DNS Antidote

Abhishek Madav
University of California, Irvine
amadav@uci.edu

Suhas Tikoo
University of California, Irvine
stikoo@uci.edu

Urjit Khadilkar
University of California, Irvine
ukhadilk@uci.edu

I. ABSTRACT:

The Domain Name System (DNS) is one of most critical foundational technologies on which the IP-driven Internet functions. This fact also makes it one of those technologies which are often targeted by cyber attackers. DNS Poisoning is one such attack wherein an attacker controls the DNS Resolving Server and provides the user with an incorrect resolution of the requested domain name. The user is thus directed to a different website as a consequence of the erroneous resolution. We have devised a service, DNS Antidote, which would monitor the resolution of the user's requests across different browsers. HTTP links accessed by the user would be checked for potential phishing against a DNS lookup performed by a secured web service. The service responds by enumerating the list of the IP addresses for the query which can then be compared with the local look-up for making a secured redirection if required.

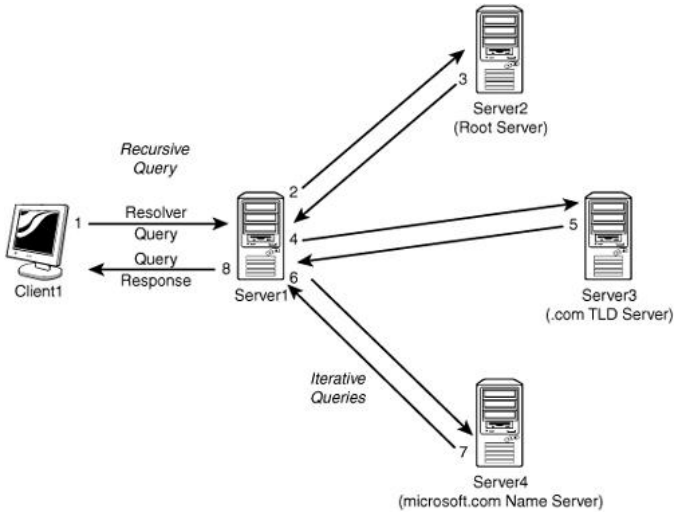
Key Terms: *DNS, DNS poisoning, Web Service, IP addresses, Cyber Attacks, REST, AWS*

II. MOTIVATION AND RELATED WORK

The Domain Name System is one of those unsung heroes of the Internet without which the internet would collapse. Today's TCP/IP based Internet works on IP addresses and while users continue to use the popular domain names to access websites, the DNS works in the background to help resolve them into IP Addresses. Like the Internet itself, the DNS wasn't built to be resilient to malicious users. This has led to many attacks on the DNS over the years. [1][2][3]

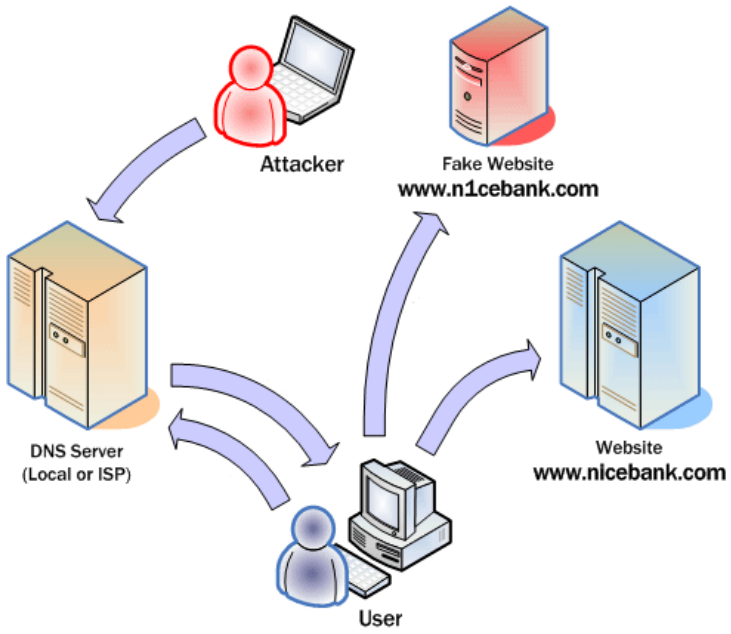
DNS Poisoning or Cache Poisoning or DNS Spoofing is one of the most lethal attack which has been mounted on the DNS. [6]

