

Exploring DNS Security

Focused and Deliciously Witty Subtitle

J. BERRETTA, J. CAPUDER, E. GUENDEL

Oberlin College

December 15, 2016

Abstract

The security of common internet protocols is an important issue when considering how to satisfy the requirements of security and privacy as technology becomes more integral to our daily lives. The components required to simulate a DNS lookup were implemented in Java. This simulation was used to attempt a common DNS cache poisoning attack, and explore methods for improving the security of DNS protocol.

Introduction

One of the most important requirements of any communication network, such as the internet or the United States Postal Service, is the ability to uniquely identify and locate participants of the network. This requirement is often met by assigning some form of address to participants: to determine the location to which an outgoing letter should be delivered, the sender must specify country of the recipient, what part of the country they live in, and finally the specific street/number combination at which the recipient resides. This system, however, was designed for human comprehension, and could be more concisely represented by numerical coordinates if readability was not an issue. A similar issue arises when considering the numerical Internet Protocol addresses (IP addresses) used to identify computers and

resources connected to the internet: IP addresses are convenient for use in networking protocols, but are difficult for humans to memorize. The Domain Name System (DNS) is in place to allow individuals or companies to register readable domain names (www.google.com) that can be mapped to the IP addresses needed to locate certain computers on the network, and is fundamental to the functionality of the internet.

The specific DNS protocol we examine in this report is referred to as a client lookup, as this protocol is vulnerable to an attack known as cache poisoning. A client lookup is initiated whenever an application on a client's computer sends a request that requires a domain name to be translated to an IP address. The application will first send a request to a domain name resolver, which will check its cache of recent lookups for a response to the request. If the cache contains an appropriate response, the resolver will return it to the sender of the request; if not, the resolver will send the request to the root name server. The root name server will check its cache, and either return a cached response, or tell the DNS resolver the location of the next name server it should query. This process is repeated until the resolver receives a final response, which it will cache and return to the client.

The ultimate goal of a cache poisoning attack is to trick the domain name resolver into caching an incorrect IP address, creating a false association between a domain name, like "www.google.com," and the location of the domain's servers. This in turn can divert users who try to visit the domain name in question to a malicious website. The process by which this attack is carried out begins with an attacker transmitting queries to the domain name resolver. The attacker then sends responses to the resolver that ordinarily come from name servers, but these responses have IP addresses that do not correspond to the domain name requested. The resolver will accept and cache these responses. Anyone who attempts to visit domains with compromised IP's before the cached responses expire will be directed to a computer of the attacker's choice.

Implementation

Node

We realized quite early in the implementation that weren't going to be able to use methods with returns statements as a way to communication between servers. This is because there's no real way to intercept a return statement. The assignment of a variable to a non-void function will always result in whatever that particular function returns. The entirety of the simulation is based on forging messages (from an attacker to a DNS server), and as such we used the idea of calling another object's method with certain parameters as a way of sending information.

This is why the attacker, the client, the DNS server and the name servers all implement the Node interface, which contains:

```
public void message(Node src, Message message);  
public String getAddress();
```

getAddress is a helper function just used for checking the source of a response, which will be explained below. The message function, is utilized as follows. If *server₁* wants to talk to *server₂* and send them a message msg, the following code would be called by *server₁*:

```
server_2.message(this, msg);
```

Whatever the destination does with the message depends on the implementation, but this basically enables a race. In the case of the attacker and the name server, whomever calls the DNS' message function first and gets its answer accepted successfully caches that answer, be it malicious or not.

Message

The message class encapsulates both a request and response into one object. What it acts as depends on the context and its construction. Its basic usage is to be able to send information across all of the nodes (the attacker, client and all the servers). There are multiple constructors for different purposes but the general class variables are these:

```

private String type;
private Url query;
private String TXID;
private String answer;
private NameServer nextServer;

```

The type is especially important as it determines the purpose of the message. A message of type *WHERE* is recognized by the DNS server as a request from a client. A *TRY* message is also recognized by the DNS as a message which contains the next relevant name server to query about the given URL. A *FINAL* message is received by the DNS from a name server and then passed onto the client; this message type contains the IP address of the URL.

