Lightweight protection against brute force login attacks on web applications

Carlisle Adams, *Senior Member, IEEE,* Guy-Vincent Jourdan, *Member, IEEE*,
Jean-Pierre Levac and François Prevost
School of Information Technology and Engineering
University of Ottawa
Ottawa, Canada
cadams@site.uottawa.ca, gvj@site.uottawa.ca, jleva084@uottawa.ca, fprev019@uottawa.ca

Abstract— **Password-based systems and, more generally, authentication systems based on *something you know*, are commonplace on the Internet. Web applications using these systems can be the target of brute force login attacks, in which an attacker tries to compromise a given account or any user account on the system. These applications rarely implement effective protection mechanisms against these attacks. In this paper, we review the situation and propose a practical, simple, security mechanism. Our system is non-intrusive and can be incorporated into most web applications with very little modification to the application code.**

Keywords- web applications, brute force attacks, trawling attacks, denial of service

I. Introduction

Password-based authentication mechanisms are both extremely common and highly criticized in the world of computer systems. They are criticized because they do not provide adequate protection in practice. For example, in [1] Bishop and Klein, reporting on an experiment conducted in the late 1980s, explained how they attempted to crack a set of nearly 14,000 real Unix passwords gathered from various sources. To perform their attack, they used a variety of brute force methods and some more adaptive techniques. They report that they were able to crack about 3.2% of the password set in the first 15 minutes, and that they cracked about 40% of the data set in about three CPU years (about 21% of them in the first week). Their result confirmed the already well known fact that human users tend to pick poor passwords (in that these passwords are relatively easy to guess), as documented, for example, in the late 1970s by Morris and Thomson [2]. However, in retrospect, these early results were in fact encouraging, even though they were not presented as such by the authors. Indeed, for example, Bishops and Klein were able to crack *only* 40% of the passwords in 3 CPU-years! The main conclusion that could be drawn at the time was that passwords as such were fine, as long as "good ones" were chosen. Unfortunately, the situation has become significantly worse since then: despite years of user education, strong evidence shows that end users still tend to use easy-to-guess passwords. It is difficult to blame end users for this, since the number of passwords to remember has ballooned to several dozen for a typical user, making it very impractical to expect people to pick hard-to-guess, yet memorable, passwords for so many accounts. But the main problem lies with password cracking techniques and computer speed. It is very likely that Bishop and Klein would now successfully guess the vast majority of their database, and do so in much less than three CPU years. It is also the case that they would have easy, cheap access to a lot more than three CPU years to run their attack if they wished. It can be argued, and it sometimes is, that the times when it was possible to create a password that a human can remember and that a determined, well equipped attacker cannot crack are behind us[1].

Even though password-based authentication systems are known not to provide very good security, they are still very popular, and represent the vast majority of authentication systems that are deployed currently. The reasons for this are simple: despite its poor security value, a password-based authentication mechanism is very easy to deploy, does not require any additional hardware, and is well accepted amongst the potential user population. It is arguably the simplest, most cost effective solution to use when you have to authenticate users with some minimal level of security. Because of this, these systems appear to be here to stay.

This bleak situation may not be as bad as it first seems. It is true that password-based authentication mechanisms do not provide adequate protection against well equipped attackers, but this analysis is based on so-called "offline" attacks, in which the attackers have direct access to some encrypted version of the passwords. In this situation, the only thing that slows down the attack is the generation of encrypted passwords on the one hand, and the comparison of the result with the list on the other hand. Thankfully, this situation is not the most common one. If we assume that the attacker does not have access to the database of passwords directly, then the attack must be performed online, through the system's legitimate gateway. The situation is very different in that case: for one thing, it is likely much slower, and for another thing designers of the system have an opportunity to act against such attacks

This work was supported in part by the Natural Sciences and Engineering Research Council of Canada

[1] For an interesting discussion regarding techniques followed by "professional password crackers", that is, vendors of password recovery solutions, see the overview published by Schneier in Wired in 2007: http://www.wired.com/politics/security/commentary/securitymatters/2007/01/72458

(e.g., by slowing them down or by stopping them completely). In fact, in a recent paper published in 2007 tellingly titled "Do strong web passwords accomplish anything?", Florêncio, Herley and Coskun point out that very little actual entropy is required to adequately protect Web applications as long as the right mechanisms are in place on the server side to prevent brute force attacks [3]. Unfortunately, one can fairly assume that for most current Web applications, the desired "right mechanisms" are actually not in place.