The way DNS works is that when a user queries for a specific domain name, say Microsoft.com the DNS systems comes into play. The system works on different levels as shown in the figure below:



The query proceeds to the next level provided that the local server does not have the response in its cache. There is another level at which the DNS system first queries. The system before moving out to the local server, first asks the Operating System, to see if it has the response in its cache. The DNS also checks for a response stored in the hosts file in the OS.

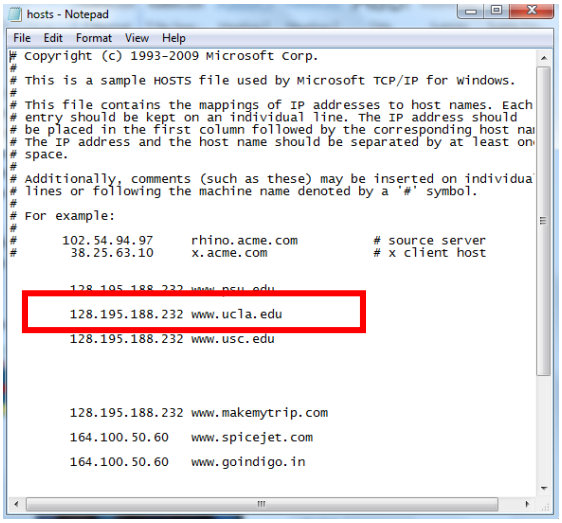
If at any of these levels, the server has been compromised, the user might be returned an address which isn't of the domain the user really wanted. In this scenario, the user will be directed to a different domain but the browser will remain oblivious to this fact as shown below:



DNS Poisoning is lethal for two major reasons:

- It has been proved that it is in fact very easy to mount an attack on the DNS server. [4]
- The attack is very different from the traditional phishing attacks. In the traditional phishing attack, the user is directed to a similar looking website. However, an alert user can quickly spot the difference if he looks at the address bar. A typical example of a phishing attack would be that the user is directed to facebooks.com instead of facebook.com. What makes DNS Poisoning scary is that the address bar will continue showing the query name which the user actually wants to see.

As an example we modified the hosts file and made the OS believe that the www.ucla.edu domain exists at 128.195.188.232 which is in reality the IP address of the domain: www.uci.edu.



In this case when the user seeks to visit the www.ucla.edu this is what happens:

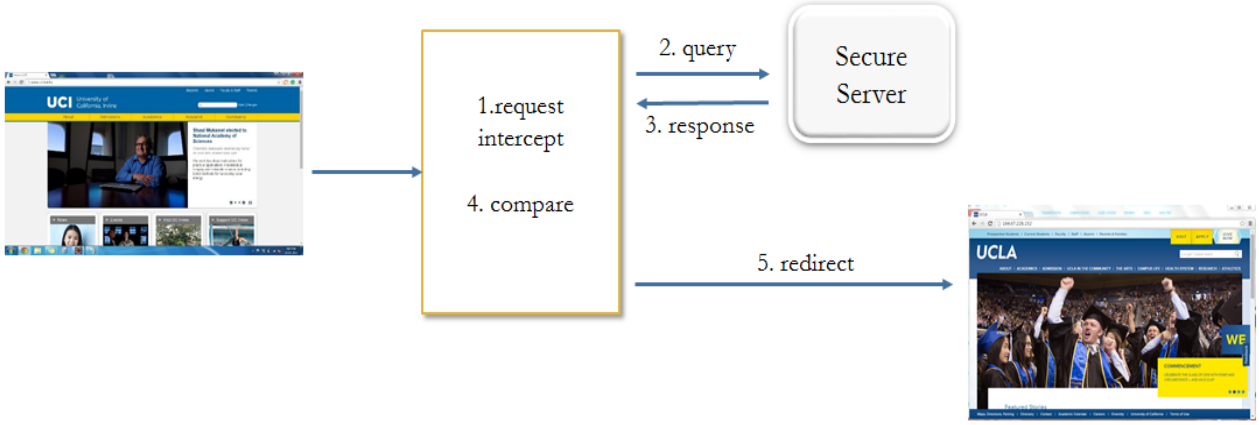
Note here that the address bar still shows that the user is on www.ucla.edu while in reality the user has been redirected to www.uci.edu which is hosted at 128.195.188.232.

