices on wired networks are EAP-PEAP and EAP-TTLS, discussed earlier in II.4.A EAP-MD5 and II.4.B EAP-PEAP / EAP-TTLS. Attacking EAP-PEAP is considerably more involved than attacking EAP-MD5 since authentication occurs through a secure tunnel and consequently cannot be passively sniffed [13].

Fortunately, an adaptation of the rogue gateway technique discussed in VI. Defeating MACsec Using Rogue Gateway Attacks is useful in this scenario. This time, however, we do not need to use the Rogue Device B configuration since we are not dealing with a scenario involving MACsec. Instead, we can perform the attack completely in software using our transparent bridge as described in IV. Improvements to Classical Bridge-based 802.1x Bypass.

VII.2.A Rogue Gateway Attack Against 802.1x-2004 and EAP-PEAP/EAP-TTLS

We begin by placing our rogue device (A configuration) between the supplicant and authenticator as shown in *Figure 31* below.

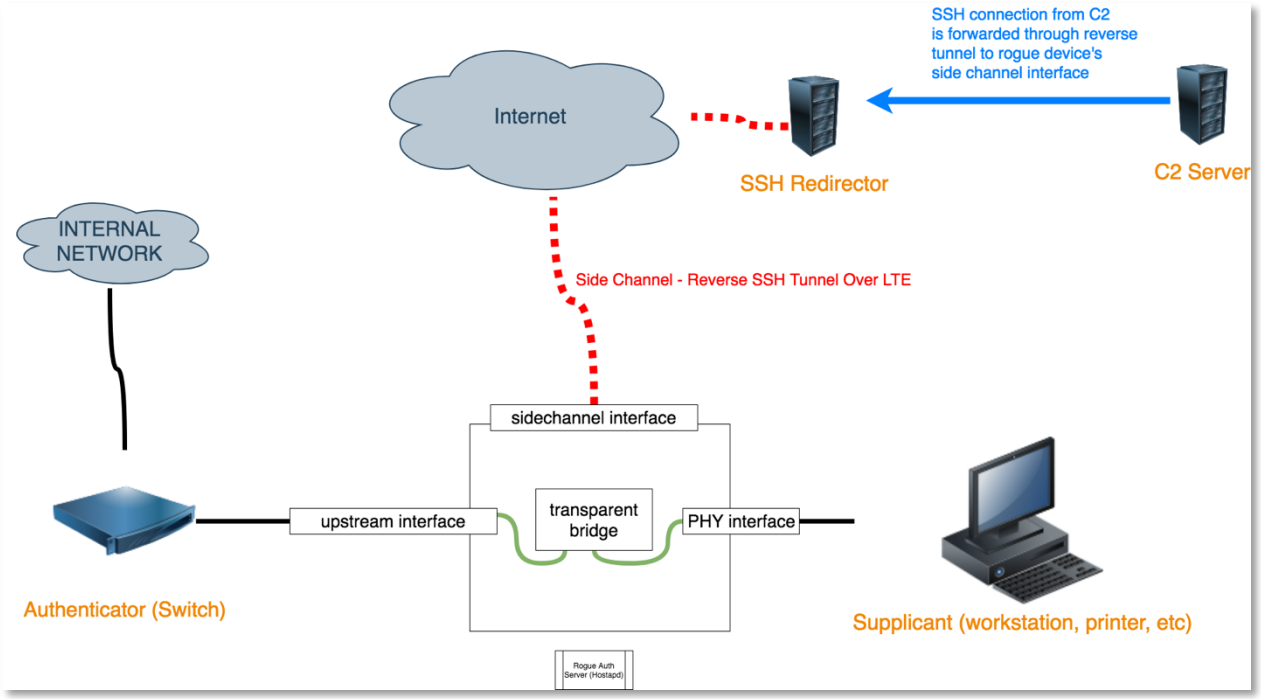


Figure 31

Once we establish that the supplicant is configured to use EAP-PEAP, EAP-TTLS, or any similarly weak EAP implementation that does not enforce mutual certificate-based validation, we bring down our bridge and upstream network interfaces, as shown in *Figure 32* below. We then start hostapd as a rogue RADIUS server and have it listen our PHY network interface.