In this paper, we try to address this situation: we define what properties such a mechanism must possess, and we then propose a very simple solution, along with an implementation in Java. Our solution is meant to enhance very significantly the security of Web applications when it comes to protecting against online brute force attacks, and is designed in such a way that it can be easily integrated into both new and existing systems.

The paper is organized as follows: in Section II, we review the situation as it is currently: the types of attacks Web application are facing in the area of brute force attacks, the types of protections that are typically in place and their shortcomings, and what would be required for a good protection system. In Section III, we provide an overview of our proposed solution, as a combination of three simple ideas: separate the protection from the authentication; apply protection along all possible entry points; and use a sliding window approach when protecting. We describe an implementation of our system in Section IV, and we discuss the strengths and weaknesses of our solution in Section V.

II. The Problem To Solve

A. Various Types of Brute Force Attacks

There are different types of (on-line) brute force attacks on authentication systems based on something the user knows. The first one that comes to mind is the so called "targeted attack", where the attacker is trying to guess, using some kind of brute force strategy, the value (e.g., the password) that would authenticate a given user. In this case, the success of such an attack depends on the strength of the password used on the attacked account. The brute force strategy followed might be totally blind (e.g., try every possible password in some domain space), or it could be directed by various heuristics (such as trying all words in a dictionary for example). Whatever the strategy is, with a targeted attack the chosen account is going to receive a large number of attempted logins. This is the type of attack for which it is most common to find some level of built-in defense in existing systems, although typically the defense is not a very good one, as will be discussed below.

The second type of attack, sometimes called a "trawling attack", can be seen as the reverse of a targeted attack: in this case, the attacker chooses a password and tries to find a user account that uses this password. This kind of attack will typically be successful if, on the one hand, some accounts do use simple "common" passwords (see section II B. for a discussion of trawling attacks) and if the namespace used for the account identifier is either known or easy to guess. In [3], Florêncio, Herley and Coskun suggest spreading the entropy somewhat evenly between the account identifier and the password to make this kind of attack more difficult. This is, however, rarely done in practice and the domain space used for the account identifier is usually much less constrained than the one used for passwords in systems that constrain the domain space at all. It seems that many of the systems deployed today have no real defense mechanism to speak of against trawling attacks, and therefore are very vulnerable to it. As we discuss in Section II C., there are technical reasons to explain this lack of built-in defense mechanism, and our solution will address this.

The third kind of brute force attack, which we call "blind", is an attack that searches both the account identifier name space and the password name space at the same time. Here again, the search might follow some heuristic strategy, or simply attempt to exhaust the spaces methodically. Systems frequently offer some weak level of protection using IP address tracking (see Section II C.) which, incidentally, is typically the only protection offered indirectly against trawling attacks.

Note that it would be a mistake to limit this discussion only to searching account identifiers and corresponding passwords. One very common authentication system is the "knowledge question" approach, where the system challenges the user with a personal question which must be answered correctly, according to an answer previously provided by the user. This approach is prevalent in particular for on-line email systems, as a backup strategy used to authenticate users that have forgotten their passwords. In effect, however, when such a secondary authentication system is available, it provides another entry point and must naturally be protected as effectively as the primary authentication system, for otherwise the entire security is downgraded to the level of the less secure one. It is worth noting that many systems do not have "knowledge question" backup systems, and instead offer an option to email the forgotten password to a previously provided email account. This of course is a false sense of security, since the provider of the email account itself will quite often have a "knowledge question" backup system, and so if this system is vulnerable to attacks then indirectly every system that emails passwords to it is vulnerable as well.

Securing knowledge question systems against targeted attack is notoriously difficult: on the one hand, the same kind of exhaustive search that was possible on a password can be done here, but usually there is a much smaller domain space. On the other hand, specialized target attacks can be carried out by searching the actual answer to the question on public record[2]. Recently, Bonneau, Just and Matthews pointed out in [4] that knowledge questions are also highly vulnerable to trawling attacks. Indeed, if the questions are for example "What is your mother's maiden name?" or "What was the last name of your favourite school teacher?" (two questions that were found to be common in the survey conducted in [4]) then, to an American user base at least, it is likely that "Smith", "John" or "Johnson" are frequent answers, simply because

[2] Notoriously, an email account of US vice-presidential candidate Sarah Palin was compromised by an attacker who found the answers to the account's knowledge questions online, for example on the *Sarah Palin* Wikipedia page.

these last names are the most common ones in the population. Therefore, if a system uses one of these questions as a knowledge question (or any question whose answer is a last name, for that matter), then an effective attack strategy is to try "Smith" on as many accounts as possible. Again, it is fair to suggest that most systems providing knowledge questions do not provide any serious protection mechanism against trawling attacks.

