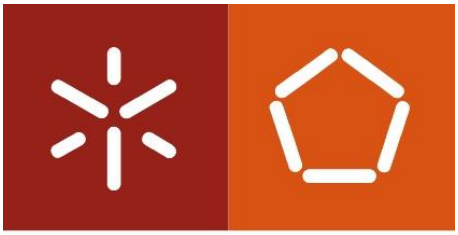

Universidade do Minho

Departamento de Informática

Mestrado em Engenharia Informática

Engenharia da Internet

Gestão e Segurança de Redes
Ficha de trabalho Nº2 - SNMPv2cSec

Luís Carneiro (PG46541)

27 de junho de 2022

Resumo

O presente documento descreve sucintamente os objetos de avaliação e de análise ao longo das duas fichas de trabalho prático inseridas na unidade curricular Gestão e Segurança de Redes, pertencente ao perfil de Engenharia da Internet inserido no Mestrado em Engenharia Informática da Universidade do Minho. O objetivo é, de uma forma resumida, garantir o mesmo nível de segurança do SNMPv3 (garantia de privacidade, autenticação e verificação da integridade) ao utilizar o SNMPv2c para comunicar com um agente.

Conteúdo

1 Discussão das estratégias escolhidas **2**

1.1 Comunicação entre as entidades envolvidas 2

1.1.1 Estrutura de um pacote SNMP 2

1.2 Mecanismos comportamentais 3

2 Explicação e definição da MIBsec **4**

2.0.1 Estrutura da MIBsec 4

2.0.2 Notas sobre a estrutura 4

3 Explicação da maneira segura de manipular objetos das MIBs **6**

3.0.1 Simulação de um pedido get pelo manager 6

4 Apresentação das principais funções/classes **8**

4.1 Manager a efetuar um pedido 8

4.2 Handling dos pedidos do manager pelo proxy 10

5 Conclusão **12**

6 Referências **13**

Capítulo 1

Discussão das estratégias escolhidas

1.1 Comunicação entre as entidades envolvidas

A comunicação entre o manager e o proxy foi bastante simplificada, pois em vez de a transmissão ser feita com o SNMP e codificação BER, foi implementada de raíz com uma estrutura de dados que representa apenas os campos necessários para realizar as operações de leitura dos agentes (GET e GETNEXT) através de UDP.

1.1.1 Estrutura de um pacote SNMP

Nesta secção será apresentada brevemente a estrutura de um pacote SNMP (alguns campos, como o oid e o valor, serão melhor explicados na secção 3.0.1).

Um pacote SNMP é composto por:

- um id inteiro com 4 dígitos (o seu limite inferior é 1000 e o superior é 9999);
- uma *community string*;
- um tipo de pacote representado por um inteiro (GET é igual a 0, GETNEXT é igual a 1 e SET é igual a 2 para os pedidos do manager e RESPONSE é igual a 3 para as respostas do proxy);
- uma string que representa o oid;
- uma string que representa o valor (abstrato);
- uma string que representa o manager que efetua o pedido;
- uma string que representa o agente que responde ao pedido;

O manager e o agente são destacados em campos separados pois simplificam bastante os pedidos do manager, visto que desta forma o manager não tem de fazer pedidos separados para os identificar na MIBsec; possibilita ainda o manager de verificar se a resposta recebida pelo proxy tem, de facto, os agentes e managers originais.

O valor é cifrado com uma cifra simétrica, cuja chave secreta é distribuída através de um ficheiro de configuração colocado no diretório dos ficheiros do projeto, pelo que a privacidade é garantida. Tanto a autenticação como a verificação da integridade deste campo são efetuadas implicitamente pois, à partida, apenas o manager x tem a chave y, e caso um ataque Man-in-the-Middle seja efetuado e o valor do pacote for alterado, como vai ser cifrado com outra chave, o proxy não vai conseguir decifrar o seu conteúdo e, consequentemente, vai detetar o ataque.

1.2 Mecanismos comportamentais

De modo a impedir que atacantes externos se façam passar por um manager legítimo indefinidamente, o proxy não responde aos pedidos de um manager se:

- a chave utilizada para cifrar o campo valor de um pacote SNMP for incorreta;
- o tipo identificado no pacote SNMP não for nem GET, nem GETNEXT nem SET.