While the final perfect solution to DNS Poisoning is DNSSec [8][9], its deployment in the near future seems a little difficult[7]. Certain other solutions have been proposed [5][10]. While most solutions try and work on recursive solutions to counter DNS Poisoning, we propose a new approach to deal with the problem.

III. IMPLEMENTATION AND RESULTS

We present a patch solution for a kind of security flaw known as DNS poisoning. There are various levels of hierarchy where the Domain Name can be resolved to an IP address. This attack can take place at any level and force the client to visit some unintended destination.

Our project has a client server architecture and includes a 5 step process as shown in below block diagram.



Step 1: The client types in a URL in the address bar and the local machine generates a list of associated IP's by doing a local lookup.

Step 2: The client sends this URL as a request message to a secure server which generates a list of associated IP's by doing another lookup.

Step 3: The server sends this list of IP's as a response message to the client.

We used different techniques for client-server communication namely TCP, SSL and Web Service. At present, our client code runs on a local machine and the server side code is hosted on an EC2 instance. The connection is established by including the cloud instance IP in the client code and then communicating using a RESTful web service.

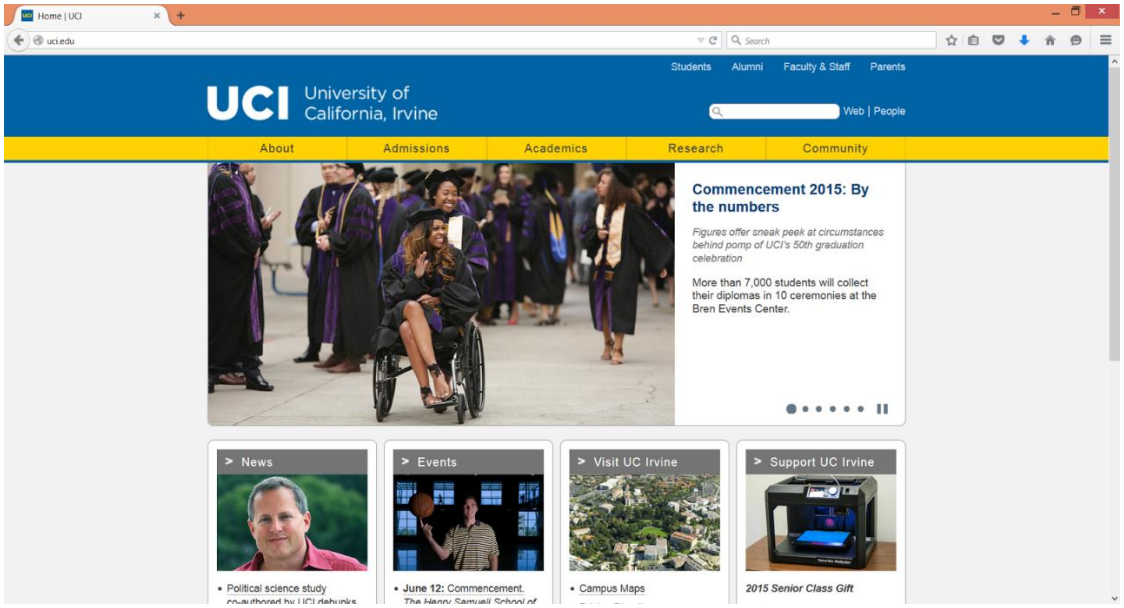
Step 4: The local lookup results are then compared with the response received from the server. Also, the server results are stored in a local cache along with the URL to avoid unnecessary overhead of communicating with the server.