The query contains the URL as well as other components. The TXID is used as a method of checking the validity of the message, which is explained later on. The answer is the IP address (for *FINAL* types) and *nextServer* is used to store the next name server the DNS should query (for *TRY* types).

Client

The client is an important piece in the implementation, since it's essentially the target of the attack. The main objective is to have the DNS return a malicious IP from its cache when it receives a request from the Client. As such, the Client class implements the Node interface to be able to communicate with the DNS server through the message() function calls.

The following is the class variables and the constructor for a Client object.

```

private static final String TAG = "Client";
private DNS dns_;
private Cache cache_;

public Client(DNS dns) {

```

```

        this.dns_ = dns;
        this.cache_ = new Cache();
    }

```

The tag is included strictly for logging purposes. The constructor takes in a DNS server, as it should know, at the moment of its instantiation, what server it's going to refer to for requests. The cache is initialized right away and similarly to the DNS server, functions to avoid requesting a web address' IP it has recently queried.

The following is the method we use to send a message to the DNS.

```

public void makeRequest(Url query){
    // If not in cache, send request and wait for
    // reply
    Message request = new Message(query);
    dns_.message(this, request);
}

```

This takes in a Url object, which contains the web address, encapsulates it in a Message object as a request, and calls the DNS' receive method, passing the request as a parameter. Note that the message method and the Message class are two different components of the implementation. One is used to package the information, the other to send/receive it.

This leads me to the Client's implementation of the message method, following is an excerpt:

```

// If we already have an entry in our cache, then we
// ignore any received messages
if (!cache_.isEmpty()){
    return;
}
// Check that this is indeed a final answer from the
// DNS
if (message.getType() == MessageType.FINAL){
    // Add entry to cache
}

```

```

        Log.i(TAG, "Adding the following to cache: " +
            "{" + message.getQuery().toString() + " : " +
            message.getAnswer() + "}");
        cache_.addEntry(message.getQuery(),
            message.getAnswer());
    }
    else {
        return;
    }
}

```

Once the DNS server has an answer for us, itâ€™ll call this method. The message type will be FINAL since it contains the IP address that the client desires. Any other message types such as TRY and WHERE, which are meant only for the DNS, will be ignored. Once that message is received, we add the entry to our cache and log the event.

All of these pieces are used together in visitWebPage, which returns the corresponding IP of a URL.

```

public String visitWebPage(Url query){
    String address = cache_.lookupEntry(query);
    while (address == null){
        makeRequest(query);
        address = cache_.lookupEntry(query);
    }
    return address;
}

```

Here we consistently check the cache to see if we have an entry for the corresponding query, making a new request every time we donâ€™t. Eventually, once the DNS gets an answer back from the name servers or looks it up in its own cache, itâ€™ll call the Clientâ€™s message function, which will put the answer in the cache and cause the function above to break out of the while loop and return the address.

0.1 DNS

The DNS is the main tool for the attack and the server whose cache gets poisoned. It operates in many ways similarly to the Client class, except that its message function is more sophisticated because of the interactions with the name servers.

These are its class variables:

```
private static final String TAG = "DNS";
private NameServer root_;
private Cache cache_;
private Node client_;
private Map<String, Integer> expectedServers_;
private Set<String> TXIDS_;
private final ReentrantLock lock_;
```

The tag is for logging purposes, which allows us to track the progress of the simulation. The next three variables, a root name server, a cache and a client are all essentials for the basic functionality. We require a root name server to send requests to, who respond with either the IP address or a reference to the another name server to ask. The cache is required to be poisoned with a malicious address that is sent to the client. The client instance is required to send this malicious address back. Of these three, only the root name server is required as a parameter for initialization, since the DNS must know who to initially ask about a URL. The client could be anyone trying to communicate with the server, but we'll see how this is assigned later.