Capítulo 2

Explicação e definição da MIBsec

2.0.1 Estrutura da MIBsec

A estrutura da MIBsec é bastante semelhante à do segundo teste escrito. A sua estrutura é:

- idOper – identificador inteiro da operação recebida pelo agente proxy (número aleatório e unívoco, gerado pelo gestor e verificado pelo proxy); este objeto serve de chave da tabela; tem 5 dígitos, portanto o seu limite inferior é 100000 e o superior é 99999;
- typeOper – tipo de operação SNMP (get, getnext); a identificação é feita através dum número inteiro (0 para o get e 1 para o getnext);
- idSource – identificador string da fonte do pedido (um alias/nome que identifica um gestor);
- idDest – identificador string do destino onde a operação será executada (um alias/nome que identifica um gestor SNMPv2c);
- oidArg – OID em string do objeto da MIB que é argumento da operação a ser executada no agente SNMPv2c remoto;
- valueArg – valor em string do objeto referido por oidArg que é o resultado recebido no agente proxy vindo do agente SNMPv2c remoto;
- sizeArg – tamanho em bytes do valueArg; enquanto o valor de valueArg não é devolvido pelo agente SNMPv2c remoto, sizeArg deve ser igual a zero; quando o agente proxy recebe a resposta do agente SNMPv2c remoto grava o valor da instância do objeto em valueArg e coloca sizeArg com o tamanho desse valor (em bytes);
- statusOper - do tipo inteiro, assume o valor 0 quando o objeto não está completamente definido na tabela (falta o OID), 1 quando está (e está à espera da resposta), 2 quando ocorreu um erro ao fazer query, e 3 quando o objeto está válido;
- requestTimestamp - do tipo Date, marca o timestamp que o manager colocou o pedido na tabela;
- ttlOper - número inteiro que identifica por quanto tempo em segundos o pedido do agente é válido;

2.0.2 Notas sobre a estrutura

Infelizmente, devido a limitações do PySNMP (biblioteca usada para fazer queries SNMP ao computador local), não foi possível averiguar o tipo do objeto guardado no valueArg, pelo que é sempre assumido que é uma string.

Foram acrescentados os campos da statusOper, requestTimestamp e ttlOper de modo a, periodicamente, limpar a tabela de operações que tenham tenham sido efetuadas há mais de ttlOper segundos. Desta forma, é possível limitar o crescimento da tabela à medida que os pedidos são efetuados. A statusOper ajuda ainda ao preenchimento da tabela, visto que a thread que a preenche só o faz se o objeto estiver definido na sua totalidade (e já não tiver sido preenchido).

Capítulo 3

Explicação da maneira segura de manipular objetos das MIBs

3.0.1 Simulação de um pedido get pelo manager

O pedido get do manager tem que ter a estrutura:

```
GET [OID] [alias_agente]
```

Devido a limitações do PySNMP, o OID apenas pode ser dado em formato numérico. Por exemplo, para fazer query do sysDescr, a query é da seguinte forma:

```
GET .1.3.6.1.2.1.1.1.0 agente1
```

O alias do manager pode ser dado opcionalmente como um argumento na linha de comandos ao executar o programa (se não for, é assumido o alias "manager1") e é inserido automaticamente nos pedidos efetuados ao proxy (mais especificamente é inserido no pacote SNMP usado para comunicar entre eles).

Depois da validação do input inserido, o manager vai efetuar dois pedidos do tipo SET antes de poder ler o resultado com um pedido GET.

Como já referido anteriormente, para os pedidos SET efetuados vai ser indicado se a operação ocorreu com sucesso ou não no campo valor do pacote SNMP (o erro comum a estes pedidos é se o OID estiver mal formatado). O pedido GET recebe tanto a mensagem de erro como o objeto lido nesse campo.

Adicionalmente, todos os campos têm um ID do pacote aleatório.

Primeiro pedido SET

O primeiro pedido SET vai ter um OID de "TableReq.typeOp.145"e um valor (cifrado) "GET", que vai colocar na tabela do proxy uma nova entrada com o ID 145, tipo de operação GET (valor decifrado), o alias do manager igual a manager1 e o alias do agente igual a agente1.

Na resposta, recebe no campo do valor o resultado da operação: se for com sucesso, segue para o próximo pedido SET; se o ID da tabela 145 já existir, tenta novamente com um ID diferente; caso

contrário, aborta a operação.

