2-21-2014

Dictionary Attacks and Password Selection

Tarun Madiraju

R·I·T

Rochester Institute of Technology

B. Thomas Golisano College of

Computing and Information Sciences

Thesis Report
Dictionary Attacks and Password Selection

By

Tarun Madiraju

A Thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science
Computing Security & Information Assurance

Department of Computing Security

Date: 02/21/2014

Committee Approval

10 - MARCH - 14

Bill Stackpole Date

Thesis Committee Chairperson

3/10/14

Daryl Johnson Date

Thesis Committee Member

3/6/2014

Yin Pan Date

Thesis Committee Member

ABSTRACT

Passwords, particularly text-based, are the most common authentication mechanisms across all platforms and services like computers, mobiles, web and network services. Existing password strength evaluators and online service providers (Gmail, Yahoo, Paypal, Twitter, etc) password strength estimators determine the effectiveness of passwords chosen by user based on entropy techniques or a similar function of the parameters: length, complexity and predictability. Such implementations often ignore passwords part of publicly available password dictionaries and password leaks which are often the primary choice for malicious adversaries and particularly script kiddies. This paper presents an application that would help in preventing the use of such passwords thereby reducing the impact of dictionary based password attacks significantly. The application maintains a database of unique passwords by gathering publicly available password dictionaries and passwords leaked over the Internet. The application provides users with an interface to query the database and verify if their passwords are already available on the Internet thereby preventing them from the use of such passwords.

List of Tables

List of Figures

Table of Contents

1 INTRODUCTION

Passwords are the primary choice for authenticating users and are likely to remain for a significant amount of time in the future [1] [2] because of the practicality and convenience aspects associated with them for service providers and end- users respectively. Qualys [3] identified password guessing attacks as a top cyber security risk after analyzing attack data from TippingPoint intrusion prevention systems deployed for protecting 6000 organizations.

A pre-known list of passwords is gathered together to form a password dictionary file. Dictionary attacks leverage such password dictionaries and automate the attempt of breaking in to password-protected applications and systems by trying each word/password listed in the password dictionary. Dictionary based password cracking is easy to execute and almost every password cracking tool is bundled with a pre-built dictionary thereby aiding script-kiddies in performing dictionary based password attacks. Attackers build comprehensive password dictionaries by compiling passwords from various dictionaries and password leaks.

Alexander Lystad analyzed password leaks from password attacks on Rockyou and phpBB [4], and observed that the percentage of unique passwords that can be cracked using common well-defined dictionaries (like John-The-Ripper, Cain & Abel, etc) is 33.41% and 39.43% respectively. The analysis indicates the necessity of enabling users to be aware of such passwords and thereby prevent them from using those passwords. With passwords being the most common form of authentication and a critical aspect of identifying users while providing necessary privacy, password based attacks and defense systems are subjects of ongoing research.

The primary goal of this project is to build a database of unique passwords from publicly available password dictionaries and passwords leaked over the internet, and implement an application that would allow regular users and administrators to verify if their passwords are susceptible to dictionary attacks.

The remainder of this paper is organized as follows. Section 2 discusses the literature review which details existing work in authentication mechanisms particularly passwords and dictionary attacks (their detection, prevention and response systems). Section 3 details the design and implementation aspects of the application developed. Section 4 presents statistics related to password dictionaries, leaks and attacks. Section 5 and 6 discuss future work and conclusion respectively.

2 LITERATURE REVIEW

Linux pluggable authentication module (PAM) pam_cracklib [5] incorporates a similar approach to the proposed as it is used to test passwords against dictionary words. The Linux pam_cracklib module allows system administrators to incorporate strength checking for user passwords against a system dictionary followed by a set of password policies [5]. The drawbacks of pam_cracklib module are poor documentation, limited implementation (few *nix distributions) and that it is primarily aimed for use by system administrators [6].

Saikat Chakrabarti and Mukesh Singhal present an analysis of existing dictionary attack prevention techniques and their drawbacks. They discuss about encrypted key exchange based protocols for protection against offline dictionary attacks. For thwarting online dictionary attacks, they discuss account lockout, delayed response, extra computations and Reverse Turing Tests (RTT) [7]. The Password-based Encrypted Key Exchange (EKE) by Steven Bellovin and Michael Merritt incorporates a combination of cryptographic schemes to prevent offline dictionary attacks but it was later observed that the EKE and variants of EKE protocols are vulnerable to plain text equivalence which would allow an adversary to masquerade as a victim by using his hashed password captured through eavesdropping [8] [9]. Delayed responses and account locking are common countermeasures for online dictionary attacks as they reduce the number of passwords that can be guessed in a given time and lock the user account after reaching a threshold set for failed login attempts. However, as outlined by Pinkas et. al these countermeasures can result in denial of service and increased customer service costs as a result of account locking. Further, an attacker can try multiple login attempts in parallel with different user accounts to circumvent delayed response and account lockout countermeasures [1]. The extra computation based technique involves the inclusion of non- trivial computation in addition to providing the password. The idea of such a technique is to incorporate a large overhead for password attack tools as it would require computation for every login attempt thereby reducing the number of attempts significantly. The extra computation technique might present usability issues for a legitimate user while an adversary can tackle the overhead by using a powerful attack machine or environment [7]. Pinkas et al. use a similar approach by incorporating RTT technique to prevent automated programs from carrying out dictionary attacks. In this approach, the user needs to present his password and pass the RTT to ensure a successful login [1]. However, it was observed by Stuart Stubblebine and Paul van Oorschot that the RTT-based implementation is prone to RTT relay attacks [7]. Several real world RTTs (also called as CAPTCHAs which refer to Completely Automated Public Turing test to tell Computers and Humans Apart) implemented by popular online service providers have been broken in the past using computer character recognition based projects [10], [11] [12] [13].

Techniques for improving authentication or protection against password attacks in general range from hardware based solutions, biometric authentication, client certificate mechanisms, graphical password schemes, grid based logins, multi- factor authentication etc [1]. Pinkas et. al present a range of such existing authentication mechanisms and drawbacks associated with them [1], which are discussed as follows. The hardware and biometric based authentication solutions form a robust authentication technique but also include a range of drawbacks like additional costs and overhead associated with the need for additional devices and migration from traditional password based authentication. Additionally, hardware based authentication solutions also involve usability issues as a result of losing or forgetting device. Multi-factor authentication schemes often combine passwords (something you know) with hardware (something you have) or biometric solutions (something you are) and hence demonstrate the same drawbacks as of hardware and biometric solutions along with other convenience issues. Client certificates is another solution that implements a software based authentication approach but include drawbacks associated with key portability and storage.

There are a vast number of password strength evaluating applications available over the Internet that assist users in determining the strength of the passwords chosen by them [14]. Examples for such password evaluating applications are Password Strength Checker at passwordmeter.com [24], GeodSoft Password Evaluator [15], GetSecurePassword [16], How Secure is My Password? [17], etc. Further, almost every online service provider implements a password strength indicator to allow users to gauge the strength of their password while setting up a new account or changing their existing password. Serge Egelman et. al observed that password strength meters result in stronger password selection by users for important accounts [18] but these online password strength determining tools and password strength meters do not take publicly released passwords into consideration while determining the strength of user chosen password.

The Online Domain Tools team analyzed common password strength evaluating tools available over the Internet and noted that entropy and complexity are the primary strength determination parameters while only a few tools considered dictionary attacks [14]. Online domain tools incorporate dictionary attack based password strength determination application [19] on their website where they break down passwords, observe the presence of regular dictionary words or parts of it, the presence of leetspeak and other pattern analysis techniques but the application doesn't consider publicly available passwords. Online services like pwnedlist.com [20] and shouldichangemypasswordnow.com [21] obtain the user's email address and verify if their accounts have ever been compromised based on emails and passwords gathered from password leaks over the internet. These services inform the user about the presence of their passwords in public on the basis of their email address. The passwords from dictionaries and leaks that are not associated with any emails are ignored by these services and hence do not prevent users from using passwords already available on the Internet.

Another online project leakdb by Abusix states gathering leaked and publicly available human generated hashes as its objective [22]. This project is similar to the concept demonstrated in this paper but it is different from the application presented in this paper for the following reasons. The leakdb project maintains a database of hashes available over the Internet. It takes a hashed value as input and reports the corresponding clear text if it is present in the leakdb database. The project only stores cracked hashes in the database thereby missing leaked hashes (available on the internet) that haven't been cracked yet. Also, it doesn't consider passwords from common password dictionaries and passwords leaked in plain text over the Internet. Further, querying uncracked hashes results in submission of such hashes to leakdb password cracking feature without notifying the user about such an implementation prior to querying. This presents a risk to the users as their password is added to the leakdb search database after being cracked.

The various limitations in terms of costs, usability, awareness, etc. associated with alternate authentication schemes often result in users sticking with the traditional password based authentication schemes. These alternate authentication schemes and password attack defense systems require selection of a regular password and an additional item (something you have/something you are). Dictionary attacks can be tackled at the root level by aiding the users in selecting passwords that are relatively less prone to dictionary attacks. In this paper, we demonstrate a technique that aims at equipping users with a dictionary attack based password evaluation system thereby encouraging them to avoid publicly available passwords and preventing them from being simple targets.

3 APPLICATION DETAILS

3.1 Application

The application built comprises of a user interface, a database and helper components. The database for this application primarily contains schemas for each hash value supported by the application. These schemas maintain the database of passwords gathered from various password leaks and dictionaries over time. The database also has a submission schema that is responsible for maintaining database of user input ranging from feedback to information about password leaks and dictionaries. The user interface is a web application that provides a search bar in order to allow users to input passwords and query the presence of their password in the password database. The helper components of the application are responsible for working behind the scenes to facilitate gathering of password dictionary or leak sources and import passwords into the password database. The web application was built on PHP backed by a MySQL database. The helper components (importer, harvester and statster) were implemented in python. The helper components are discussed next in this section.

3.2 Helper Components

1) **Importer:** The importer component of the application is responsible for obtaining passwords from a file (password dictionary or password leak) and importing each password into the database. The user passes the following parameters to the importer utility: FILENAME, FORMAT, FILETYPE, DATE and URL. Passwords in the password dictionary or leak are typically in the same format - plaintext or a hash type. This input type (plaintext or hash) of passwords is provided by the FORMAT parameter. The FILETYPE parameter refers to the type of the password file and it is either of leak type or dictionary type. The DATE parameter, as the name implies, refers to the date on which the security breach happened or the date the passwords were released to public. The URL parameter can either be a media post about the breach or the location where the password leak or dictionary are hosted.

2) **Harvester:** The harvester component of the application is responsible for searching and harvesting information about password dictionaries, security breaches and password leaks. The harvester component is a collection of scripts that generate an output file with URLs of potential password dictionaries, password leaks or information associated with them.

3) **Statster:** Statster is a collection of scripts used for generating statistics outlined in Table 1. These scripts calculate the total number of entries contributed by a leak or a dictionary to the database and the crackability aspects. The crackability aspect for online service providers (like Google) or password strength estimating websites (like passwordmeter.com) refer to the percentage of passwords (in a leak file) that were identified as weak by those online service providers or password strength estimating websites while for a password database (built using leaks, dictionaries, etc) like the password database built in this paper (referred to as CommonDB) and crackstation's password dictionary [25] (referred to as CrackstationDB). Additional information about the statistics is provided in the statistics section of this document. The scripts for calculating the crackability aspect for Google's password rating mechanism [23], passwordmeter.com [24] and CommonDB were implemented in python while the utilities for calculating crackability aspect for CrackstationDB were obtained from the crackstation-hashdb github repository [26] and modified to our requirements. Additional information about the statistics is provided in the statistics section of this document.

3.3 Working

The application working comprises of two processes: building the password database and the web application for serving user queries.

3.3.1 Building the password database

The first step in the process of building the database involves gathering password dictionaries and leak sources by the application maintainer. These sources are then inserted into the submission schema of the database that also stores user feedback and details of password leak or dictionary sources submitted by application users. These sources (typically URLs) are then verified manually in order to avoid potential database poisoning by malicious users.

The verification process involves identifying at least 3 media posts relevant to each URL. After verification, password files are downloaded from URLs obtained using harvester utility, user submission and manual research. The importer utility is leveraged to upload passwords into the database. A database schema is created for each new hash format. This way the database would have a collection of schemas like md5, sha1, etc. Passwords are uploaded to the specific schemas based on their hash type. If a leak or a dictionary file has passwords in plaintext, they are converted to md5 hash and stored in md5 schema. This is to ensure that the application works with hashed version the password only to prevent any potential misuse.