B. *About Trawling Attacks*

Trawling attacks have been discussed in the security literature, but are they actually carried out in practice? We offer the following indicators that suggest an affirmative answer to this question.

First, there is strong evidence that trawling attacks would in practice be quite effective.

- In October 2007, a set of 43,713 passwords was leaked from the MySpace social network. According to various reports[3], the most common password in the set, "password1", was used by 0.23% of the users.
- In September 2009, a set of 9,843 passwords was leaked from the Hotmail online email service. This time, according to the reports[4], the most common password in the set, "123456", was used by 0.65% of the users.
- In December 2009, a set of almost 32,000,000 passwords was leaked from the site RockYou.com. According to the reports[5], the most common password in the set, which was again "123456", was used by 0.90% of the users.
- According to the calculation presented in [4], the amount of entropy in knowledge questions is usually so low that it doesn't provide any significant security against trawling attacks. For example, they have calculated that a trawling attack on a knowledge question requiring a last name as an answer, and for which three guesses are allowed, will break into 1 in 84 accounts on average by using the most common last names!

Second, there is also some evidence that these attacks are actually carried out. Anecdotally, we are told that the network of a major university in Canada is routinely targeted by such attacks. More importantly, groups such as the WASC Distributed Open Proxy Honeypot Project[6] have reported instances of such attacks, for example a distributed brute force attack against the Yahoo email service[7].

[3] See http://www.the-interweb.com/serendipity/index.php?/archives/94-A-brief-analysis-of-40,000-leaked-MySpace-passwords.html

[4] See http://www.acunetix.com/blog/websecuritynews/statistics-from-10000-leaked-hotmail-passwords/

[5] See http://www.imperva.com/docs/WP_Consumer_Password_Worst_Practices.pdf

[6] See http://projects.webappsec.org/Distributed-Open-Proxy-Honeypots

[7] See http://tacticalwebappsec.blogspot.com/2009/09/distributed-brute-force-attacks-against.html

C. *Defense Mechanisms in Existing Systems*

Most systems do provide some kind of defense mechanism against brute force login attacks. Perhaps the most common one, sometimes referred to as the "three strikes rule", consists of blocking access to an account after a number of failed login attempts (usually three). The obvious problem with such a mechanism is that it provides an easy denial of service attack vector, since any attacker can trivially prevent any user from logging in by providing the wrong credential for that account enough times. Another point of interest to us is the questionable choice of number of attempts, which is usually very low (3 most of the time) before the defense mechanism kicks in. With such a low number, it is easy for legitimate users to get their own account locked out. Using a larger number of attempts before locking the account, say 20 or 30 attempts, would be a lot more user-friendly and should not significantly impact the security of the system[8].

A more appropriate mechanism, a variation of the "three strikes" rule, is to temporarily disable the account for some period of time after the set number of failed attempts (the time could be preset or it could increase with the number of consecutive failed attempts). This reduces the problem of the denial of service attack, but does not completely solve it since the attacker can regularly attempt logging in to deny access to legitimate users. Another variation on this idea is to provide a set minimum interval between two login attempts for an account. All of these variations suffer from at least two flaws: first, in most cases of which we are aware, only existing accounts are monitored. The reason for this is that the information is stored along with the account data in the database, and thus cannot be stored for non-existing accounts. This provides an easy way for an attacker to validate whether an account exists or not. More importantly, these mechanisms do not provide any kind of protection against trawling attacks.

Some systems will modify the authentication mechanism after a set number of failed attempts, for example by showing a CAPTCHA. This, however, simply at best increases the entropy, and sometimes not very effectively since it may be possible to write a specialized program to break the CAPTCHA itself; see, for example, [5] as a starting point on CAPTCHAs.