Todos os restantes pedidos vão ter na última coluna do OID o mesmo ID (145) deste primeiro (depois de confirmação pelo proxy que não está em uso).

Segundo pedido SET

O segundo pedido SET vai ter um OID de "TableReq.oidArg.145"e um valor (cifrado) ".1.3.6.1.2.1.1.1.0", ou seja, vai colocar na tabela do proxy na mesma linha definida anteriormente o OID referido.

Na resposta, recebe no campo do valor o resultado da operação: se for com sucesso, segue para o próximo pedido GET; caso contrário, aborta a operação.

Pedido GET

Finalmente, o pedido GET é efetuado, com o OID pedido, e o resultado é mostrado no ecrã, quer seja um erro ou o valor pedido.

Para um pedido GETNEXT, os passos são praticamente iguais, o que muda é o tipo de pedido presente no pacote SNMP e depois o proxy efetua o pedido GETNEXT ao respetivo agente.

Capítulo 4

Apresentação das principais funções/classes

Devido à já existente explicação do código na sua documentação auto-contida através de comentários e na exemplificação dada ao pormenor na secção 3.0.1, o código a seguir vai ser meramente apresentado.

4.1 Manager a efetuar um pedido

```
        def send_req_recv_reply(req:str, my_manager_alias:str,
        ↪  received_operation_id:str=None): # pdu_t:PDUType, pdu_s:str,
        ↪  manager:str, agent:str
            if DEBUG_FLAG==1:
                    print("A fazer parse do pedido " + req)
            succ, request_type, real_oid, agent_alias = parse_request(req)
            if succ == False:
                    print("Erro ao fazer parse do pedido " + req)
                    return

            # Os pedidos para definir os 5 parâmetros
            # (id operação, tipo, idSource, idDest e oidArg)
            # vão ser do tipo SET
            generic_set_type = PacketType.SET_REQUEST.value

            # ID da operação é escolhido aleatoriamente no início
            # depois é sempre o mesmo
            # (se o ID for None, estamos a realizar esta operacao pela primeira vez
            # caso contrário, já obtivemos ID inválido na chamada passada)
            if received_operation_id is None:
                    operation_id_str = str(RequestsTable.get_random_operation_id())
            else:
                    operation_id_str = received_operation_id

            # OID para definir o tipo de operação é tipo TableReq.typeOp.145
            packet_oid = RequestsTable.name + "." +
            ↪  RequestsTableColumn.TYPE_OPER.value + "." + operation_id_str

            packet_to_send = create_packet_to_send(
                    packet_type=generic_set_type,
```

```
        packet_oid=packet_oid,
        value=request_type,
        manager=my_manager_alias,
        agent=agent_alias
)

status = ResponseStatus.ID_ALREADY_EXISTS.value
while status == ResponseStatus.ID_ALREADY_EXISTS.value:
        # Enviar pedidos continuos até obter resposta
        status = send_requests_till_answer(packet_to_send)
        #response = SNMPPacket.proxy_response_to_message(code_response)
        #print("Proxy respondeu: " + response)

        #  SUCCESS INVALID_OID INVALID_TYPE ID_ALREADY_EXISTS
        if status is None: # Nº pedidos excedido
                return
        if status != str(ResponseStatus.SUCCESS.value):
                print(SNMPPacket.response_status_to_message(status))
                # Se ID já existir, temos de ler o sugerido e tentar
                ↪  outra vez...
                if status ==
                ↪  str(ResponseStatus.ID_ALREADY_EXISTS.value):
                        if received_operation_id is None:
                                if DEBUG_FLAG==1:
                                        print("ID inválido, a tentar
                                        ↪  outra vez!")
                                send_req_recv_reply(req,
                                ↪  my_manager_alias, value)
                return

# 2o pedido - definir o oid de operação (145)

# OID para definir o tipo de operação é tipo TableReq.typeOp.145
packet_oid = RequestsTable.name + "." +
↪  RequestsTableColumn.OID_ARG.value + "." + operation_id_str

packet_to_send = create_packet_to_send(
        packet_type=generic_set_type,
        packet_oid=packet_oid,
        value=real_oid,
        manager=my_manager_alias,
        agent=agent_alias
)

# Enviar pedidos continuos até obter resposta
status = send_requests_till_answer(packet_to_send)
#response = SNMPPacket.proxy_response_to_message(code_response)
#print("Proxy respondeu: " + response)
# SAME_OID_ALREADY_EXISTS DIFFERENT UNAUTHORIZED
if status is None: # Nº pedidos excedido
        return
if status != str(ResponseStatus.SUCCESS.value):
        print(SNMPPacket.response_status_to_message(status))
        return

# dar algum tempo ao proxy de definir o valor na tabela...
sleep(SET_TO_GET_DELAY)
```

```
        packet_oid = RequestsTable.name + "." +
        ↪  RequestsTableColumn.VALUE_ARG.value + "." + operation_id_str

        packet_to_send = create_packet_to_send(
                packet_type=request_type,
                packet_oid=packet_oid,
                value=None,
                manager=my_manager_alias,
                agent=agent_alias
        )
        value = send_requests_till_answer(packet_to_send)
        if value is None: # Nº pedidos excedido
                return
        print(value)
```