Step 5: If the local lookup result is not a subset of the server result, then the DNS is poisoned. The user is displayed a message and is redirected to the first IP returned from the server. Otherwise, everything is fine and continues without any redirections.

When a user enter a new URL in the address bar, the local cache is first checked for a match. We have used HashMap for local storage to ensure efficient searching. If URL is present in cache, then the local lookup is compared with the results stored locally and the user is redirected if the DNS is poisoned. Otherwise, the client contacts the server for obtaining a list of IP's and this result set is stored in the cache after the comparison step.

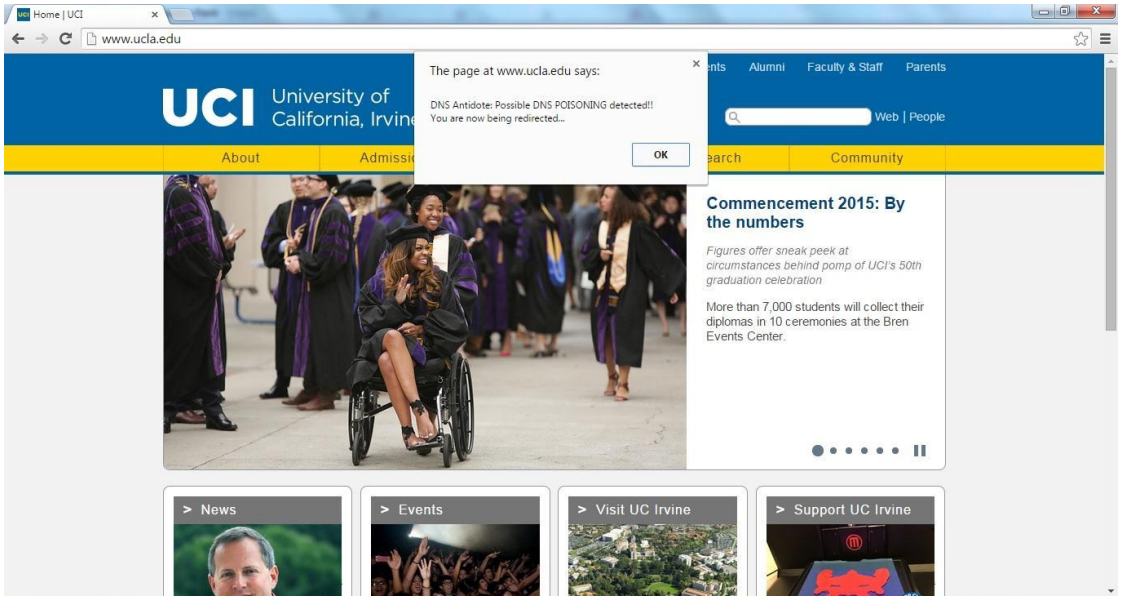
The below screenshots give a brief description about the functioning of our project.

Case 1: No DNS poisoning.