Other systems implement similar mechanisms, but based on the IP address of the requester rather than on the user account being accessed. This simple modification is more effective, since it can be used to prevent target, trawling and blind brute force attacks at once, and does not pose the same risk of denial of service attacks, since it is the IP address of the attacker that is banned, not the account being targeted. If well implemented, it also works with existing and non-existing accounts. This solution is however more difficult to employ since it does not integrate nicely with the application database and requires building and storing a list of IP addresses separately. This is likely why it does not seem very common in the systems deployed today. Moreover, it suffers from at least two flaws: first it does not provide adequate defense against distributed brute force attacks, in which several thousand computers (usually themselves compromised) are used to carry out the

[8] Said differently, if the system is at significant risk after four attempts, then perhaps the problem lies elsewhere.

attack. In this case, the range of IP addresses used can be very large. Moreover, it opens up another vector for denial of service, even in an all-legitimate usage situation, because it is common for a large set of users in the same network to access the Internet via a single gateway (because they use DHCP usually) and thus share the same IP address. If enough users access the system with a shared IP address, then the system's defense mechanisms will be enacted on this IP address. (It should be noted that the reverse situation is also possible, where a user is assigned a different IP address by his/her Internet service provider at each request.)

Instead of using IP address, a more commonly deployed mechanism is to use the notion of session provided by all common Web server applications today (apache, IIS, etc.) usually via cookies and URL rewriting. This solution is simpler to implement, but is unfortunately inadequate and should never be used. An attacker can without difficulties create new session at will.

None of the defense systems described so far directly addresses trawling attacks (IP-based mechanisms do so indirectly). In fact, as far as we are aware, systems usually do not address this issue, possibly because system designers fail to understand the importance of the attack, but certainly also because it is a difficult problem to address: indeed, most systems have unique identifiers for the users, but the password is not stored as a unique system-wide value, so the database typically is indexed by user identifiers, with passwords being stored along with the user credential. In other words, the passwords are repeated if they are used by several users. In this situation, it is not easy to track attempts on the same password. Doing so in an efficient way would probably require a new table, indexed by the passwords, and a join table between the user identifiers and the passwords. This would in turn create a new set of problems when dealing with password changes and removal of unused passwords.

None of the systems addresses the problem of attacks on knowledge questions either (even the system based on IP address will probably not address it in practice since such tracking will likely occur only during the identifier and password verification step in most systems). We are not aware of any system at all trying to address the issue of trawling attacks on knowledge question. Doing so would be as difficult as doing it on passwords, and the problem may not be well understood by many systems designers.

D. *Requirements for an Effective Defense System*

Our aim is to provide a system that can adequately protect against the attacks identified above but, in order to be practical, the solution must also be easy to use. More precisely, we want to create a system with the following properties:

- Can slow down targeted, trawling, and blind attacks on user identifiers, passwords, knowledge questions, and whatever other direct means are employed to authenticate users based on something they know;
- Cannot be misused as a denial of service tool to prevent access to legitimate users (or by legitimate users accidentally denying themselves access to the system);
- Does not leak information regarding the existence or the absence of the identifier, password, knowledge question answer, etc., in the database, not even through timing attacks (where the time required to respond to a request is measurably different on average depending on whether the identifier/password exists or not);
- Gracefully handles distributed attacks, in that if the system becomes overwhelmed by the magnitude of the attack, it will at least recover automatically soon after the attack stops;
- Can be adapted to any Web application, from the largest to the smallest, providing the appropriate level of protection for each situation;
- Is fast and can be used on an existing system without noticeable penalty in the system's response time;
- Can work with passwords in any format (clear text, hashed, encrypted, etc.);
- Is very easy to use even in an existing application, does not require any change in the structure of the database or in the architecture of the application, and requires very minimal change in the application code.

III. OUR SOLUTION

In this Section, we provide a very simple, yet effective solution to the problem. We split our solution into three separate ideas which must be combined to provide the protection we want, with the characteristics outlined in Section II D:

A. *Separate the Subsystems*

The first simple idea is that it is not necessary to integrate the protection mechanism with the authentication mechanism. Because the two subsystems work with the same data (user identifiers, password, knowledge questions …), it is customary to implement the protection mechanisms within the authentication subsystem. We believe that this is not the best approach, for at least two reasons.

The first problem with coupling the two activities is the coupling itself: the actual authentication mechanism is system dependant and can be relatively complex. The protection mechanisms are also often somewhat complex and serve a different purpose. It makes good software engineering sense to cleanly separate the two activities to avoid code entanglement, and to facilitate maintenance and evolution of both sub-systems. Moreover, in our case, a coupling of both subsystems would mean that it would be more difficult to integrate our solution into existing systems.