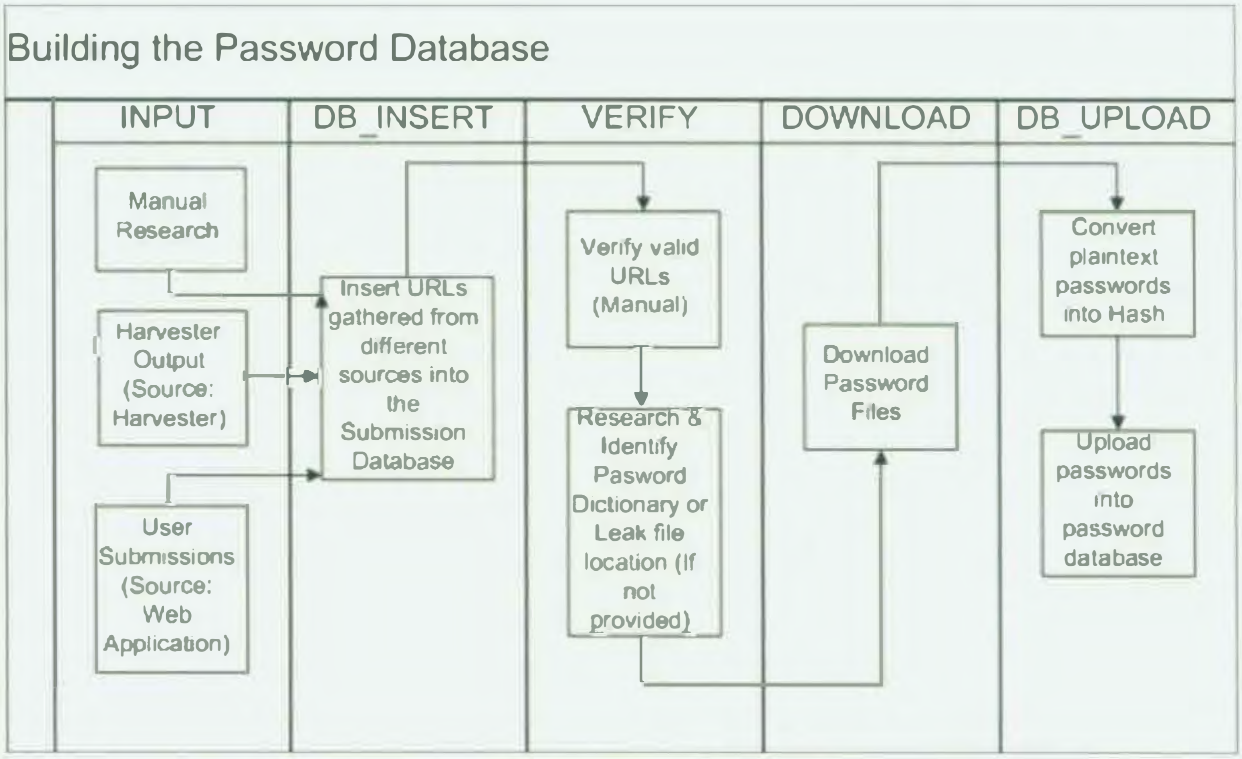

Building the Password Database
INPUT
DB_INSERT
VERIFY
DOWNLOAD
DB_UPLOAD
Manual Research
Harvester Output (Source: Harvester)
User Submissions (Source: Web Application)
Insert URLs gathered from different sources into the Submission Database
Verify valid URLs (Manual)
Research & Identify Pasword Dictionary or Leak file location (If not provided)
Download Password Files
Convert plaintext passwords into Hash
Upload passwords into password database

Figure 1: Building the Password Database

The application database has been designed considering the performance and scalability aspect. If a password file has been identified with a hash format that does not have a corresponding schema yet, the importer utility will create a schema for that hash format and subsequently all the leaks or dictionaries that have passwords in this hash format will be uploaded to this schema as separate tables as opposed to maintaining a single table for storing all the hashes. This way, for a specific hash input the respective hash schema will be searched rather than searching the entire database. The same has been depicted using the following two figures. Figure 2: Current database represents the current database while Figure 3: Future Database refers to the database structure when different hash inputs are imported into the database.

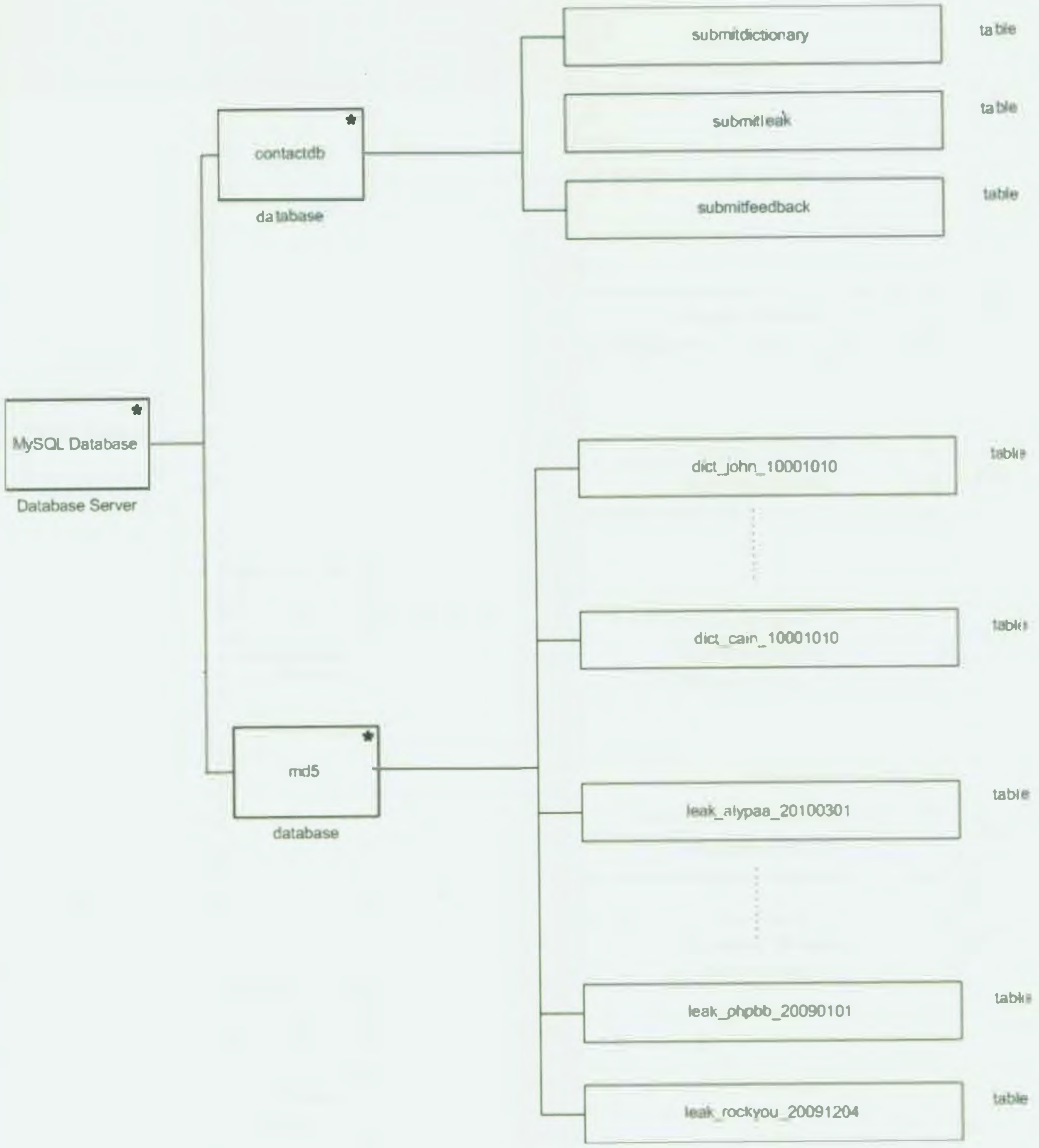


Figure 2: Current Database Structure

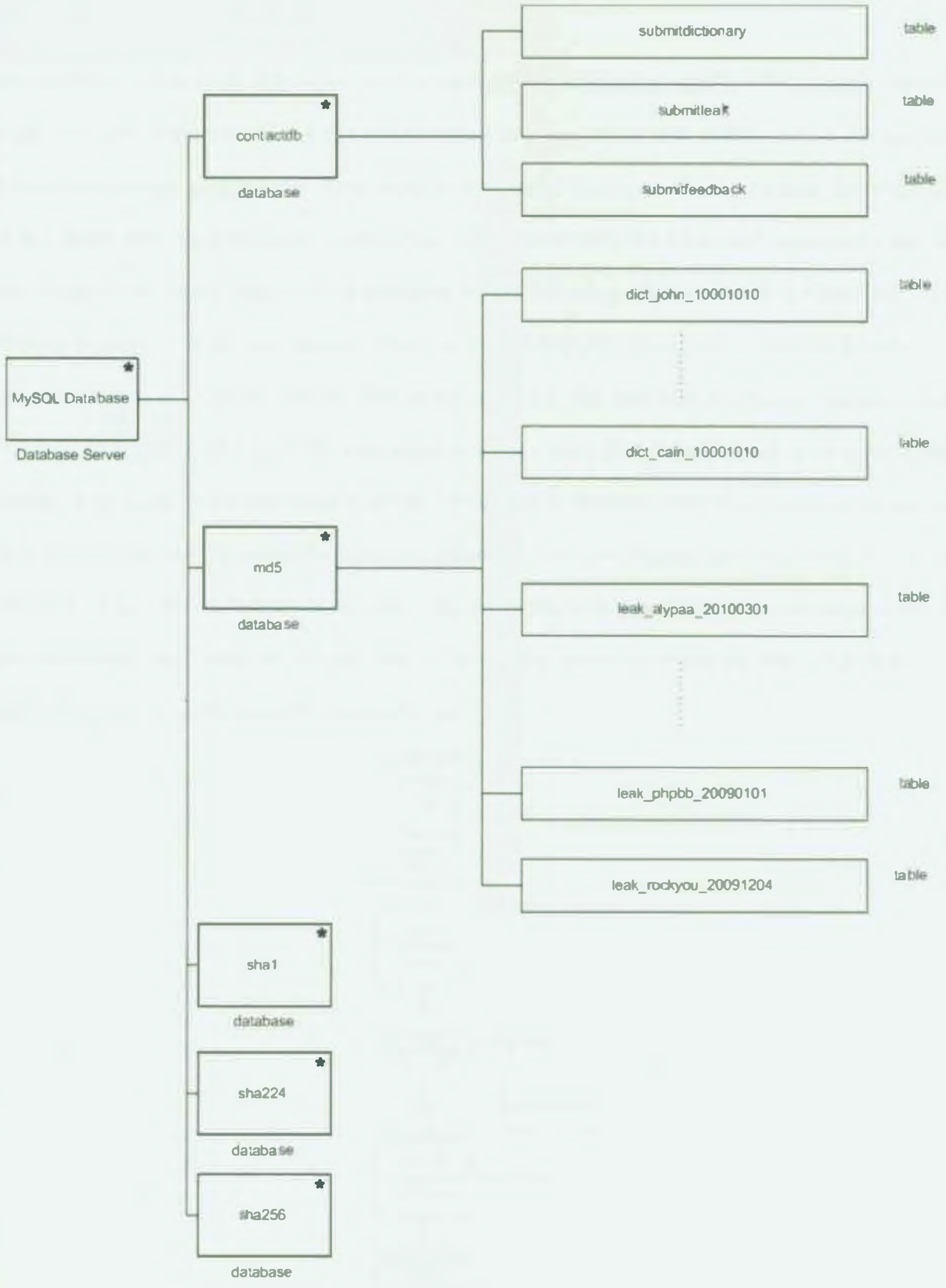


Figure 3: Future Database Structure

3.3.2 Application Workflow

The workflow of this application is similar to a standard web application workflow for a search operation. Using our application, the user enters the password in a search bar and selects one of the two radio buttons that are used to determine if user input is in plaintext format or in hashed format. On submission of this search form, the PHP search script is fired on the server side that takes user's input and converts it into a hash if the input supplied is in plaintext format following which a search for that hash in the database is initiated. If the user input is already in hash format, the input hash is searched directly in the database without any hashing process. The application looks for user hash in specific database schema based on hash format (for eg. if the user input is a sha1 hash, then the password is searched in sha1 schema only) while on the other side if the user input was in plaintext, user input is converted into each hash format supported by our application and searched in all hash schemas until the hash is found. The final step of this process is alerting the user if the password hash was found in our database or not and also informing them about the source (leak or dictionary name) in which the hash was found. The application doesn't store passwords inputted by user.