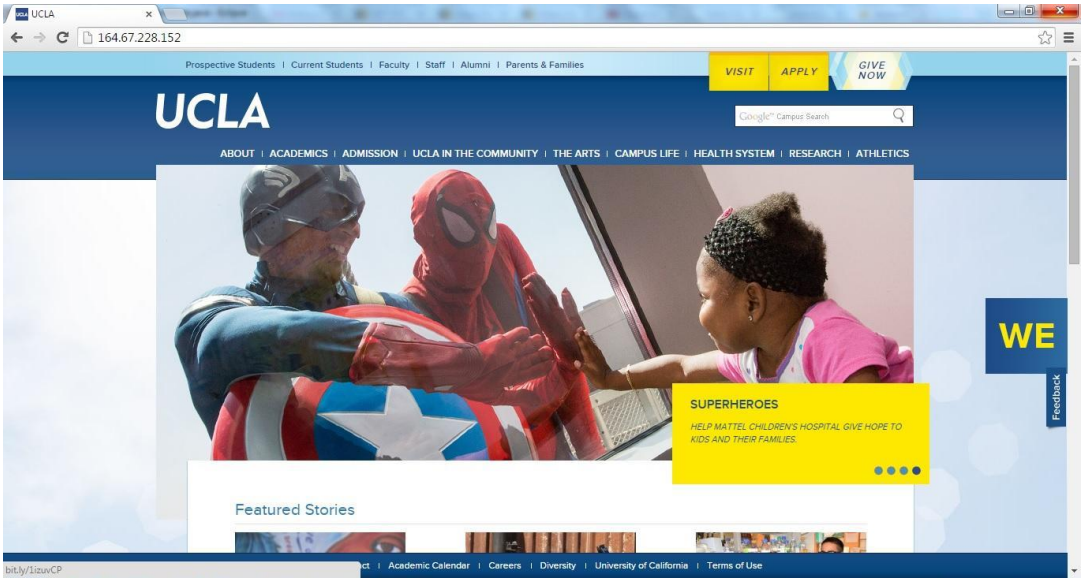


Case 2: DNS poisoning detected.

The user is displayed a message as shown below when poisoning is detected.



Finally the user is redirected to correct destination as shown below.

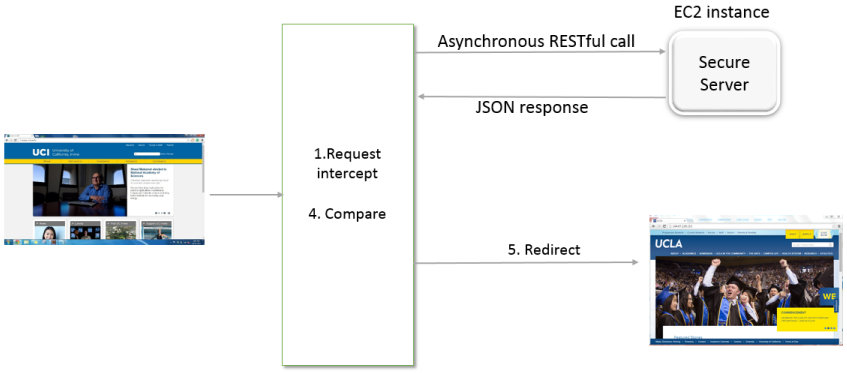


IV. RESTful Web Services

A web service is a collection of open protocols and standards used for exchanging data between applications or systems. [11] Software applications written in various programming languages and running on various platforms can use web services to exchange data over computer networks like the Internet in a manner similar to inter-process communication on a single computer. This interoperability (e.g., between Java and Python, or Windows and Linux applications) is due to the use of open standards. We thought of extending the project scope with the usage of a RESTful Web Service for communicating data with the secured server. The method of web service invocation helps us achieve [13]

- Service available over the Internet or private (intranet) networks
- Uses a standardized XML messaging system
- Is not tied to any one operating system or programming language
- Is self-describing via a common XML grammar

The system model after the web service incorporation can be shown as follows:



The address to access the web service hosted on the Amazon EC2 instance is hardcoded for the time-being to avoid any attach from the already poisoned DNS cache. The service availability can be guaranteed by using Elastic IP allocation to the computing instance from AWS.

JAX-RS: Java API for RESTful Web Services (JAX-RS) is a Java programming language API that provides support in creating web services according to the Representational State Transfer (REST) architectural pattern. JAX-RS uses annotations, introduced in Java SE 5, to simplify the development and deployment of web service clients and endpoints. The implementation of the JAX-RS used for the project is the widely used Jersey API for Java.

The following components are part of Jersey:

- Core Server: For building RESTful services based on annotation (jersey-core, jersey-server, jsr311-api)
- Core Client: Aids you in communicating with REST services (jersey-client)
- JAXB support
- JSON support
- Integration module for Spring and Guice.

The secured server instance on the AWS serves as a reliable DNS look-up for the input URL. The communication between the client and the server is possible though a JSON based object transport. The client parses the server response for further steps of matching.