The following two variables are used to ensure the legitimacy of name server responses. `expectedServers` is a hashmap which is used to keep track of the name server from whom we are expecting a response following a request. The string is the address of such server and the integer is used to keep count of how many responses we expect from that server. This is done to avoid an attacker to flood the DNS with responses that have not been requested. While this is used to verify the source of the response, `TXIDS` is used to verify the validity of the response itself. This set contains

TXIDs which are created at the moment the DNS receives a request from a client and included in the message itself. Once the final response arrives from a name server with the address, the TXID in the message is checked against the set, and only accepted if it exists. What this means is that the attacker only has to be able to guess one TXID within the set to have its response be cached. Additionally, it will have to spoof the address, in order to pretend like it's coming from a legitimate name server, which we assume the attacker knows. Finally, the ReentrantLock is used to protect the TXID set from one thread attempting to traverse it while another is editing it. This lock uses queue based priority to ensure thread safety.

Following is the excerpt within message that handles a client's request:

```
if (message.getType()==MessageTypes.WHERE){
    Log.i(TAG, "Received a message of type: " +
        message.getType());
    String TXID = Txid.genTxid();
    lock_.lock();
    while (TXIDS_.contains(TXID)){
        TXID = Txid.genTxid();
    }
    TXIDS_.add(TXID);
    lock_.unlock();
    message.setTXID(TXID);
    this.client_ = src;
// Now expecting a message back from root
    addToExpected(root_.getAddress());
    root_.message(this, message);
}
```

As can be seen, we also ensure that every single client request gets assigned a unique TXID by avoiding creating one that's already in the set - all while maintaining thread safety. Since we know it was a client request due to the message type (WHERE), we assign the class variable of client to the source, effectively granting us the ability to contact it back.

Lastly, we send a request to the root name server, to which we simply pass on the message given by the client, with the only modification being the addition of the TXID. The root server is also added to the expected server hashmap, as to accept its response.

The next message type, TRY, is the name server's response which refers to another name server to query. Once a TRY message goes through the two security checks (TXIDS and expectedServers), this occurs:

```
NameServer nextServer = message.getNextServer();
// Got message back from expected server, add next to
  the map
decrementExpected(src.getAddress());
addToExpected(nextServer.getAddress());
nextServer.message(this, new
    Message(message.getQuery(), message.getTXID()));
```

The response contains the next server, which is extracted. The address of the server we just received from is subtracted from the expected count, and the next name server's address is added to the expected. Finally, a new request is constructed and sent to the next name server.

Once we get a FINAL message, the DNS has completed its objective:

```
decrementExpected(src.getAddress());
this.cache_.addEntry(message.getQuery(),
    message.getAnswer());
this.client_.message(this, message);
lock_.lock();
TXIDS_.remove(message.getTXID());
lock_.unlock();
```

We decrement the count of expected messages from that particular server and cache the response. The final answer is sent to the client immediately after the caching and the TXID is removed from the set, as it has served its purpose.

So what happens if the client request an url that is already in the cache?

```
if (message.getType() == MessageTypes.WHERE) {  
    Url query = message.getQuery();  
    Message toSend = new Message(query,  
        cache_.lookupEntry(query), message.getTXID());  
    src.message(this, toSend);  
} else {  
    return;  
}
```

The first check is to make sure that this is coming from client, and if itâ€™s not, simply return. If it is, obtain the query and extract the address from the cache before packaging it into a response object (Message) and sending it back.

Conclusion

Fusce in nibh augue. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. In dictum accumsan sapien, ut hendrerit nisi. Phasellus ut nulla mauris. Phasellus sagittis nec odio sed posuere. Vestibulum porttitor dolor quis suscipit bibendum. Mauris risus lectus, cursus vitae hendrerit posuere, congue ac est. Suspendisse commodo eu eros non cursus. Mauris ultrices venenatis dolor, sed aliquet odio tempor pellentesque. Duis ultricies, mauris id lobortis vulputate, tellus turpis eleifend elit, in gravida leo tortor ultricies est. Maecenas vitae ipsum at dui sodales condimentum a quis dui. Nam mi sapien, lobortis ac blandit eget, dignissim quis nunc.

1. First numbered list item
2. Second numbered list item