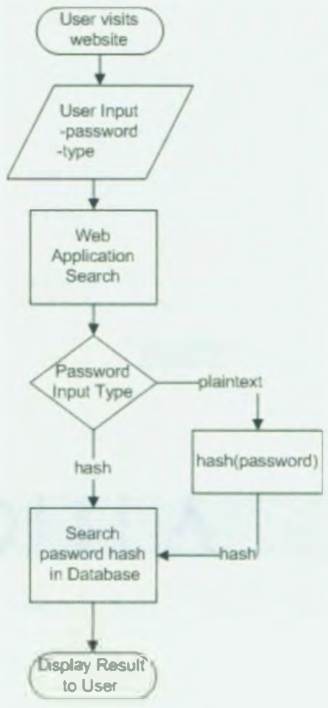
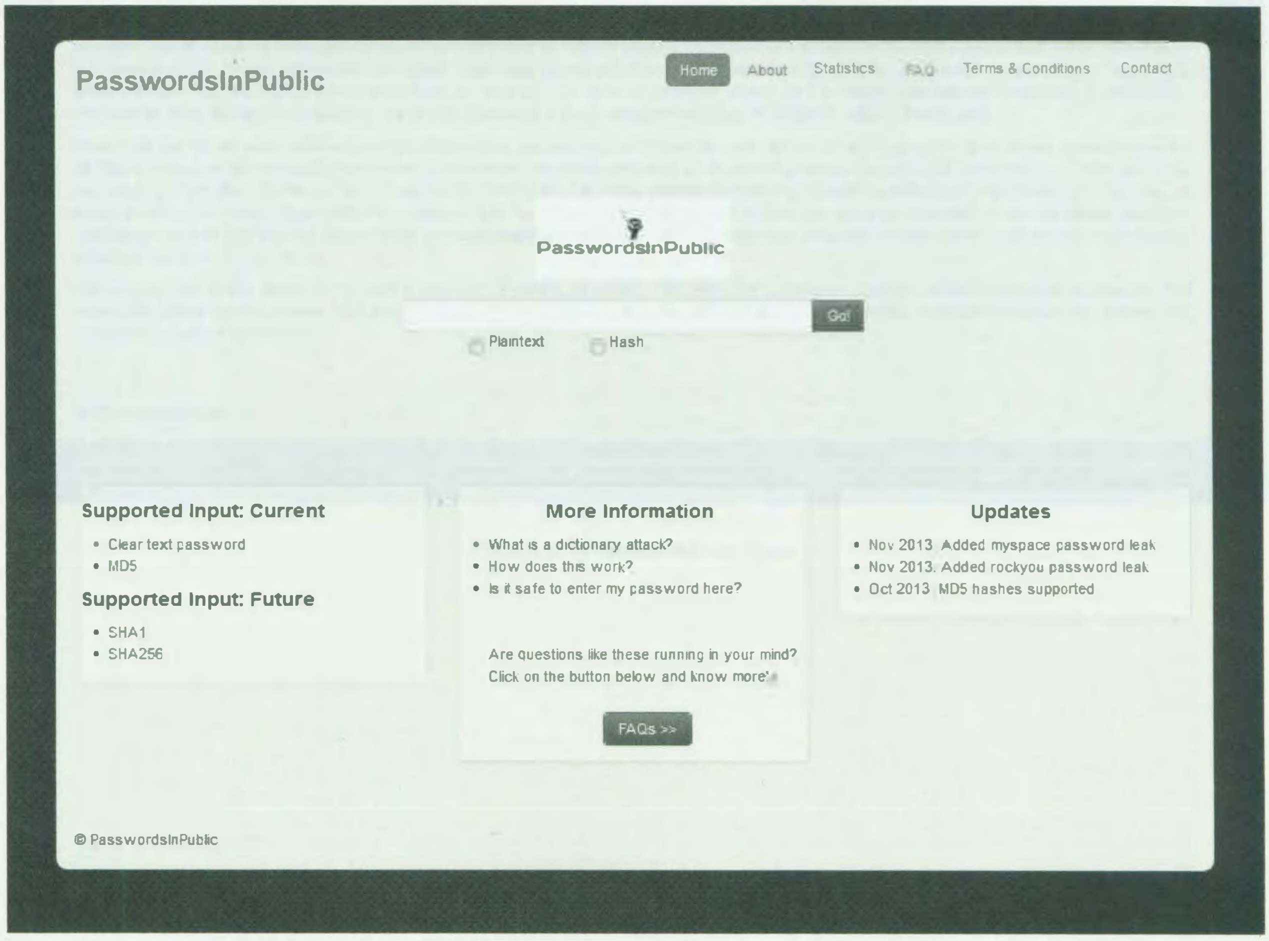


Figure 4: Web Application Workflow

3.4 Application GUI & Usage

Website: Home Page

The home page of the web application presents a search bar for receiving user's password. Since, passwords are in plain text or hash format we present the user with two radio buttons to present the application with the type of input. In addition to the user input, the home page also focuses on providing users with some important information about the application with the intention of encouraging them to use this application.


PasswordsInPublic

Home About Statistics FAQ Terms & Conditions Contact

PasswordsInPublic

Go!

Plaintext Hash

Supported Input: Current

- Clear text password
- MD5

Supported Input: Future

- SHA1
- SHA256

More Information

- What is a dictionary attack?
- How does this work?
- Is it safe to enter my password here?

Are questions like these running in your mind? Click on the button below and know more!

FAQs >>

Updates

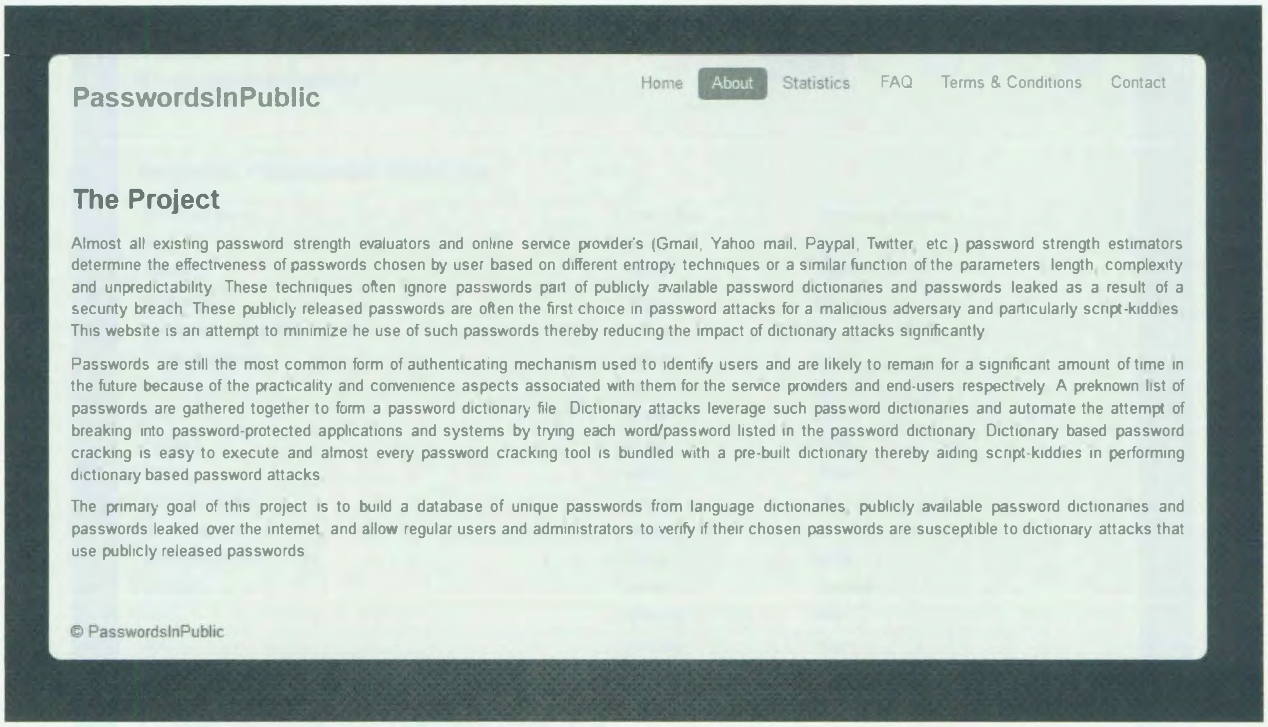
- Nov 2013: Added myspace password leak
- Nov 2013: Added rockyou password leak
- Oct 2013: MD5 hashes supported

© PasswordsInPublic

Figure 5: Website - Home Page

Website: About Page

The about page of the website is a simple page provides general background information about the most common authenticating mechanism - passwords and the state of existing password strength estimation utilities. The page also describes the goals of this project.



The Project

Almost all existing password strength evaluators and online service provider's (Gmail, Yahoo mail, Paypal, Twitter, etc.) password strength estimators determine the effectiveness of passwords chosen by user based on different entropy techniques or a similar function of the parameters: length, complexity and unpredictability. These techniques often ignore passwords part of publicly available password dictionaries and passwords leaked as a result of a security breach. These publicly released passwords are often the first choice in password attacks for a malicious adversary and particularly script-kiddies. This website is an attempt to minimize he use of such passwords thereby reducing the impact of dictionary attacks significantly.

Passwords are still the most common form of authenticating mechanism used to identify users and are likely to remain for a significant amount of time in the future because of the practicality and convenience aspects associated with them for the service providers and end-users respectively. A preknown list of passwords are gathered together to form a password dictionary file. Dictionary attacks leverage such password dictionaries and automate the attempt of breaking into password-protected applications and systems by trying each word/password listed in the password dictionary. Dictionary based password cracking is easy to execute and almost every password cracking tool is bundled with a pre-built dictionary thereby aiding script-kiddies in performing dictionary based password attacks.

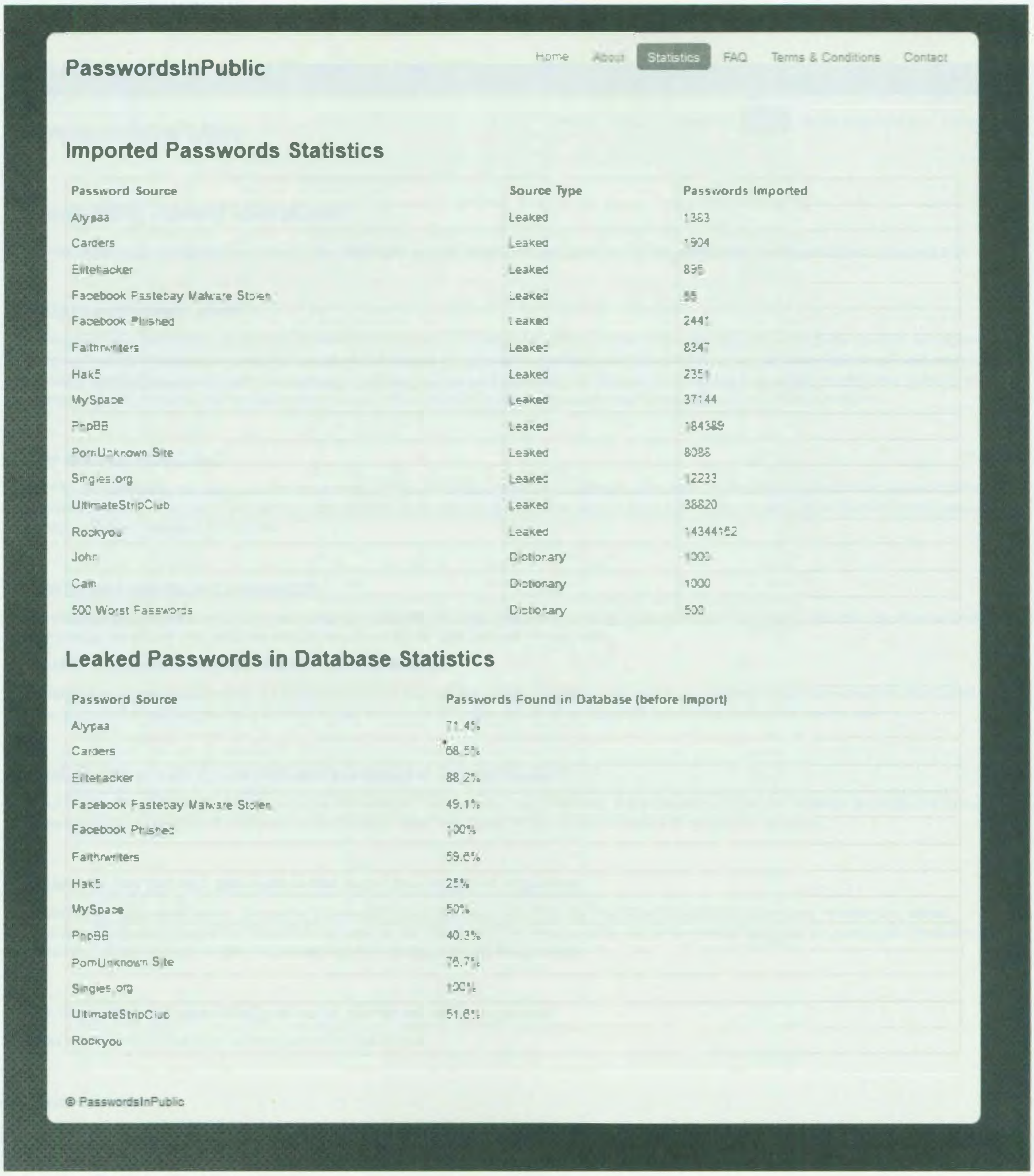
The primary goal of this project is to build a database of unique passwords from language dictionaries, publicly available password dictionaries and passwords leaked over the internet, and allow regular users and administrators to verify if their chosen passwords are susceptible to dictionary attacks that use publicly released passwords.

© PasswordsInPublic

Figure 6: Website - About Page

Website: Statistics Page

The statistics page of the website provides various statistics associated with the observations of this project. The statistics page presents the imported passwords statistics and the crackability statistics. The imported passwords statistics list the source type (password leak or password dictionary) of an import and the number of passwords imported using each file.