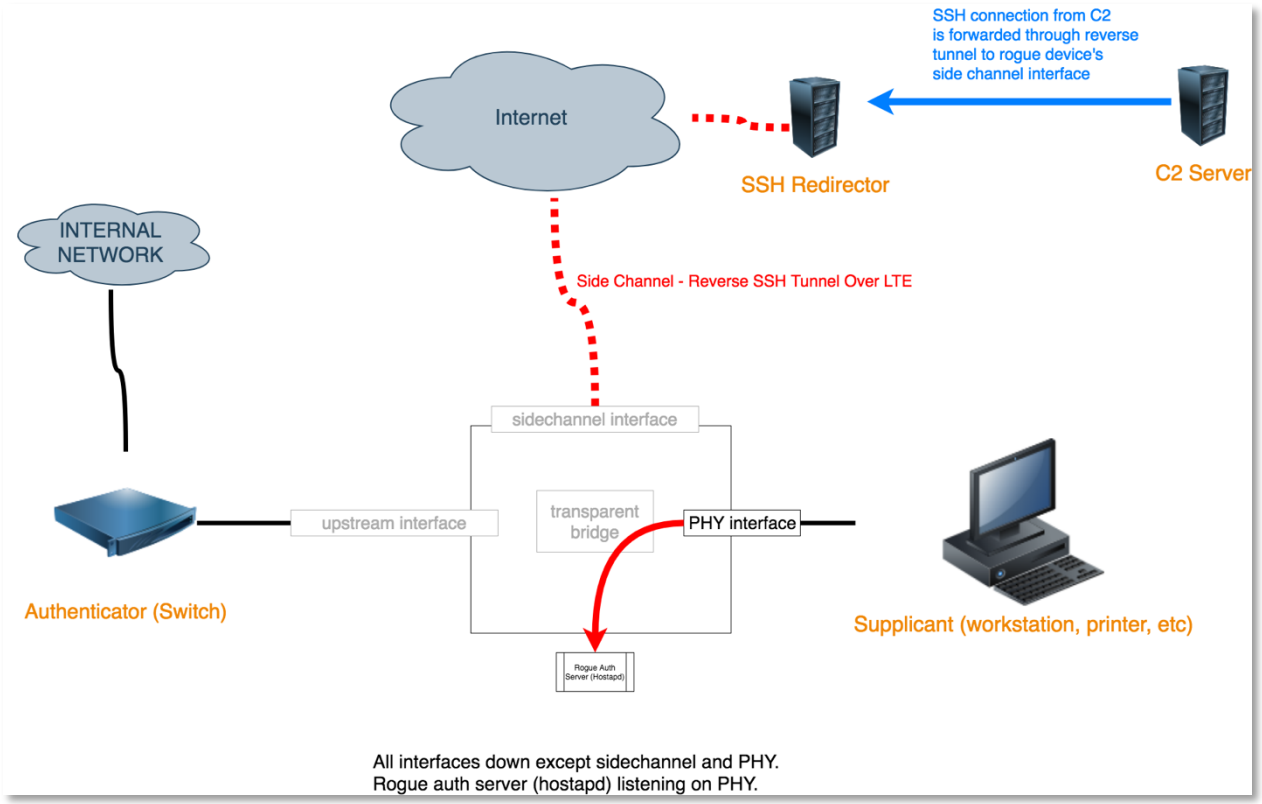


Figure 32

We then send a spoofed EAPOL-start frame to hostapd, causing hostapd to send an EAP-Request-Identity frame to the supplicant (see: VII.1 EAP-MD5 Forced Reauthentication Attack).

This causes the supplicant to authenticate with the rogue device. As long as the authorized client accepts the rogue device's x.509 certificate, the attacker will succeed in capturing an MS-CHAPv2 challenge and response which can be used to obtain plaintext credentials.

Finally, we connect the rogue device to the network using a Bait n Switch.