The second problem is that coupling the two subsystems seems natural only when the provided identifier exists in the database. If not, then the authentication subsystem has nothing to do while the protection subsystem must still act. It is even more difficult when it comes to passwords or knowledge questions, since what is typically available to the authentication subsystems (repeated values associated with unique user identifiers) is not what is required by the protection subsystem.

We therefore suggest a complete decoupling of both activities. One criticism that can be made against such a decoupling is that it seems inefficient to search for the identifiers and passwords twice, once with each subsystem. We believe that such a penalty is acceptable, especially with our solution that provides a fast, in-memory lookup for the protection subsystem.

B. *Identify Independent "Directions"*

Another simple idea is to recognize the fact that attacks are done along various "directions" and that while it is necessary to protect all possible directions, it may not be useful to tackle all of the directions simultaneously.

As it stands, the following directions can be easily identified.

- Id direction: tracking the authentication request for each user identifier (existing or not in the database). This direction is important for protecting against trawling attacks and blind attacks.
- Password direction: tracking the authentication request for each password (existing or not in the database). The passwords could be tracked in plain text form, or any consistently transformed form, and the tracking does not need to be aware of the form being used. This direction is important for protecting against targeted attacks and blind attacks.
- IP direction: tracking the IP address from which authentication requests originate. This direction is important for protecting against all types of attacks, and against all identifiers (user identifiers, passwords, knowledge questions …).
- Knowledge question category direction: tracking knowledge question by "category" of expected answers, such as "family name", "first name", "pet name", "location", and so on. If a site offers a choice of knowledge questions, then the questions should be grouped by category of answers (that is, all the knowledge questions that require an answer which is a last name will be bundled up together in the same tracked direction etc.). This direction is important for protecting against trawling attacks and blind attacks against knowledge questions.
- A few important clarifications are in order: first, even though the directions are handled independently from each other, it is important to track them all. In particular, it is not enough to just track the IP direction as a catch-all for all of the attacks. It would simply fail to detect distributed attacks. Second, the IP direction must be handled with caution: as explained above, it is misleading to think that a given IP address necessarily represents a single user. It may well be the case that users reaching the application from the same network will all share the same IP address, and it may conversely be the case that a user will be assigned a different IP address with each request. Third, if various data (identifiers, passwords, answers to knowledge questions …) must be tracked, the sole purpose of assigning different directions is to be able to adjust the parameters of the protection to the data being tracked, based on the domain space and the expected rate of legitimate request. If several data items will be monitored with the same global parameters, then these items can be bundled together along the same direction. In particular, if one does not want to adapt the level of tracking by category of answers for knowledge questions, then a single general "knowledge question direction" can be used. It is even possible (although perhaps not advisable) to track all the parameters along a single direction.