Imported Passwords Statistics

Password Source	Source Type	Passwords Imported
Alypaa	Leaked	1383
Carders	Leaked	1904
Elitehacker	Leaked	895
Facebook Pastebay Malware Stolen	Leaked	55
Facebook Phished	Leaked	2441
Faithrwriters	Leaked	8347
Hak5	Leaked	2351
MySpace	Leaked	37144
PhpBB	Leaked	184389
PornUnknown Site	Leaked	8085
Singles.org	Leaked	12233
UltimateStripClub	Leaked	38820
Rockyou	Leaked	14344182
John	Dictionary	1000
Cain	Dictionary	1000
500 Worst Passwords	Dictionary	500

Leaked Passwords in Database Statistics

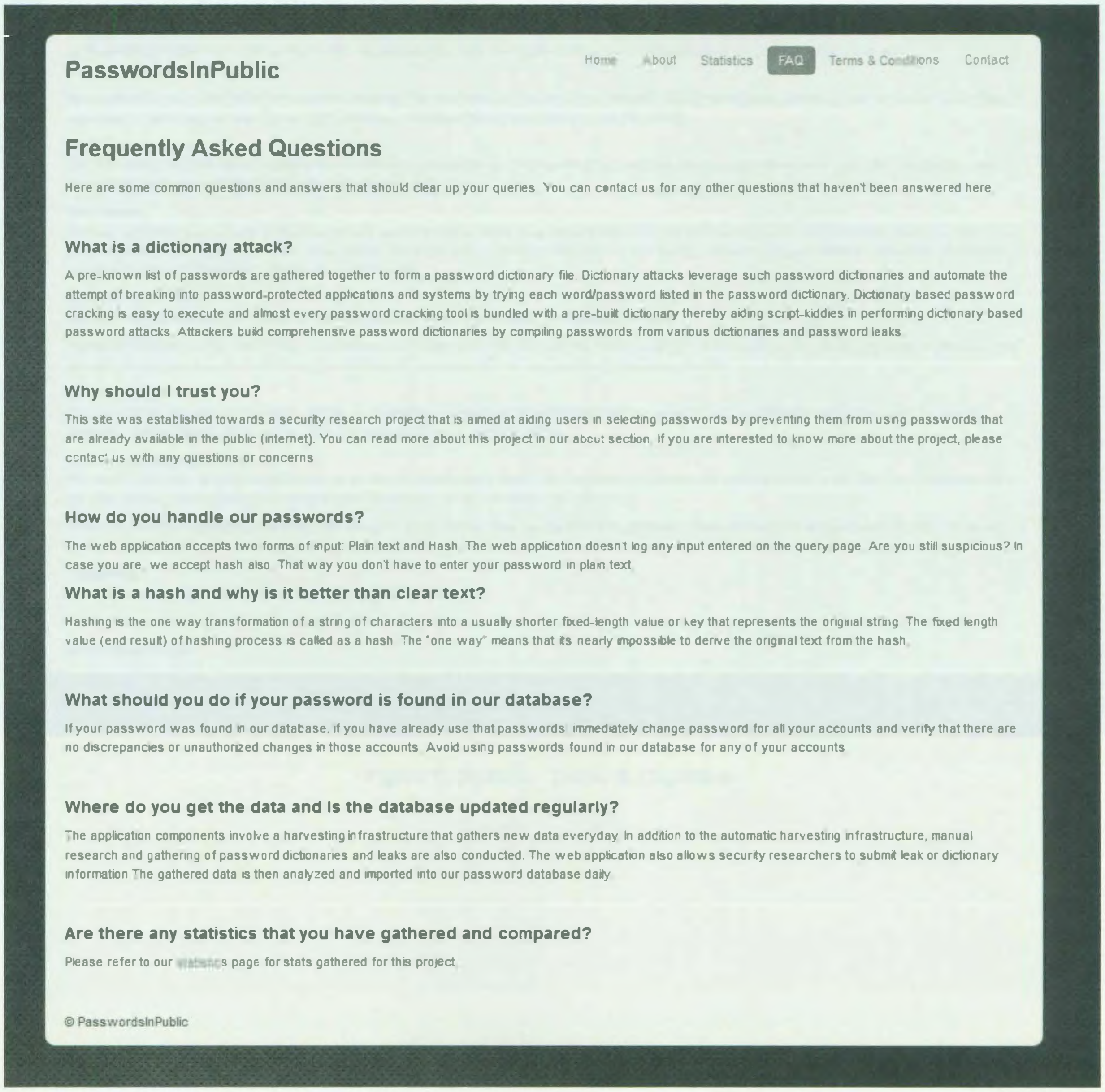
Password Source	Passwords Found in Database (before Import)
Alypaa	71.4%
Carders	68.5%
Elitehacker	88.2%
Facebook Pastebay Malware Stolen	49.1%
Facebook Phished	100%
Faithrwriters	59.6%
Hak5	25%
MySpace	50%
PhpBB	40.3%
PornUnknown Site	78.7%
Singles.org	100%
UltimateStripClub	51.6%
Rockyou	

© PasswordsInPublic

Figure 7: Website - Statistics Page

Website: FAQ Page

The FAQ button on the first page and the FAQ tab present the below shown Frequently Asked Questions. This page outlines basic information related to dictionary attack and additional information about the way the application is built which includes handling user's password, password database updates and statistics. The idea of the FAQ page is to provide transparency about the web application functioning and encourage user's to try this application.



Frequently Asked Questions

Here are some common questions and answers that should clear up your queries You can contact us for any other questions that haven't been answered here

What is a dictionary attack?

A pre-known list of passwords are gathered together to form a password dictionary file. Dictionary attacks leverage such password dictionaries and automate the attempt of breaking into password-protected applications and systems by trying each word/password listed in the password dictionary. Dictionary based password cracking is easy to execute and almost every password cracking tool is bundled with a pre-built dictionary thereby aiding script-kiddies in performing dictionary based password attacks Attackers build comprehensive password dictionaries by compiling passwords from various dictionaries and password leaks

Why should I trust you?

This site was established towards a security research project that is aimed at aiding users in selecting passwords by preventing them from using passwords that are already available in the public (internet). You can read more about this project in our about section If you are interested to know more about the project, please contact us with any questions or concerns

How do you handle our passwords?

The web application accepts two forms of input: Plain text and Hash The web application doesn't log any input entered on the query page Are you still suspicious? In case you are we accept hash also That way you don't have to enter your password in plain text

What is a hash and why is it better than clear text?

Hashing is the one way transformation of a string of characters into a usually shorter fixed-length value or key that represents the original string The fixed length value (end result) of hashing process is called as a hash The "one way" means that its nearly impossible to derive the original text from the hash

What should you do if your password is found in our database?

If your password was found in our database, if you have already use that passwords immediately change password for all your accounts and verify that there are no discrepancies or unauthorized changes in those accounts Avoid using passwords found in our database for any of your accounts

Where do you get the data and Is the database updated regularly?

The application components involve a harvesting infrastructure that gathers new data everyday In addition to the automatic harvesting infrastructure, manual research and gathering of password dictionaries and leaks are also conducted. The web application also allows security researchers to submit leak or dictionary information The gathered data is then analyzed and imported into our password database daily

Are there any statistics that you have gathered and compared?

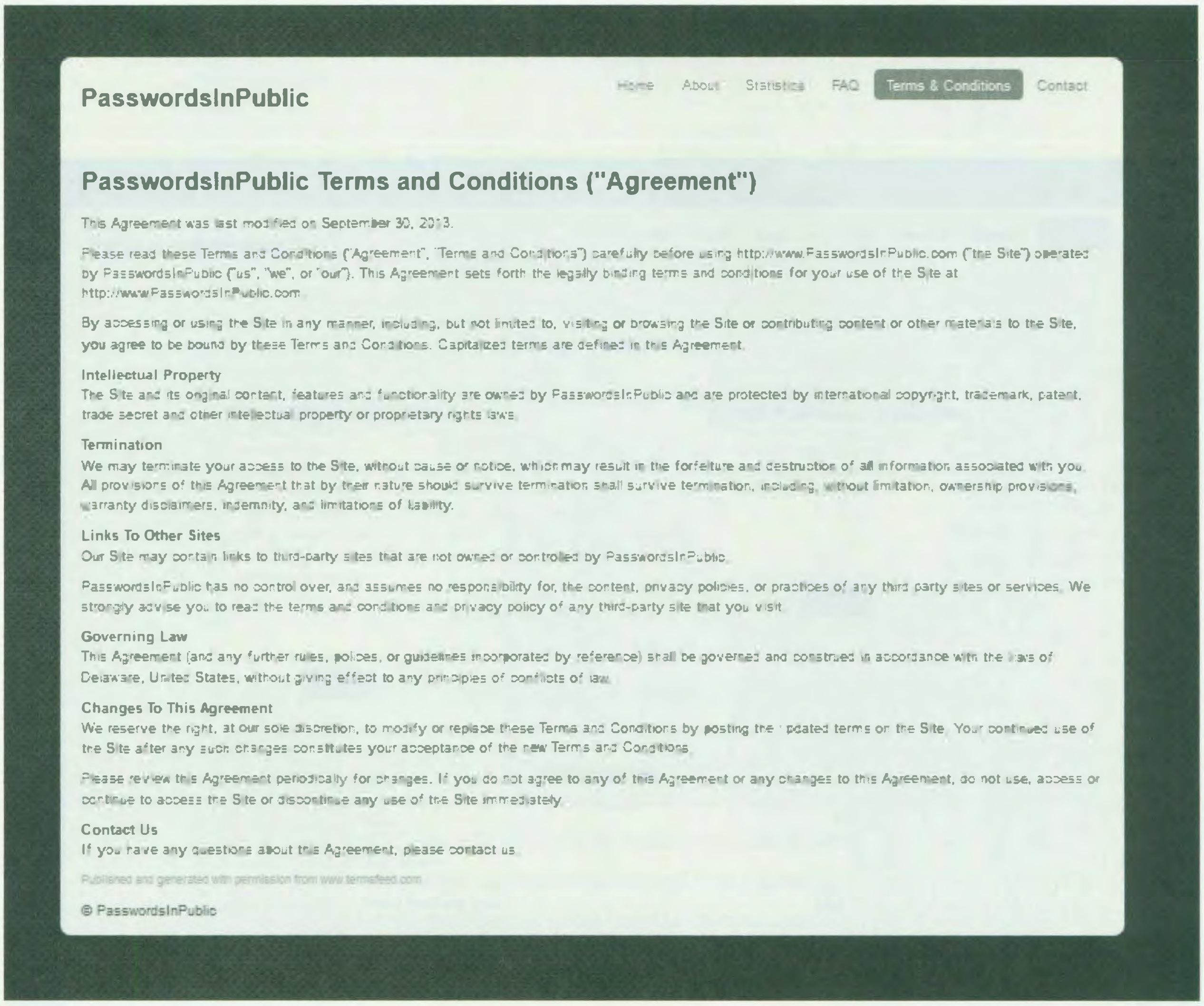
Please refer to our statistics page for stats gathered for this project

© PasswordsInPublic

Figure 8: Website - FAQ Page

Website: Terms & Conditions (Generic)

The Terms & Conditions page lays down standard agreement for a online web application.



PasswordsInPublic Terms and Conditions ("Agreement")

This Agreement was last modified on September 30, 2013.

Please read these Terms and Conditions ("Agreement", "Terms and Conditions") carefully before using http://www.PasswordsInPublic.com ("the Site") operated by PasswordsInPublic ("us", "we", or "our"). This Agreement sets forth the legally binding terms and conditions for your use of the Site at http://www.PasswordsInPublic.com.

By accessing or using the Site in any manner, including, but not limited to, visiting or browsing the Site or contributing content or other materials to the Site, you agree to be bound by these Terms and Conditions. Capitalized terms are defined in this Agreement.

Intellectual Property
The Site and its original content, features and functionality are owned by PasswordsInPublic and are protected by international copyright, trademark, patent, trade secret and other intellectual property or proprietary rights laws.

Termination
We may terminate your access to the Site, without cause or notice, which may result in the forfeiture and destruction of all information associated with you. All provisions of this Agreement that by their nature should survive termination shall survive termination, including, without limitation, ownership provisions, warranty disclaimers, indemnity, and limitations of liability.

Links To Other Sites
Our Site may contain links to third-party sites that are not owned or controlled by PasswordsInPublic.

PasswordsInPublic has no control over, and assumes no responsibility for, the content, privacy policies, or practices of any third party sites or services. We strongly advise you to read the terms and conditions and privacy policy of any third-party site that you visit.

Governing Law
This Agreement (and any further rules, polices, or guidelines incorporated by reference) shall be governed and construed in accordance with the laws of Delaware, United States, without giving effect to any principles of conflicts of law.

Changes To This Agreement
We reserve the right, at our sole discretion, to modify or replace these Terms and Conditions by posting the updated terms on the Site. Your continued use of the Site after any such changes constitutes your acceptance of the new Terms and Conditions.

Please review this Agreement periodically for changes. If you do not agree to any of this Agreement or any changes to this Agreement, do not use, access or continue to access the Site or discontinue any use of the Site immediately.

Contact Us
If you have any questions about this Agreement, please contact us.