Web Service hosting platform rationale.

The project heavily relies on the response of the secured server for the correctness of the application. It is therefore imperative for the server to be always online. Surveying the various available platforms like GCC, AWS, Heroku and more, and matching the application environment requirements, we finalized on the Amazon Elastic Beanstalk. The platform provides simple deployment with the Tomcat environment installed on the running EC2 instance. The description of the AWS Elastic Beanstalk from the website [14]

- AWS Elastic Beanstalk is an easy-to-use service for deploying and scaling web applications and services developed with Java, .NET, PHP, Node.js, Python, Ruby, Go, and Docker on familiar servers such as Apache, Nginx, Passenger, and IIS.
- You can simply upload your code and Elastic Beanstalk automatically handles the deployment, from capacity provisioning, load balancing, auto-scaling to application health monitoring. At the same time, you retain full control over the AWS resources powering your application and can access the underlying resources at any time.
- There is no additional charge for Elastic Beanstalk - you pay only for the AWS resources needed to store and run your applications.

IV. LIMITATIONS AND FUTURE SCOPE

1) The current implementation works for websites hosted with static IP's. The implementation for websites with dynamic IP allocation needs to care for the different IP allocation mechanisms and to keep the local application cache updated to reflect the same. Current implementation redirects to a valid IP resolution from the secured servers' scope, which might necessarily be not required. [12]

2) The application client requests response for each request made by the clients' browser. A degree of optimization achieved by caching the previous requests' secured server response helps in reducing the redirection latency and communication overhead for the client. However, the application then faces the problem of cache being stale and needs to employ controls for cache invalidation.

3) During the course of application testing, we adhered to few hosting companies like 'Cloudflare', which do not allow direct application redirection via an IP address in the browser. Websites which have similar hosting platforms have a limitation to the redirection mechanism adopted by our project implementation. A work-around for such cases is required.

References:

1. D. Atkins and R. Austein. Threat Analysis of the Domain Name System (DNS). RFC 3833 (Informational), August 2004.
2. C. Jackson, A. Barth, A. Bortz, W. Shao, and D. Boneh. Protecting browsers from DNS rebinding attacks. In CCS, 2007
3. M. Olnet, P. Mullen, and K. Miklavcic. Dan Kaminsky's 2008 DNS vulnerability, 2008. http://www.ietf.org/mail-archive/web/dnsop/current/ pdf2jgx6rzxN4.pdf
4. D. Kaminsky, DoxPara Research. Retrieved July 25, 2008, from DoxPara Research: http://www.doxpara.com/
5. R. Arends, R. Austein, M. Larson, D. Massey, and S. Rose. DNS security introduction and requirements. RFC 4033, Internet Engineering Task Force, Mar. 2005.
6. D. Kaminsky, "It's the End of the Cache As We Know It," in Black Hat conference, August 2008.
7. "Towards adoption of dnssec: Availability and security challenges," Cryptology ePrint Archive, Report 2013/254, 2013, http://eprint.iacr.org/.
8. R. Arends, R. Austein, M. Larson, D. Massey, and S. Rose. DNS Security Introduction and Requirement. RFC 4033, March 2005.
9. R. Arends, R. Austein, M. Larson, D. Massey, and S. Rose. Protocol Modifications for the DNS Security Extensions. RFC 4035, March 2005.
10. Network path problems in dnssec's deployment. In IETF 75 - dnsext, 2009. http://www.ietf.org/proceedings/75/slides/dnsext-3.pdf.
11. "Web Services- Tutorial" – Retrieved from https://docs.oracle.com/javaee/6/tutorial/doc/bnayk.html
12. "Static vs. Dynamic IP addressing" – Retrieved from https://support.google.com/fiber/answer/3547208?hl=en
13. RESTful Web Services Developer's Guide – Oracle, Part No: 820-4867-11 Sun Microsystems, Inc. 4150 Network Circle Santa Clara, CA 95054 U.S.A.
14. AWS Elastic Beanstalk - Retrieved from http://aws.amazon.com/elasticbeanstalk/