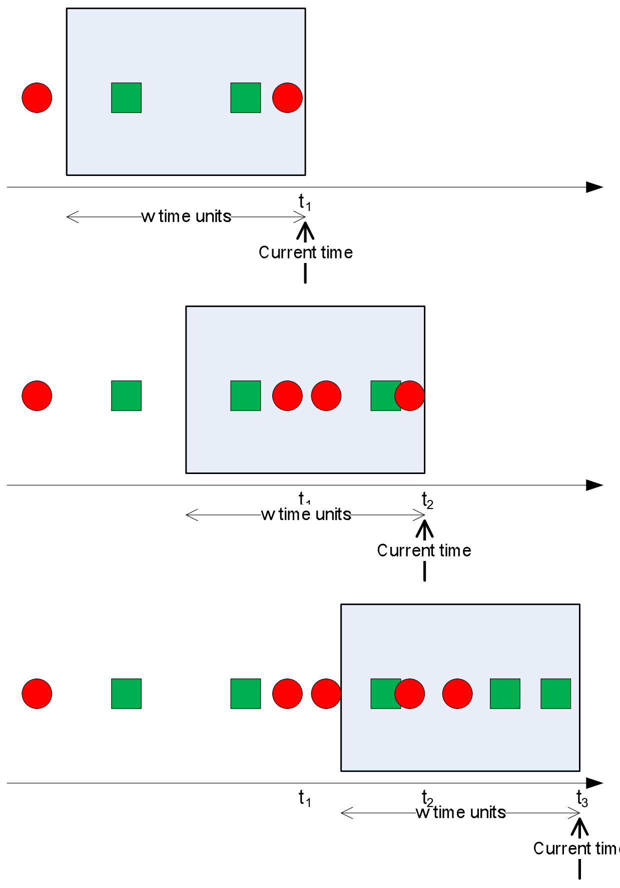

Figure 1. *A sliding window tracking of two values (represented by red circles and green squares), over* w *time units.* At time t_1, there are two hits for the green square value and one for the red circle. At time t_2, there are still two hits for the green square (although not because of the same instances), and three for the red circle. The situation reverses at time t_3. If the maximum number of hits allowed is three, then defensive measures against the red circle value will be taken when the third hit in the sliding window occurs, slightly before time t_2, while preventive action against the green square value will be taken slightly before time t_3.

C. *Sliding Window*

The last simple idea that our solution uses is a sliding window along each direction. The principle is the following: for each direction, define a size of a window (in time units) and a number of acceptable "hits" in that window, that is, a maximum number of times a particular value can appear (a "hit") within the window. All the values of the data items being tracked along this direction will be monitored for the length of the window, and if a value reaches the maximum number of hits allowed in this direction, then a defensive measure will be taken against that value.

It is important to note that the tracking is done at the level of the value of the data, not at the level of the direction. Each value is tracked independently. Figure 1 illustrates the concept with two values being tracked along the same direction.

The "defensive measures" that are taken when the maximum number of hits is reached is in fact a refusal to process requests for this value for a given period of time. A possible choice for the period could be the width of the window (so once the maximum number of hits allowed has been received for a given value over the last w units of time, then subsequent requests for this value will not be processed for w units of time). However, a different choice can be made, and this parameter can be set in our implementation.

One point of interest is whether to count the attempts for a value even when the value is being denied. In Figure 1, if the threshold is 3 and the penalty time is w, then at time t_2 the red circle value is being blocked, and it is still blocked when the last red circle value arrives later. In time, when the red circle that created the blocking exits from the sliding window, the current number of hits for red circle value can be 0 (if we just ignore non-processed requests) or 1 (if we don't). In our implementation, we have decided to ignore the attempts that are received while a value is being blocked, in effect resetting the counter when a value is unblocked. The main reason is to avoid denial of service attacks that would simply "hit" the system regularly to keep a value above the threshold.

In the next section, we will explain how to efficiently compute the number of hits within a window of size w. The solution based on sliding windows seems like a particularly good choice, since it only requires a limited amount of "history" for each value (the size of the window) and as time progresses, the older history can be safely ignored. Thus, there are no long-lasting effects with this solution; everything is always temporary and once the window is cleared the processing resumes.

Since the number of hits and the size of the window are adapted for each direction, it is clear that any brute force attack can be slowed down as required. Imagine that your system forces passwords with 20 bits of entropy for example (which is only 6 digits, so very easy to search). If the system is set to limit the requests to 60 per minute, then a targeted attack cannot exhaust the search space in less than 291 hours (12 days). If the system is set to a more reasonable threshold of 3 attempts per minute, then 242 days are necessary for an exhaustive search.

IV. An Implementation

A. *Technical Details*

We have created an implementation of the solution outlined above. Our implementation is in Java, using Servlet technologies, and should be easily deployable in any Web application using compatible technology. It currently consists of 11 Java classes totaling about 2000 lines of code.

For each direction, we maintain a list of current values, along with the current "front tile" value (see below) for each value. These lists are stored in memory using a hash table for fast access. It is important to note that because we do not store our information in a database, we can provide very fast access, but the information is volatile and will disappear every time the application is restarted. We claim that this is in fact not a problem: as explained previously, the only important data is that within the sliding window, typically a few seconds to a couple of minutes, and older information is obsolete. If the application must be restarted, then in effect the windows are reset, and the running system will very shortly be in the state it would have been in if we had saved the data and restored it at startup. Since such a restart is rare and the difference obtained by storing the lists in a database is minuscule and the gains for not storing it are very large (both in terms of execution time and in terms of setup) we have opted for an all in-memory solution.