Published and generated with permission from www.termsfeed.com

© PasswordsInPublic

Figure 9: Website - Terms & Conditions

Website: Contact Page

The contact page is an import aspect of this application. The contact page is used for two important things: receiving password submissions and receiving feedback. The password database is a collection of password leaks harvested over the internet and the commonly used dictionaries.

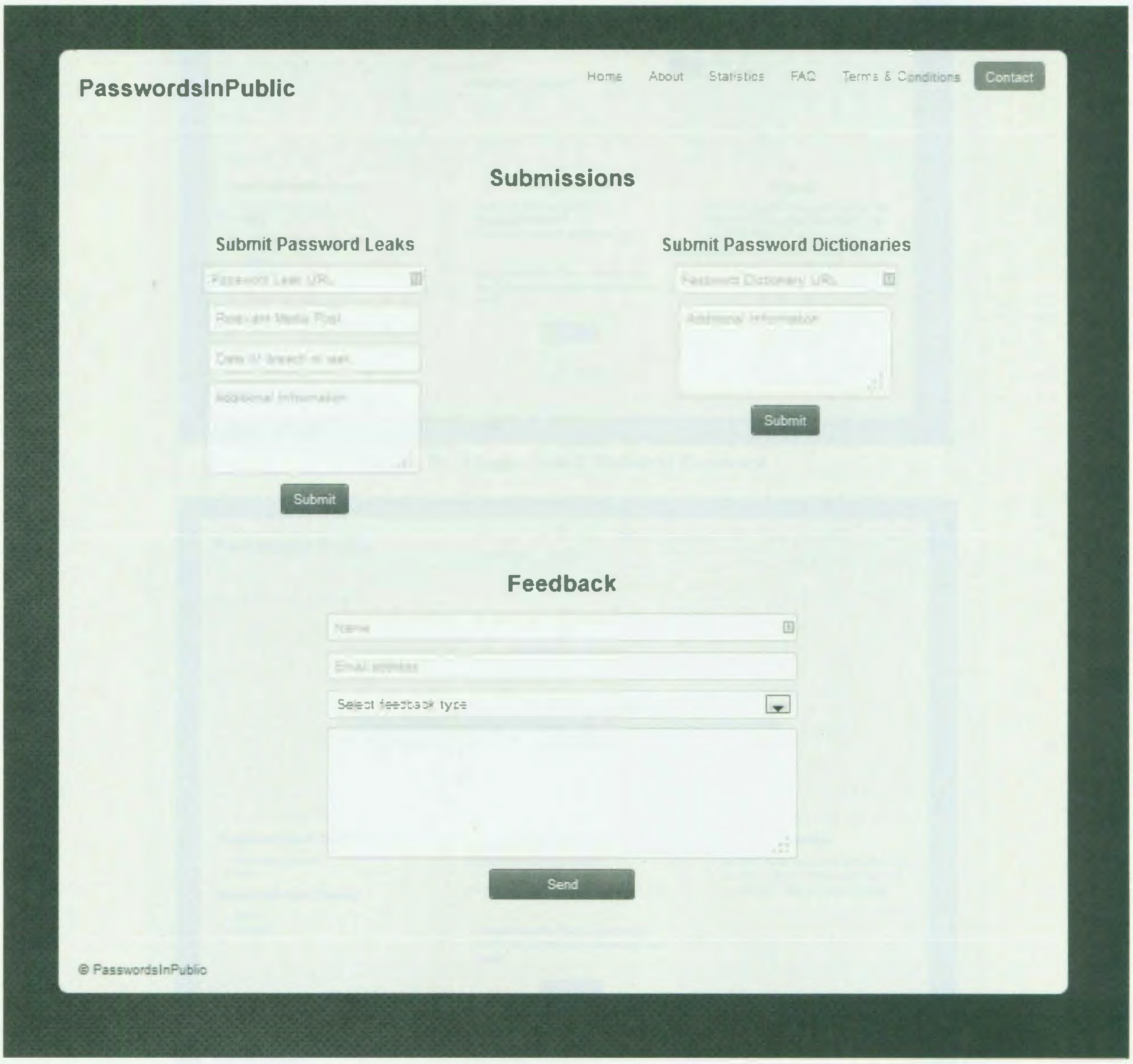
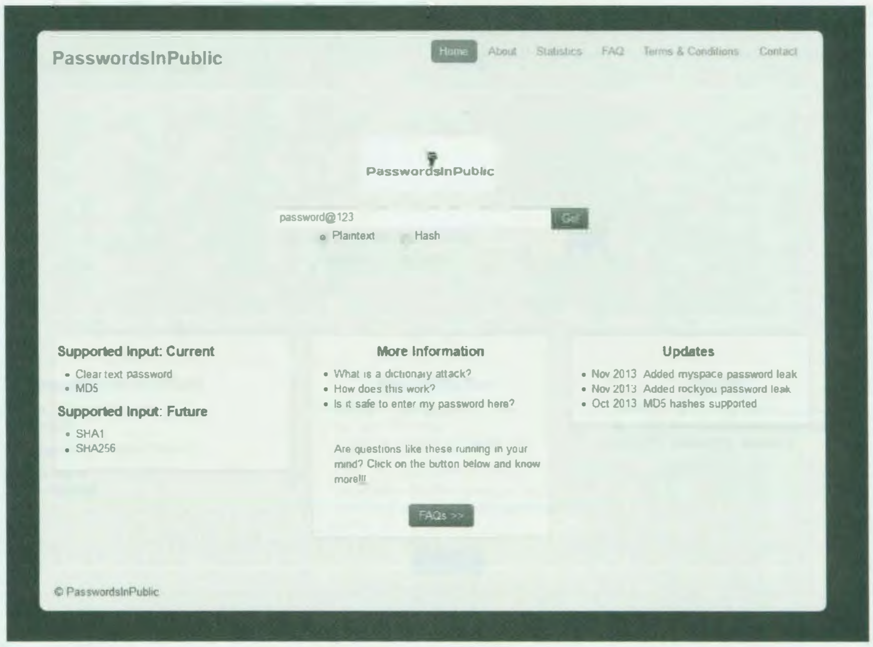


Figure 10: Website - Contact Page

Usage: Plain text based password query

The following screenshots demonstrate the query for a plain text password input and the result based on the password's presence in the password database built from leaks and dictionaries.



Supported Input: Current

- Clear text password
- MD5

Supported Input: Future

- SHA1
- SHA256

More Information

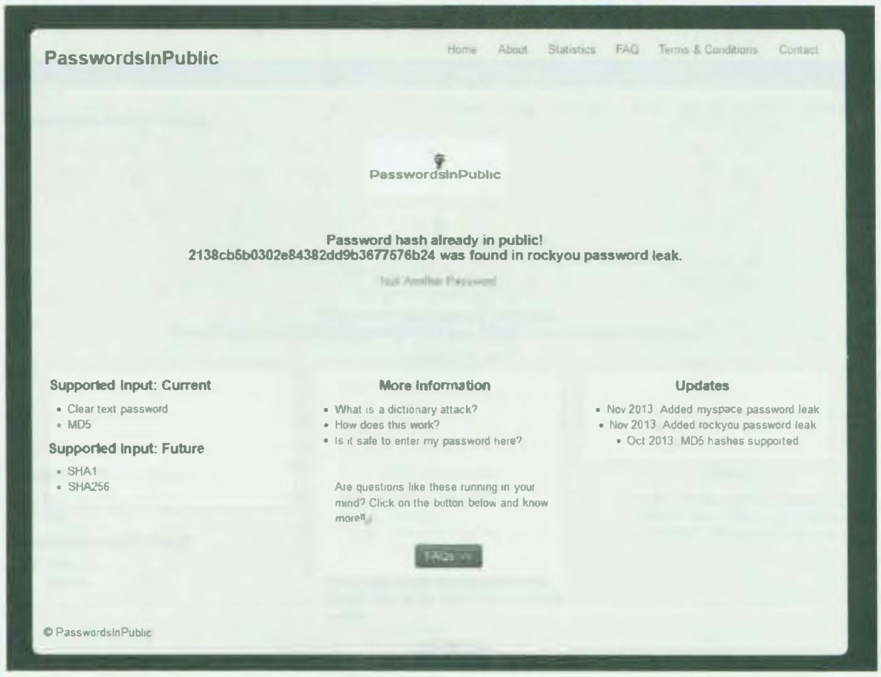
- What is a dictionary attack?
- How does this work?
- Is it safe to enter my password here?

Are questions like these running in your mind? Click on the button below and know more!!!

Updates

- Nov 2013 Added myspace password leak
- Nov 2013 Added rockyou password leak
- Oct 2013 MD5 hashes supported

Figure 11: Usage - Input Plaintext Password



Password hash already in public!
2138cb5b0302e84382dd9b3677576b24 was found in rockyou password leak.

Supported Input: Current

- Clear text password
- MD5

Supported Input: Future

- SHA1
- SHA256

More Information

- What is a dictionary attack?
- How does this work?
- Is it safe to enter my password here?

Are questions like these running in your mind? Click on the button below and know more!!!

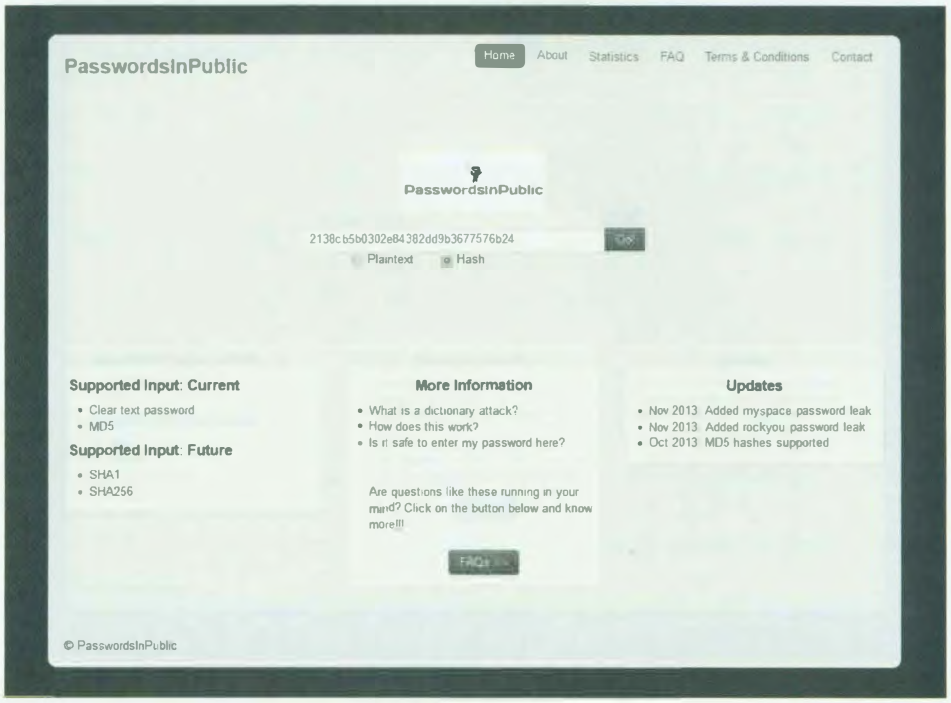
Updates

- Nov 2013 Added myspace password leak
- Nov 2013 Added rockyou password leak
- Oct 2013 MD5 hashes supported

Figure 12: Usage - Output Password Presence

Usage: Hash based Query

The following screenshots demonstrate the query for a hashed password input when the password is found in the password database built from leaks and dictionaries.



PasswordsInPublic

Home | About | Statistics | FAQ | Terms & Conditions | Contact

PasswordsInPublic

2138cb5b0302e84382dd9b3677576b24 [Go]

Plaintext [✓ Hash]

Supported Input: Current
- Clear text password
- MD5

Supported Input: Future
- SHA1
- SHA256

More Information
- What is a dictionary attack?
- How does this work?
- Is it safe to enter my password here?

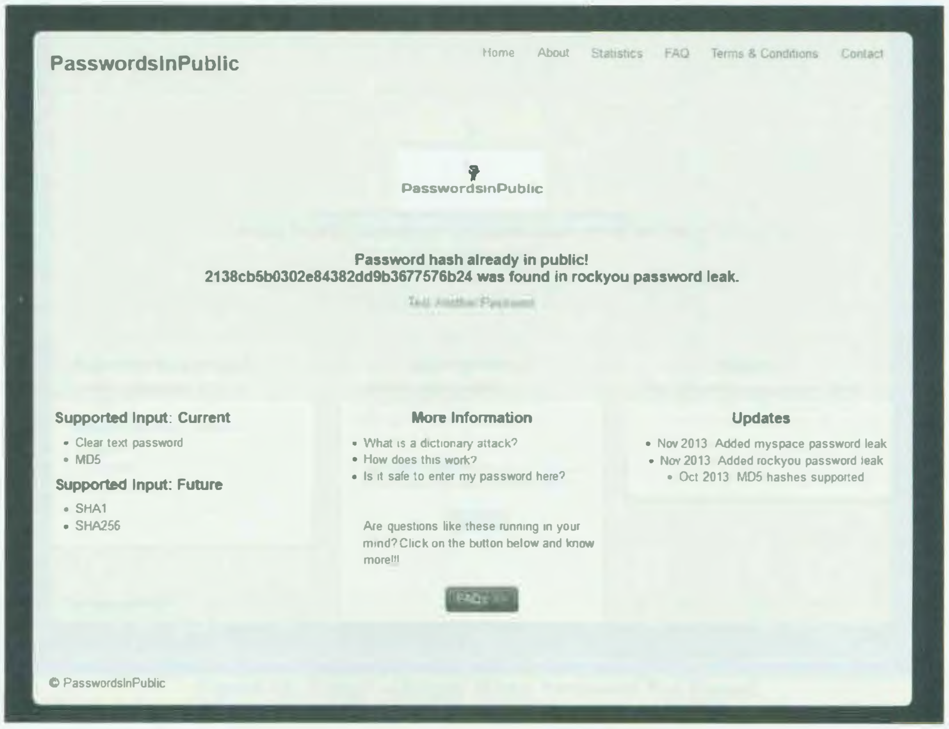
Are questions like these running in your mind? Click on the button below and know more!!!