4.2 Handling dos pedidos do manager pelo proxy

```
    def handle_manager_request(request:SNMPPacket) -> bytes:
""" Recebe um pacote SNMP e devolve a resposta
    (Pacote SNMP para bytes) """

respond_flag = True # por norma, é para responder
# exceto quando a chave de decifra está inválida
response:str = None

packet_oid = request.object_identifier
manager_alias = request.manager

# Ver se o manager tem uma chave guardada
manager_secret_key = get_manager_key_from_alias(
    manager_alias
)
if manager_secret_key is None:
    print("Chave do manager " + manager_alias + " nao guardada...")
    respond_flag = False
elif manager_alias in global_manager_blacklist:
    print("Manager " + manager_alias + " na blacklist...")
    respond_flag = False
else:
    table_id,table_column = get_table_id_and_column_from_oid(packet_oid)
    if request.packet_type == PacketType.SET_REQUEST.value:
        if table_id is None:
            # oid mal formatado...
            print("Erro de formatacao no oid " + packet_oid)
            response = str(ResponseStatus.INVALID_TABLE_OID.value)
        else:
            # ver se o value fornecido no pedido é válido
            # (tentar decifrá-lo)
            decrypted_value_bytes = CryptoOperation.aes_decryption(
                request.value, manager_secret_key)
            if decrypted_value_bytes is None:
                # Nao vai responder por questoes de segurança...
                # Se for um atacante, a informaçao que a chave é inválida
                # pode ser benéfica
                respond_flag = False
```

```
                    global_manager_blacklist.append(manager_alias)
                    print("Chave inválida...")
                else:
                    decrypted_value = decrypted_value_bytes.decode()
                    #print("Valor do pedido: " + decrypted_value)

                    response = save_request_in_table(table_id, table_column,
                    ↪  decrypted_value,
                    manager_alias, request.agent)
                    if response == str(ResponseStatus.ID_ALREADY_EXISTS.value):
                        # fornecer outro ID para o manager tentar outra vez
                        response = str(generate_unused_table_id())
                        print("NOVO ID: " + response)

        elif request.packet_type==PacketType.GET_REQUEST.value or
        ↪  request.packet_type==PacketType.GET_NEXT_REQUEST.value:
            # buscar valor à tabela
            response = get_object_from_table(table_id)
        else:
            respond_flag = False
            global_manager_blacklist.append(manager_alias)
            print("Tipo de pedido invalido...")


    if respond_flag:
        response_snmp_packet = SNMPPacket(
            packet_id=request.packet_id,
            comm_str=COMM_STRING,
            packet_type=PacketType.RESPONSE.value,
            oid=None,
            value=response,
            manager=manager_alias,
            agent=request.agent,
            secret_key=manager_secret_key,
        )
        #print("A responder: " + str(response.value))
        response_bytes = response_snmp_packet.convert_to_bytes()
        UDPCommunication.send_UDP(response_bytes, SEND_TO_PORT)
```

Capítulo 5

Conclusão

Para concluir, gostaria de poder procurar os objetos com o PySNMP pelo nome e descobrir o seu tipo, permitir ao manager editar os objetos de um agente, e os mencionados no enunciado, nomeadamente introduzir mais mecanismos comportamentais, mecanismos de controlo de acesso definíveis para cada manager (sem usar community strings) e implementar uma troca contínua das chaves usadas pelos managers para aumentar a segurança das comunicações efetuadas.

Capítulo 6

Referências

- https://pysnmp.readthedocs.io/en/latest/quick-start.html