B. *Maintaining Hit Count*

One problem to address is to efficiently maintain the number of "hits" for each value within the sliding window. A naïve approach would be to maintain a list of values, along with a timestamp for each of them. When the value is accessed again, the list is scanned, values falling outside the window are removed, the new hit is recorded in the list, and the new hit count is calculated. Such an approach would work but would be too inefficient. Instead, we approximate this number using the notion of "front tile" in the window, which gives us a single value recording how open or closed the sliding window currently is.

Specifically, if the sliding window has a width of w time units, and the threshold is n hits within that time frame, in effect each hit closes the window by w/n, so that after n hits it is completely closed. We thus associate a "tile" of width n/w to each hit, and whenever a new hit is recorded for that value, a tile is added after the last one. If that tile closes the window entirely, then the corresponding value is blocked. Calling $V_{FrontTile}$ the current position of the front tile for the value V, *CurrentTime* the time at which a hit for the value V occurs and $WindowTail = CurrentTime - w$ the current tail of the sliding window, we update $V_{FrontTile}$ as follows:

```
If $V_{FrontTile} > WindowTail$ then
    $V_{FrontTile} = V_{FrontTile} + w/n$
Else
    $V_{FrontTile} = WindowTail + w/n$
```

Once $V_{FrontTile}$ has been updated, V is blocked if and only if $V_{FrontTile} > CurrentTime$, that is, the tiling process has completely closed the window.

This technique is just an approximation (it is possible, depending on the pattern, to have up to 2n-1 hits in the window), but it is enough for our purpose, it is fast to compute, and requires only a small constant amount of memory per value.

C. Using The Tool

Using our tool is very simple. By default, it defines three directions: Id, Passwords, and IP. For each direction, one can define three values: the size of the window, the number of hits allowed within the window, and the duration of the penalty for going over that limit. All these values are stored in a configuration file, and one can remove and add directions at will by editing the file.

Once this is done, using the tool merely involves calling a method of our main object, passing as parameters the name of the direction and the value of the hit. Our function will return a Boolean value TRUE if the value must be blocked. For example, assume that a direction "KnowledgeQuestionName" has been defined in the configuration file, and that the name "Smith" has been received by the application. The following call must be added, where "doSomething();" is what the application does when a value is blocked (usually simply refuse to process the request) and answeredName is a variable that stores the answer provided by the user ("Smith" in this case):

```
if(AttackCheck.isBlocked(
        "KnowledgeQuestionName", answeredName){
                doSomething(); return();

        }
```

For convenience, a single method to check Id, Password and IP direction at once is provided as well. A typical usage in a Web application is as follows, where the user identifier is obtained from the parameter "ID" and the password from the parameter "PWD" ("request.getRemoteAddr()" returns the IP address of the client)

```
if(AttackCheck.isBlocked(request.getParameter("ID"),
                         request.getParameter("PWD"),
                         request.getRemoteAddr()){
                doSomething(); return();

}
```

As can be seen, the impact of the tool on the existing code is very minimal, basically a single method invocation (and subsequent preventive action if necessary). It is of course important to invoke the method each time a new value is provided, for all the relevant directions. So a typical web application, with a backup knowledge question system, will usually add two method invocations, one for the primary login, and one for handling the knowledge question. Everything else remains unmodified.

In terms of efficiency, we ran tests on a Linux server with an Intel Dual core processor at 3 GHz with 3 Gigabytes of RAM, running apache TomCat. The server was "attacked" by a similar machine executing 10 threads, each attempting 10,000 "login" operations, choosing uniformly randomly from a domain space of size 30,000 for both Ids and passwords. Checking all three default directions (Id, password, and IP) took an average of 7 milliseconds during the attack when the lists are stored in a hash table. This time includes the "cleanup" done in the background to remove older values from the list.

V. Discussion

A. Strengths of the System

Our system does provide an adequate solution to the goals outlined in Section II D. It can slow down all the outlined attacks as much as deemed necessary by modifying the parameters, and can do it in any number of directions required. It does not leak any information regarding the validity of the processed values in the database, since it does not know the information to begin with[9]. Thanks to the sliding window approach, it has no long lasting effect; any denial is temporary. Thus, users cannot lock themselves out by mistake, and in fact will not notice that the system is in place as long as the parameters used are not too stringent. A targeted denial of service attack will need to be distributed (since otherwise the attacker's IP will soon be blocked if the system is properly configured), and its effect will disappear very soon after the attack stops. The system is fast (checking all three default directions under stress takes on average 7 milliseconds in our tests), and requires minimal code modification and no architecture or database changes. Alternatively, it is possible to modify the flow of login requests to go through our system first, then to the original login, in which case no modification of code at all is required.