[FAQs »]

Updates
- Nov 2013: Added myspace password leak
- Nov 2013: Added rockyou password leak
- Oct 2013: MD5 hashes supported

© PasswordsInPublic

Figure 13: Usage - Input Password Hash



PasswordsInPublic

Home | About | Statistics | FAQ | Terms & Conditions | Contact

PasswordsInPublic

Password hash already in public!
2138cb5b0302e84382dd9b3677576b24 was found in rockyou password leak.

Test Another Password

Supported Input: Current
- Clear text password
- MD5

Supported Input: Future
- SHA1
- SHA256

More Information
- What is a dictionary attack?
- How does this work?
- Is it safe to enter my password here?

Are questions like these running in your mind? Click on the button below and know more!!!

[FAQs »]

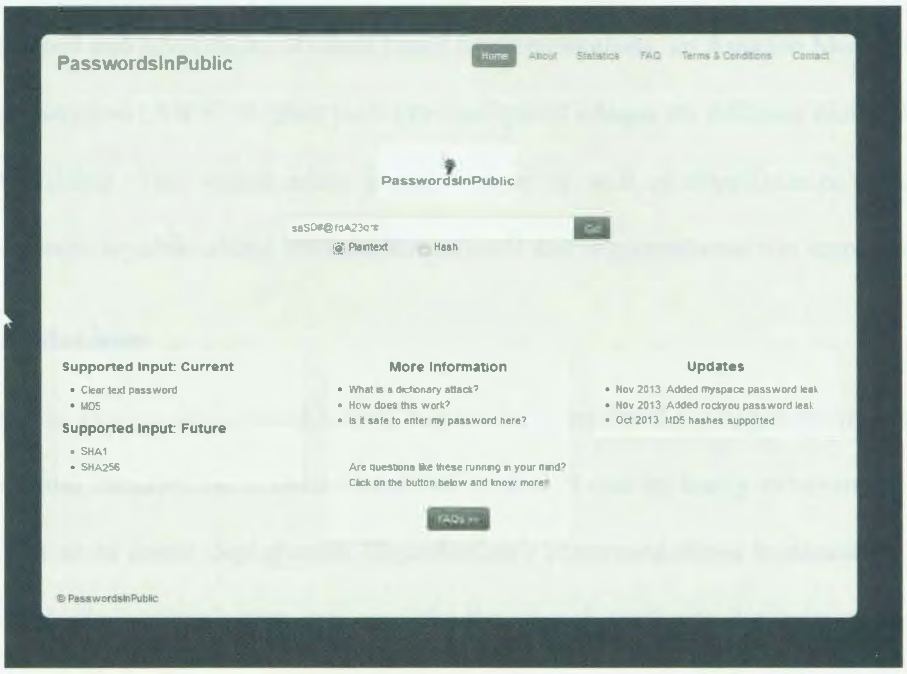
Updates
- Nov 2013: Added myspace password leak
- Nov 2013: Added rockyou password leak
- Oct 2013: MD5 hashes supported

© PasswordsInPublic

Figure 14: Usage - Output Password Presence

Usage: Password not found in database

The following screenshots demonstrate the query for a plain text password input when the password is not found in the password database built from leaks and dictionaries.



Supported Input: Current

- Clear text password
- MD5

Supported Input: Future

- SHA1
- SHA256

More Information

- What is a dictionary attack?
- How does this work?
- Is it safe to enter my password here?

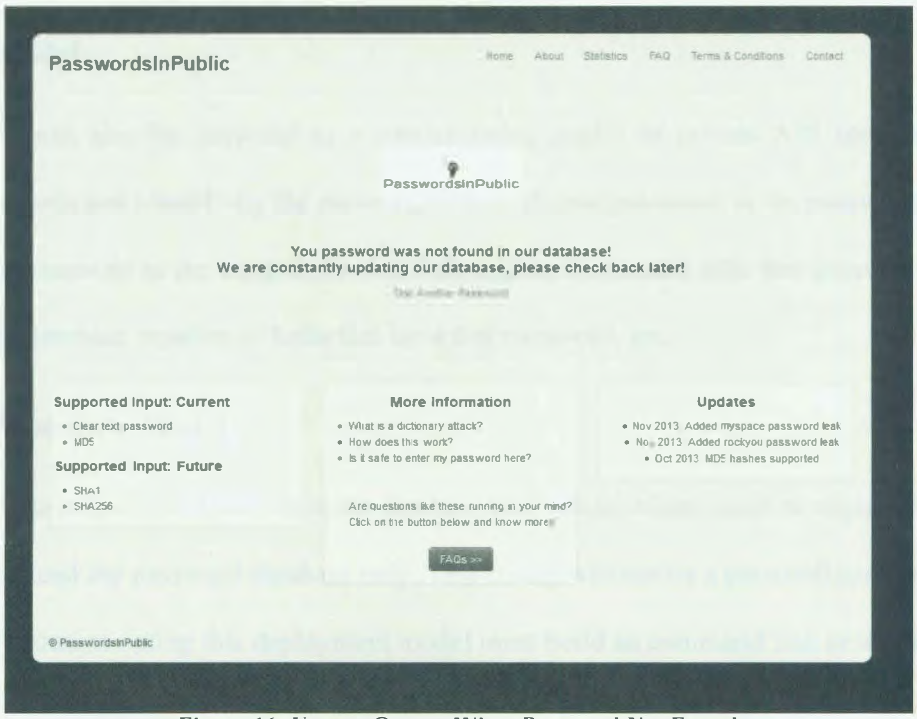
Are questions like these running in your mind?
Click on the button below and know more!!

Updates

- Nov 2013 Added myspace password leak
- Nov 2013 Added rockyou password leak
- Oct 2013 MD5 hashes supported

© PasswordsInPublic

Figure 15: Usage - Input Password



You password was not found in our database!
We are constantly updating our database, please check back later!

Supported Input: Current

- Clear text password
- MD5

Supported Input: Future

- SHA1
- SHA256

More Information

- What is a dictionary attack?
- How does this work?
- Is it safe to enter my password here?

Are questions like these running in your mind?
Click on the button below and know more!!

Updates

- Nov 2013 Added myspace password leak
- Nov 2013 Added rockyou password leak
- Oct 2013 MD5 hashes supported

© PasswordsInPublic

Figure 16: Usage - Output When Password Not Found

3.5 Application Deployment Models

The application can be deployed using different models as outlined below:

3.5.1 Cloud Instance

With the convenience and advantages of cloud based implementations, an Amazon Machine Image (AMI) on Amazon Web Services (AWS) or other such pre-configured images on different cloud platforms would provide great flexibility. This would allow a regular user as well as organizations to launch the pre-configured passwords-in-public cloud instance for personal and organizational use respectively.

3.5.2 Virtual Machine

Virtual Appliance is another deployment model that can be considered for organizations. The application can be packaged and installed on a virtual machine so that it can be easily setup in the organization's private data center or in house deployment. Organization's concerned about transmitting password over the network outside their organization can consider the virtual appliance deployment model. Different virtual appliances can be built to support variety of virtualization platforms like VMware, KVM, etc.

3.5.3 API Model

The application can also be provided as a service using public or private API for querying user or employee passwords and identifying the presence of their chosen password in the password database. The API can accept password as the input and return information associated with that password like presence in the password database, number of leaks that have this password, etc.

3.5.4 Database Only Model

The last model described in this section is the database only model where users or organizations would be allowed to download the password database only. This model will not be a pre-configured application and the user or organization opting this deployment model must build an command line or web based utility to interact with the application.

The deployment models cloud instance, virtual appliance and database only can be bundled with an update script that would allow users to update their database periodically. Additionally, these models can also be leveraged to build a scalable implementation. The cloud instance deployment model would allow organizations to spin multiple instances to cater to the needs of a large user base. Similarly, they can deploy multiple virtual appliances to on their virtualization platform when option for virtual appliance deployment model while the database only deployment model would allow the users to clone or replicate the password database to serve large user requests.

4 STATISTICS

The proof of concept for this paper is based on our database that included a sample of passwords (CommonDB column) harvested and gathered from easily available password leak files and commonly used dictionaries. The commonly used dictionaries considered for this test included john-the-ripper [27], cain [28], twitter's banned passwords and 500 worst passwords [29]. To demonstrate the potential of the concept presented in this paper, we also considered Crackstation dictionary [25] which is a popular password dictionary that has "1212356398" passwords gathered from various sources.

Table 1 presents the statistics that compare how an online service provider (Google), an online password strength estimating utility (passwordmeter.com) and password databases built for this paper (CommonDB and CrackstationDB) fared when tested using password leak files listed in Leak Name column. Actual password leaks contain real passwords of people and were hence considered as a basis for the comparison.

For the comparison, the number of passwords from password leak files identified as weak were noted for Google's password rating mechanism and for PasswordMeter's password rating website while on the other hand the number of passwords from leak files found in CommonDB and CrackstationDB were noted.

Table 1: Password Strength Comparison

Leak Name	Total Passwords	Google	SimpleDB	Password Meter	CrackstationDB
Alypaa	1384	35.48	71.37	98.99	100
Carders	1904	26.79	68.54	80.73	93.22
Elite Haccker	895	45.81	88.16	98.99	100
Facebook (Pastebay Malware)	55	20	49.09	87.27	100
Facebook (Phished)	2442	13.96	100	76.6	99.84
Faithwriters	8347	19.98	59.6	26.67	100
Hak5	2351	9.23	24.97	60.95	100
MySpace	37144	8.06	49.92	76.82	99.97
PhpBB	184389	7.35	40.29	79.49	100
Porn Unknown Site	8088	30.46	75.64	95.57	100
Singles.org	12233	22.11	100	91	100
Ultimate Strip Club	38820	14.34	51.65	81.84	100
Rockyou	12413667	0.37	1.95	72.91	99.97

The statistics demonstrated that the percentage of passwords identified as weak by Google's password rating mechanism (Google) was significantly lower than the percentage of passwords that are part of other leak files and common dictionaries (CommonDB). On the other hand, the password rating mechanism at passwordmeter.com (PasswordMeter) fared significantly better than Google and CommonDB. The concept of password database was expanded from CommonDB to CrackstationDB to better demonstrate the potential of the application. The statistics for CrackstationDB presented an average of 99.4% success rate in terms of finding a password in the Crackstation dictionary thereby demonstrating the best performance.

These statistics justify that existing password strength evaluating applications and online service's password strength estimator primarily focus on parameters like entropy, length and/or complexity and do not consider dictionary attacks while evaluating user passwords, thereby providing incomprehensive password strength estimation.

5 FUTURE WORK

The existing implementation does not incorporate other password strength determination parameters for determining the strength level of the password selected by the user. Future work could be integrating demonstrated approach that focuses on dictionary-based attacks with existing password strength evaluators that primarily focus on entropy, complexity and randomness. Integrating the demonstrated technique into password settings of applications, password setup routines of operating systems, online web services and password strength indicators can also be considered as an avenue of future work. The technique discussed in this paper can also be integrated with password managers (LastPass, Browser password managers, etc) to alert users when the passwords selected by them are found in the database.

6 CONCLUSION

The area of implementing authentication mechanisms, improving password based authentication mechanisms and assisting users in selecting secure passwords is under constant research. In addition to the improvement in password based authentication systems, password attack defense techniques and password strength evaluators, security researchers and evangelists must make efforts to continually educate and guide users about alternate authentication mechanisms, selection of passwords and the various password based attack techniques. The technique demonstrated in this paper incorporates the concept of guiding and enabling the users to select relatively secure passwords from a dictionary attack perspective by informing them about passwords that are part of publicly released password dictionaries and leaks. Integrating such a technique with existing password rating mechanisms would result in a thorough and comprehensive password strength estimation.

7 REFERENCES

[1] B. Pinkas and T. Sander, "Securing passwords against dictionary attacks," in Proceedings of the 9th ACM conference on Computer and communications security, ser. CCS '02. New York, NY, USA: ACM, 2002, pp. 161–170. [Online]. Available: http://doi.acm.org.ezproxy.rit.edu/10.1145/586110.586133

[2] C. Herley and P. Van Oorschot, "A research agenda acknowledging the persistence of passwords," Security Privacy, IEEE, vol. 10, no. 1, pp. 28–36, 2012.