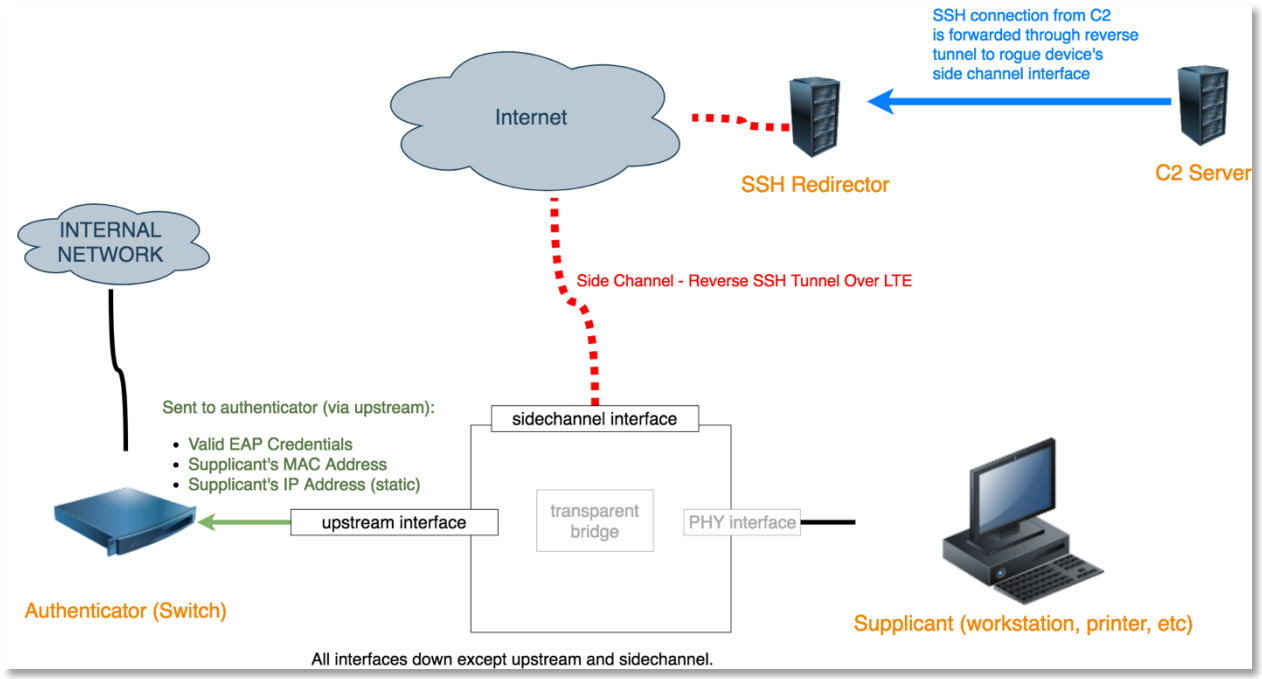


Figure 33

VIII. Proof of Concept and Source Code Release

We have packaged each of the attacks described in this document into a tool called *silentbridge*, which can be downloaded at the following URL:

- https://github.com/s0lst1c3/silentbridge

The silentbridge repository contains both source code and documentation that describes how to build each of the rogue devices described in this paper.

Conclusion

While 802.1x-2010 is a remarkable improvement over 802.1x-2004, weak authentication protocols can undermine the enhancements provided by MACsec. Even though 802.1x-2010's use of MACsec successfully prevents the bridge and injection-based attacks introduced by "Abb" and Alva Duckwall, the protocol's reliance on EAP means that the standard is only as secure as the EAP methods deployed. This is demonstrated by the effectiveness of the Rogue Gateway and Bait n Switch attacks introduced in this document.

If parallels between MACsec and WPA2 are any indication, we can expect the use of weak EAP implementations to become more and more prevalent as adoption rates for 802.1x-2010 increase. This is especially true considering that the 802.1x-2010 standard does not mandate the use of strong EAP methods.

Additionally, although improved 802.1x support by peripheral device manufacturers is a step in the right direction, it is not enough. We need to incentivize device manufacturers to create products that both support strong EAP implementations and make it easy for organizations to deploy them. Until then, peripheral devices will largely be susceptible to the Rogue Gateway, Bait n Switch, and EAP-MD5 Forced Reauthentication attacks introduced in this document. Furthermore, these devices will remain susceptible to the bridge and injection based 802.1x bypasses introduced by Duckwall and "Abb" until both adoption of and support for 802.1x-2010 becomes widespread.