[3] "The top cyber security risks." [Online]. Available: http://www.cs.vu.nl/ crispo/teaching/seceng2012/Assignment1/toprisk.pdf

[4] "How effective is a straigth dictionary attack," Feb. 2012. [Online]. Available: http://thepasswordproject.com/2012-02- 01 how effective is a straight dictionary attack

[5] "sourCEntral - PAM CRACKLIB." [Online]. Available: http://man.sourcentral.org/SLES11/8+pam cracklib

[6] H. Pomeranz, "Linux password security with pam cracklib." [Online]. Available: http://www.deer-run.com/ hal/sysadmin/pam cracklib.html

[7] S. Chakrabarti and M. Singhal, "Password-based authentication: Pre- venting dictionary attacks," Computer, vol. 40, no. 6, pp. 68–74, 2007.

[8] S. Bellovin and M. Merritt, "Encrypted key exchange: password-based protocols secure against dictionary attacks," in Research in Security and Privacy, 1992. Proceedings., 1992 IEEE Computer Society Symposium on, 1992, pp. 72–84.

[9] D. Jablon, "Extended password key exchange protocols immune to dictionary attack," in Enabling Technologies: Infrastructure for Collaborative Enterprises, 1997. Proceedings., Sixth IEEE Workshops on, 1997, pp. 248–255.

[10] G. Mori and J. Malik, "Recognizing objects in adversarial clutter: breaking a visual captcha," in Computer Vision and Pattern Recognition, 2003. Proceedings. 2003 IEEE Computer Society Conference on, vol. 1, 2003, pp. I–134–I–141 vol.1.

[11] S. Li, S. A. H. Shah, M. A. U. Khan, S. A. Khayam, A.-R. Sadeghi, and R. Schmitz, "Breaking e-banking captchas," in Proceedings of the 26th Annual Computer Security Applications Conference, ser. ACSAC '10. New York, NY, USA: ACM, 2010, pp. 171–180. [Online]. Available: http://doi.acm.org/10.1145/1920261.1920288

[12] S. Hocevar, "PWNtcha caca labs." [Online]. Available: http://caca.zoy.org/wiki/PWNtcha

[13] K. A. Kluever, "Breaking the paypal hip: A comparison of classifiers," 2008.

[14] "Quality of online password checkers," Mar. 2013. [Online]. Available: http://blog.online-domain-tools.com/2013/03/12/quality-of- online-password-checkers/

[15] "GeodSoft password Evaluator/Checker." [Online]. Available: http://geodsoft.com/cgi-bin/pwcheck.pl

[16] "Check passwords strength. online. free." [Online]. Available: http://www.getsecurepassword.com/CheckPassword.aspx

[17] "How secure is my password?" [Online]. Available: http- s://howsecureismypassword.net/

[18] S. Egelman, A. Sotirakopoulos, I. Muslukhov, K. Beznosov, and C. Herley, "Does my password go up to eleven?: the impact of password meters on password selection," in Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, ser. CHI '13. New York, NY, USA: ACM, 2013, pp. 2379–2388. [Online]. Available: http://doi.acm.org.ezproxy.rit.edu/10.1145/2470654.2481329

[19] Password checker - evaluate pass strength, dictionary attack. [Online]. Available: http://password-checker.online-domain-tools.com/

[20] "PwnedList.com." [Online]. Available: https://www.pwnedlist.com/

[21] "Should i change my password? j how safe is your password?" [Online]. Available: https://shouldichangemypassword.com/

[22] "Abusix leakdb project." [Online]. Available: http://leakdb.abusix.com/

[23] "Google password rater." [Online]. Available: https://www.google.com/accounts/RatePassword?Passwd=

[24] "Password strength checker." [Online]. Available: http://www.passwordmeter.com/

[25] "CrackStation - online password hash cracking - MD5, SHA1, linux, rainbow tables, etc." [Online]. Available: https://crackstation.net/

[26] "defuse/crackstation-hashdb GitHub." [Online]. Available: https://github.com/defuse/crackstation-hashdb

[27] "John the ripper password cracker." [Online]. Available: http://www.openwall.com/john/

[28] "oxid.it - cain & abel." [Online]. Available: http://www.oxid.it/cain.html

[29] "Passwords - SkullSecurity." [Online]. Available: https://wiki.skullsecurity.org/Passwords

8 Appendix

8.1 Appendix: Scripts

8.1.1 Password Importer

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

from __future__ import division
from warnings import filterwarnings
import MySQLdb
import hashlib
import sys
import argparse
import ConfigParser
import os
import pdb

# suppress warnings
filterwarnings('ignore', category=MySQLdb.Warning)

# number of passwords (in the password file being imported) that already
# exist in the database
def crackability_db(infile, dbcreds):
    ef = open('/var/www/bootstrap/statistics/entries.csv', 'a')
    leakname = infile.split(".")[0]
    hashtype = infile.split(".")[1]
    datevalue = int(infile.split("_")[2].split(".")[0])

    if hashtype == 'plaintext':
        hashtype = 'md5'

    # db connection - change the fucking user acess from root
    db = MySQLdb.connect(dbcreds[0], dbcreds[1], dbcreds[2])

    # Creating a cursor object
    cur = db.cursor()

    try:
    # Use database for the hash type the passwords belong to
    db.select_db(hashtype)

        # Build and execute query to retrieve tables for hash type of the password
        # file
    query = "SELECT table_name FROM INFORMATION_SCHEMA.TABLES WHERE
TABLE_SCHEMA = '%s'" % hashtype
        cur.execute(query)

    # retrieve all tables in that hashtype database
        tables = []
        for table in cur.fetchall():
            if leakname in table[0]:
                # print leakname
                continue
            tables.append(table[0])
        # print tables

    # actual calculation happens here
        crackcount, hashcount = 0, 0
        with open(infile, 'r') as cf:
            for hash in cf:
                hash = hash.rstrip()
                hashcount += 1
                for table in tables:
                    # if int(table.split("_")[2]) < datevalue:
                    query = "SELECT hashvalue FROM %s WHERE hashvalue = '%s' LIMIT 1" %
                        (table, hash)
                    res_rows = int(cur.execute(query))
                    if res_rows == 1:
                        print query
                        crackcount += 1
                        break
                    elif res_rows == 0:
                        continue

            print "=========================================="
            print "Final: ", crackcount, hashcount
            crackability = (crackcount / hashcount) * 100
            print crackability
            print leakname + ": " + str(crackability)
            entry = leakname + "." + \
                str(crackcount) + "." + str(hashcount) + "." + str(crackability) + "\n"
            ef.write(entry)
            print "Completed"
            print "=========================================="

        except:
            print "Error!"


def import_passwords(dbcreds, infilename):
        # convert password to md5 if its plaintext, else let it remain in existing
```

```
    # hashtype
    if infilename.split(".")[1] == 'plaintext':
            hashtype = 'md5'
        else:
            hashtype = infilename.split(".")[1]
            pass
        tablename = infilename.split(".")[0]
        infilepath = str(os.getcwd())+"/"+infilename
        # print hashtype, tablename, infilepath

        # establish db connection
        db = MySQLdb.connect(dbcreds[0], dbcreds[1], dbcreds[2])

        # creating a cursor object
        cursor = db.cursor()

        try:
            # Create database
            sql_cd = "CREATE DATABASE IF NOT EXISTS %s" %hashtype
            print "\n [+] import:",sql_cd

            # Use database
            db.select_db(hashtype)

            # Create table
            sql_ct = "CREATE TABLE %s (hashvalue VARBINARY(60) NOT NULL. UNIQUE
(hashvalue))" %(tablename)
            print "\n[+] import:",sql_ct

            # LOAD DATA sql command
        sql_ld = "LOAD DATA INFILE '%s' INTO TABLE %s FIELDS TERMINATED BY ' ' LINES
TERMINATED BY '\n'" %(infilepath,tablename)
        print "\n[+] import:",sql_ld
        # sys.exit()

        # executing the statement
        cursor.execute(sql_cd)
        cursor.execute(sql_ct)
        cursor.execute(sql_ld)
        db.commit()

    except MySQLdb.Error, e:
        # rollback in case of error
    if db:
      db.rollback()
    print "Error: %d: %s" %s(e.args[0], e.args[1])
    sys.exit(1)

  finally:
    # close db connections
      db.close()


# Converts the password file into an INFILE csv as per table schema
def build_infile(pwdfile, details):
    # details list order - details = [source_name,source_type,source_url,date,hash_type]
    infilename = "%s_%s_%s.%s.temp" %(details[1],details[0],details[3].replace('-',''),details[4])

  try:
      inf = open(infilename,'w')
    with open(pwdfile, 'r') as pf:
```

```
                    for each in pf:
                            print each

                            # strip trailing \n
                            each = each.rstrip()

                            if details[4] == 'plaintext':
                                    if each.strip():
                                            hash_value = hashlib.md5(each).hexdigest()
                                            print each, hash_value
                                    else:
                                            continue

                            else:
                                    hash_value = each

                            inf.write(hash_value)
                            inf.write("\n")

    except:
        pf.close()
            inf.close()

    finally:
            pf.close()
            inf.close()
        return infilename


def main():
        # arguments options
    parser = argparse.ArgumentParser(description='Import password dictionaries or leaks into the
password database')
    parser.add_argument('-d', '--dict', help='password dictionary filename (absolute path if not in current
directory)')
    parser.add_argument('-l', '--leak', help='password leak filename (absolute path if not in current
directory)')
    parser.add_argument('-f', '--format', help='Password format in the dictionary\nValid options:
plaintext | md5 | sha | etc')
    parser.add_argument('-date', '--date', help='date of password leak')
    parser.add_argument('-url', '--url', help='URL for dictionary | leak source | media post about leak if
no leak source')
    parser.add_argument('-s', '--stats', help='Perform statistics calculation only | No infile generation | No
upload | Existing infile must be provided')
    parser.add_argument('-b', '--build', help='Use this for building infile')
    args = vars(parser.parse_args())
    print args, len(args)


    if args['stats']:
        # building db configuration list
    config = ConfigParser.ConfigParser()
    config.read("conf/importer.conf")
    host = config.get('dbaccess','dbhost')
    user = config.get('dbaccess','dbuser')
    pwd = config.get('dbaccess','dbpass')
    dbcreds = [host, user, pwd]

        infile = args['stats']
        crackability_db(infile, dbcreds)

    elif len(args) != 7:
```

```
    print "Arguments Error"

else:
    # password dictionary file
    try:
        if args['dict']:
            pwdfile = args['dict']
            type = 'dict'
        elif args['leak']:
            pwdfile = args['leak']
            type = 'leak'
        else:
            raise Exception("Arguments Error")
    sys.exit()

        # building db configuration list
        config = ConfigParser.ConfigParser()
        config.read("conf/importer.conf")
        host = config.get('dbaccess','dbhost')
        user = config.get('dbaccess','dbuser')
        pwd = config.get('dbaccess','dbpass')
        dbcreds = [host, user, pwd]

        # building dictionary|leak characteristics
        date = args['date']
        source_name = args[type].split("/")[-1].split(".")[0]
        source_type = type
        source_url = args['url']
        hash_type = args['format']
        details = [source_name,source_type,source_url,date,hash_type]
        #print "\nmain ",details
```

```
                # call for building INFILE
                infilename = build_infile(pwdfile, details)

                # call for insert data into database
                import_passwords(dbcreds,infilename)

                # calculate and print crackability after import
                crackability_db(infilename, dbcreds)


            except IOError:
                print "File doesn't exist or incorrect file path"

        except:
            print "Error!!!"


if __name__ == '__main__':
        main()
```

8.1.2 Google Password Rating URL - Stats Generator

```
#!/usr/bin/python
import requests
import sys
import urllib
import subprocess
import time
from collections import defaultdict

def stats():
    '''
    #labels
    WEAK = 1
    MEDIUM = 2
    STRONG = 3
    VERYSTRONG = 4
    '''
    filename = sys.argv[1]
    pf = open(filename, 'r')
    count = defaultdict(int)


    for pwd in pf.readlines():
        url = 'http://www.google.com/accounts/RatePassword?Passwd=' \
                + urllib.quote_plus(pwd.rstrip())
        try:
            print pwd, url
            r = requests.get(url)
            if r.status_code == 104:
                r = temp_sleep(url)
        elif r.status_code == 200:
            res = r.text.rstrip()
            if res == '1':
                count['Wcount'] += 1
            elif res == '2':
                count['Mcount'] += 1
            elif res == '3':
                count['Scount'] += 1
            elif res == '4':
                count['VScount'] += 1
            else:
                count['unknown'] += 1

        else:
            print "HTTP status code is %s" % r.status_code
            print r.text
            count['not104not200'] += 1

    except Exception, e:
        print repr(e)
        time.sleep(30)

    print count

def temp_sleep(url):
    print "[+] Sleeping for 5 mins"
    time.sleep(300)
    r = requests.get(url)
    if r.status_code == 104:
        temp_sleep(url)
        elif r.status_code == 200:
            return r


def main():
    stats()



if __name__ == '__main__':
    main()
```

8.1.3 PasswordMeter.com - Stats Generator

```
#!/usr/bin/python
import sys
from pyvirtualdisplay import Display
from splinter import Browser

'''
    #labels
    WEAK = 1
    MEDIUM = 2
    STRONG = 3
    VERYSTRONG = 4
'''

def stats():

  display = Display(visible=0, size=(800, 600))
  display.start()

  filename = sys.argv[1]
  pf = open(filename,'r')
  count = {'VWcount':0, 'Wcount':0, 'Gcount':0, 'Scount':0, 'VScount':0, 'Unicode':0}

  browser = Browser('firefox')
  fileurl = "file:///path/to/passwordmeter.html"
  browser.visit(fileurl)



  for pwd in pf.readlines():
    pwd = pwd.rstrip()
```

```
        if not pwd:
            continue

        pwdtype = is_ascii(pwd)

        if pwdtype:

            browser.fill('passwordPwd',pwd)
            result = browser.find_by_id('complexity').value

            if result  == 'Very Weak':
                count['VWcount'] += 1
            elif result == 'Weak':
                count['Wcount'] += 1
            elif result == 'Good':
                count['Gcount'] += 1
            elif result == 'Strong':
                count['Scount'] += 1
            elif result == 'Strong':
                count['VScount'] += 1

            print pwd+" : "+ result
            print count

        else:
            count['Unicode']+=1
            print count

browser.quit()
display.stop()
```

```
    print "==============Final======================="
    print count


def is_ascii(pwd):
    return all(ord(c) < 128 for c in pwd)

def main():
    stats()

if __name__ == '__main__':
    main()
```

8.1.4 Password Database - Stats Generator (CommonDB)

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

from __future__ import division
from warnings import filterwarnings

import ConfigParser
import argparse
import hashlib
import os
import sys
import MySQLdb


# suppress warnings
filterwarnings('ignore', category=MySQLdb.Warning)


def crackability_db(infile, dbcreds):
    ef = open('/var/www/bootstrap/statistics/entries.csv', 'a')
    fif = open('foundinfile.txt', 'a')

    leakname = infile.split(".")[0]
    hashtype = infile.split(".")[1]
    datevalue = int(infile.split("_")[2].split(".")[0])

    if hashtype == 'plaintext':
        hashtype = 'md5'
```

```
# db connection - change the fucking user access from root
db = MySQLdb.connect(dbcreds[0], dbcreds[1], dbcreds[2])

# Creating a cursor object
cur = db.cursor()

try:
    # Use database for the hash type the passwords belong to
    db.select_db(hashtype)

    # Build and execute query to retrieve tables for hash type of the
    # password file
    query = "SELECT table_name FROM INFORMATION_SCHEMA.TABLES WHERE
TABLE_SCHEMA = '%s'" % hashtype
    cur.execute(query)

    # retrieve all tables in that hashtype database
    tables = []
    for table in cur.fetchall():
        # if table for this password file already exists, ignore that table
        print leakname, table[0]
        if table[0] in leakname:
            print "table in leakname"
            continue
        tables.append(table[0])
    print tables

    # actual calculation happens here
    crackcount, hashcount = 0, 0
    with open(infile, 'r') as cf:
        for hash in cf:
                hash = hash.rstrip()
                hashcount += 1
                for table in tables:
                    # if int(table.split("_")[2]) [illegible] datevalue
                    query = "SELECT hashvalue FROM %s WHERE hashvalue = '%s' LIMIT 1" % (
                        table, hash)
                    res_rows = int(cur.execute(query))
                    if res_rows == 1:
                        print query
                        crackcount += 1
                        break
                    elif res_rows == 0:
                        continue

        print "=============================================="
        print "Final: ", crackcount, hashcount
        crackability = (crackcount / hashcount) * 100
        print crackability
        print leakname + ": " + str(crackability)
        entry = leakname + "." + \
            str(crackcount) + "." + str(hashcount) + \
            "." + str(crackability) + "\n"
        ef.write(entry)
        print "Completed"
        print "=============================================="

    except:
        print "Error!"


def main():
    # arguments options
    parser = argparse.ArgumentParser(
        description='Import password dictionaries or leaks into the password database')
    parser.add_argument(
        '-d', '--dict', help='password dictionary filename (absolute path if not in current directory)')
    args = vars(parser.parse_args())

    infilename = args['dict']

    # building db configuration list
    config = ConfigParser.ConfigParser()
    config.read("conf/importer.conf")
    host = config.get('dbaccess', 'dbhost')
    user = config.get('dbaccess', 'dbuser')
    pwd = config.get('dbaccess', 'dbpass')
    dbcreds = [host, user, pwd]

    # calculate and print crackability after import
    crackability_db(infilename, dbcreds)


if __name__ == '__main__':
    main()
```

8.1.5 Crackstation Dictionary - Stats Generator (CrackstationDB)

```
<?php

  xdebug_disable();
  error_reporting(0);
  require_once('LookupTable.php');

  $file=fopen($argv[1], "r");
  $stats_file = "stats.txt";

  $md5 = new LookupTable("crackstation-md5.idx", "crackstation.txt", "md5");
  $total = 0;
  $count = 0;

  while(!feof($file))
  {
    $each_pwd = fgets($file);
    $each_pwd = trim ($each_pwd);
    $total = $total + 1;

    $to_crack = md5($each_pwd);
    #print $each_pwd . $to_crack . "\n";

    $result = $md5->crack($to_crack);
    #var_dump($result);

    if ($result !== FALSE) {
      $count = $count + 1;
      echo "Cracked: " . $result[0] . " Count:" . $count . " Total:" . $total . "\n";
    }
  }

  fclose($file);
  $crack_percentage = ($count/$total) * 100;
  $crack_percentage = round($crack_percentage, 2);
  $result_final =
"=====================================Final============================
Password File: $argv[1]
Total Cracked Count: $count
Total passwords in $argv[1]: $total
Percentage Cracked: $crack_percentage
======================================================================
";
  print $result_final;
  file_put_contents($stats_file, $result_final, FILE_APPEND | LOCK_EX);

?>
```

8.1.6 Harvester for Datalossdb

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

from __future__ import division
from datetime import datetime
from math import ceil
from mechanize import Browser
import bs4
import re
#import cookielib

# define urls and output file
urls = {'dbloss':
        'http://datalossdb.org/search?data_type%5B%5D=PWD&direction=asc&order=reported_date'}
target_file = open('datalossdb_urls.list', 'a+')

# initialize browser
b = Browser()
b.set_handle_robots(False)
b.addheaders = [('User-agent', 'Mozilla/5.0 (X11; U; \
                 Linux i686; en-US; rv:1.9.0.1) \
                Gecko/2008071615 Fedora/3.0.1-1.fc9 Firefox/3.0.1')]

# cookie jar, if required
#cj = cookielib.LWPCookieJar()
# b.set_cookiejar(cj)

# login to datalossdb
b.open('https://datalossdb.org/sessions/new')
```

```
b.select_form(nr=0)
b['login'] = 'tarunmadiraju'
b['password'] = 'wc*V91phR5j3lMeZ^8mv'
b.submit()

# retrieve html - retrieves first page
response = b.open(urls['dbloss'])
soup = bs4.BeautifulSoup(response.read())

# retrieving incidents from the website for the first time
 def get_first(total_incidents):
    print "get_first", total_incidents
    # pages
    pages = int(ceil(total_incidents / 20))
    print pages
    raw_input("Continue 5?")

    # get all pages for incidents tagged with PWD
    for page in xrange(1, pages + 1):
        page_url = "{pwd_base_url}&page={page}\
                ".format(pwd_base_url=urls['dbloss'], page=page)
        page_response = b.open(page_url)
        soup_in = bs4.BeautifulSoup(page_response.read())

        # write the URLs to the output file
        for each in soup_in.find_all('a'):
            if "/incidents/" in each['href']:
                # print each['href']
                target_file.write("http://datalossdb.org/{incident_url}"
                                  .format(incident_url=each['href']))
                target_file.write("\n")
    # Append Harvesting Summary
    print "Harvesting Summary = {date} : {incidents}" \
      .format(date=datetime.now().strftime('%Y%m%d'), incidents=new_count)
    target_file.write("Entry = {date} : {incidents}"
                .format(date=datetime.now().strftime('%Y%m%d'),
                        incidents=total_incidents))
    target_file.write("\n\n")
    target_file.close()


# retrieving newly reported incidents only
def get_new(new_count, old_count):
    pages_from = int(ceil(old_count / 20))
    pages_to = int(ceil(new_count / 20))

    # retrieve new pages for incidents tagged with PWD
    for page in xrange(pages_from, pages_to + 1):
        page_url = "{pwd_base_url}&page={page}\
                ".format(pwd_base_url=urls['dbloss'], page=page)
        page_response = b.open(page_url)
        soup_in = bs4.BeautifulSoup(page_response.read())

        # write the URLs to file
        for each in soup_in.find_all('a'):
            if "/incidents/" in each['href']:
                # print each['href']
                target_file.write("http://datalossdb.org/{incident_url}"
                          .format(incident_url=each['href']))
                target_file.write("\n")
        target_file.write("\n")

    # Append harvesting summary
    print "Harvesting Summary = {date} : {incidents}" \
        .format(date=datetime.now().strftime('%Y%m%d'), incidents=new_count)
    target_file.write("Entry = {date} : {incidents}"
                .format(date=datetime.now().strftime('%Y%m%d'),
                    incidents=new_count))

    target_file.write("\n\n")
    target_file.close()


def main():
    # total number of incidents
    for each_span_tag in soup.findAll('span'):
        if "Displaying Incident" in each_span_tag.text:
            current_incident_count = int(
                each_span_tag.findChildren('b')[1].text)

    # search for entries already retrieved, print previous harvesting summary
    target_file.seek(0, 0)
    file_data = target_file.read()
    entries = re.findall(r'Entry = .*', file_data)
    print "[+] Incidents harvested previously: {entries}".format(entries=entries)

    # check if new incidents have been reported
    if entries:
        old_incident_count = int(entries[-1].split(":")[1])
        diff = current_incident_count - old_incident_count
        if diff:
            print "[+] New incidents added. Difference:", diff
            get_new(current_incident_count, old_incident_count)
        else:
            print "[+] No new password incidents"
    elif not entries:
        get_first(current_incident_count)


if __name__ == '__main__':
    main()
```

8.1.7 Harvester for Twitter

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-


import codecs
import sys
import tweepy


# oath required keys and secrets
CONSUMER_KEY = "--"
CONSUMER_SECRET = "--"
ACCESS_KEY = "--"
ACCESS_SECRET = "--"


# authentication using tweepy oath
auth = tweepy.OAuthHandler(CONSUMER_KEY, CONSUMER_SECRET)
auth.set_access_token(ACCESS_KEY, ACCESS_SECRET)
api = tweepy.API(auth)


# PwnMagic account follows users and bots that publish password leak information
# retrieve friends in user account 'PwnMagic'



def get_accounts():
    accounts = {}
    friends = api.get_user('PwnMagic').friends()
    for friend in friends:
        accounts[friend.screen_name] = friend.id
    return accounts
```

```
# gather tweets

def tweets(account, tf):
    # list of tweets
    tweets_all = []

    #tweepy - tweets
    tweets_all.append(api.user_timeline(id=account, count=200))

    # maximum requests using API (3600)
    # can be tweaked to get tweets as per our requirement
    request = 0
    while request < 18:
        maxid = tweets_all[request].since_id
        tweets_all.append(
            api.user_timeline(id=account, count=200, max_id=maxid))
        request += 1
        if maxid == tweets_all[request].since_id:
            break

    tf.write("== {acc} ==\n".format(acc=account))
    for tweets in tweets_all:
        for tweet in tweets:
            tf.write(u'{tweet}'.format(tweet=tweet.text))
            # print tweet.text
    tf.write("== End of {acc} ==\n".format(acc=account))
    tf.write("\n\n")


    # Perform appropriate filtering to minimize the number of tweets
    # Currently no filtering to gather maximum data
    # Prototype of filter function included
    '''
    #PastebinLeaks Specific Filter
    for tweets in tweets_all:
        for tweet in tweets:
            if any(i in tweet.text for i in ['pass', 'password']):
                print "[+] Password leak: ", tweet.text
            else:
                #print "[-] Other leak: ",tweet.text
                pass
    '''


def main():
    #get all users PwnMagic follows currently
    accounts = get_accounts()
    tf = codecs.open('tweets_file.txt', 'w', 'utf-8')

    #retrieve tweets of PwnMagic friends
    for account in accounts:
        tweets(account, tf)



if __name__ == '__main__':
    main()
```

8.1.8 Harvester for SkullSecurity

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import bs4
from mechanize import Browser

# define URLs
urls = {'skullsecurity': 'http://downloads.skullsecurity.org/passwords/'}
target_file = open('skullsecurity_urls.list', 'w')

# initialize browser
br = Browser()
br.set_handle_robots(False)
br.addheaders = [
    ('User-agent', 'Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.9.0.1) Gecko/2008071615 Fedora/3.0.1-
1.fc9 Firefox/3.0.1')]

# request website
response = br.open(urls['skullsecurity'])
soup = bs4.BeautifulSoup(response.read())

# retrieve password file name
for each in soup.find_all('a'):
    if "/" not in each.string:
        target_file.write(each.string)
        target_file.write("\n")
```