Finally, it is important to recognize that while the use of 802.1x port security should still be considered an industry best-practice, it is not a substitute for a layered approach to network security. The use of 802.1x, or any other form of access control for that matter, should not be considered a mitigation for other host and network level security issues. Deploying 802.1x is not a substitute for good patch management practices, nor should it be used to justify the use of dangerous networking protocols such as LLMNR.

Acknowledgements

Special thanks to Dan Nelson, Justin Whitehead, and Ryan Jones for helping make this project a reality.

References

- [1] http://www.ieee802.org/1/pages/802.1x-2001.html
- [2] http://www.ieee802.org/1/pages/802.1x-2004.html
- [3] https://blogs.technet.microsoft.com/steriley/2005/08/11/august-article-802-1x-on-wired-networks-considered-harmful/
- [4] https://www.defcon.org/images/defcon-19/dc-19-presentations/Duckwall/DEFCON-19-Duckwall-Bridge-Too-Far.pdf
- [5] https://www.gremwell.com/marvin-mitm-tapping-dot1x-links
- [6]https://hackinparis.com/data/slides/2017/2017_Legrand_Valerian_802.1x_Network_Access_Control_and_Bypass_Techniques.pdf
- [7] https://www.cisco.com/c/en/us/products/collateral/ios-nx-os-software/identity-based-networking-services/deploy_guide_c17-663760.html
- [8] https://1.ieee802.org/security/802-1ae/
- [9] https://standards.ieee.org/findstds/standard/802.1X-2010.html
- [10] http://www.ieee802.org/1/files/public/docs2013/ae-seaman-macsec-hops-0213-v02.pdf
- [11] https://www.gremwell.com/linux_kernel_can_forward_802_1x
- [12] https://www.intel.com/content/www/us/en/support/articles/000006999/network-and-i-o/wireless-networking.html
- [13]http://www.willhackforsushi.com/presentations/PEAP_Shmoocon2008_Wright_Antoniewicz.pdf
- [14] https://link.springer.com/content/pdf/10.1007%2F978-3-642-30955-7_6.pdf
- [15] https://support.microsoft.com/en-us/help/922574/the-microsoft-extensible-authentication-protocol-message-digest-5-eap
- [16] https://tools.ietf.org/html/rfc3748
- [17] https://code.google.com/archive/p/8021xbridge/source/default/commits
- [18] https://github.com/mubix/8021xbridge
- [19] https://hal.inria.fr/hal-01534313/document
- [20] https://sensepost.com/blog/2015/improvements-in-rogue-ap-attacks-mana-1%2F2/
- [21] https://tools.ietf.org/html/rfc4017
- [22] http://web.archive.org/web/20160203043946/https:/www.cloudcracker.com/blog/2012/07/29/cracking-ms-chap-v2/
- [23] https://crack.sh/
- [24] https://tools.ietf.org/html/rfc5216
- [25] https://4310b1a9-a-93739578-s-sites.googlegroups.com/a/riosec.com/home/articles/Open-Secure-Wireless/Open-Secure-Wireless.pdf?attachauth=ANoY7cqwzbsU93t3gE88UC_qqtG7cVvms7FRutz0KwK1oiBcEJMlQuUmpGSMMD7oZGyGmt4M2HaBhHFb07j8Gvmb_HWIE8rSfLKDvB0AI80u0cYwSNi5ugTP1JtFXsy1yZn8-85icVc32PpzxLJwRinf2UGzNbEdO97Wsc9xcjnc8A8MaFkPbUV5kwsMYHaxMiWwTcE-A8Dp49vv-tmk86pNMaeUeumBw_5vCZ6C3Pvc07hVbyTOsjqo6C6WpfVhd_M0BNW0RQtI&attredirects=0
- [26] https://txlab.wordpress.com/2012/01/25/call-home-ssh-scripts/
- [27] https://txlab.wordpress.com/2012/03/14/improved-call-home-ssh-scripts/
- [28] https://www.cisco.com/c/en/us/td/docs/switches/lan/catalyst3560/software/release/12-2_37_se/command/reference/cr1/cli3.html#wp1948361
- [29] https://www.juniper.net/documentation/en_US/junos/topics/concept/port-security-persistent-mac-learning.html
- [30] https://tools.ietf.org/html/rfc3579
- [31] https://tools.ietf.org/html/rfc5281