B. Weaknesses of the System

Although the proposed system has many advantages, it also has some weaknesses and shortcomings.

- If the database has a few known values that are extremely frequent, then it will be difficult to adequately protect such a system, since the window will have to be sufficiently small that it will inconvenience users. If, for example, the password "123456" is used by close to 1% of the user base, as seems to be the case in some of the reported database leaks (see Section 2 B.), and the user identifiers are known, then it seems very difficult in practice to protect such a database without seriously inconveniencing the other users.
- One side effect of the Password direction is that a distributed denial of service attack is now possible on frequent passwords. In the case above, it means that a group of attackers can deny access to users that have "123456" as a password. On the other hand, this will stop when the attack stops, and the accounts of these users would otherwise be at high risk of being compromised.
- It is very important to have the tightest parameters on the IP direction, so that a single attacker cannot create denial of service attacks. This is possibly problematic for a large number of users coming to the system with the same IP address. Also, the IP address direction is truly effective only if the system's architecture guaranties at least one complete message exchange between the client and the server prior to the IP

[9] Of course, care should be taken so that the web application itself does not leak this information during the subsequent login process.

verification (otherwise, an attacker can just spoof login messages with a random IP addresses).

- Finding the right values for the parameters of a given application might be difficult (See Section VI).
- It is theoretically possible to flood the system itself to have it store very large lists and in turn impact the server using it. Given the small amount of information stored for each value, this would seem to require a massive distributed attack. Nevertheless, our system is equipped with a garbage collector that discards expired values in the background, and it can be parameterized to not consume more than a set amount of memory (if the amount is reached, then every request is denied until garbage collection occurs).

Despite these shortcomings, we believe that this system is a clear, practical step forward in securing Web applications and we encourage everyone to install it or include similar measures in their applications.

VI. Conclusion and Future Work

We have presented a system, based on the idea of a sliding window, which can be used to slow down targeted brute force attacks (where a single account is being attacked), trawling brute force attacks (where accounts are being attacked based on common passwords), and blind attacks (where both user identifiers and passwords are being searched). Our system can also prevent trawling attacks on knowledge questions. By providing several "directions", we allow our users to adjust the level of allowed requests per time unit for different types of information. By decoupling the protection subsystem from the authentication subsystem, we provide a solution that is non-intrusive and can be easily incorporated into existing applications without significant time penalty and with almost no code modification and no database changes at all.

Finding the parameters for a specific application might be difficult. For the time being, the default values are four hits per minute for id and password, and four hits every 55 seconds for IP address, but these values may not be appropriate for all sites; finding reasonable defaults for different environment types is an area of further research. Another possible extension of this work is to provide a "reporting" mode that does not actually block any requests but records the traffic as experienced on the site, and then shows what effect particular parameter settings would have had on the site over the recorded period of time. Another direction would be to come up with a more accurate but as efficient formula to compute the number of hits in the window. It would also be interesting to investigate the possibility of identifying users in a better way, beyond simple IP address. Porting the tool to other environments would also be worthwhile and increase its potential impact.

Our tool is available to use now and will soon be made more publically available; in the meantime, it can be requested by email. We are currently in discussion with some companies that are considering including it as part of their product and we are looking for more such opportunities.

References

[1] M. Bishop and D. Klein, "Improving System Security Through Proactive Password Checking," *Computers and Security* **14**(3), 1995, pp. 233–249.

[2] R. Morris and K. Thompson, "Password security: a case history." *Communications of the. ACM* **22**(11),1979, pp 594 – 597.

[3] D.Florêncio, C. Herley, and B. Coskun. "Do strong web passwords accomplish anything?". In *Proceedings of the 2nd USENIX workshop on Hot topics in security* (HOTSEC'07), 2007, pp 1 – 6.

[4] J. Bonneau, M. Just and G. Matthews, "What's in a Name? Evaluating Statistical Attacks on Personal Knowledge Questions," In *Proceedings of the 14th international conference on Financial Cryptography and Data Security* (FC'10), January 2010.

[5] L. v. Ahn, B. Maurer, C. McMillen, D. Abraham and Manuel Blum, "reCAPTCHA: Human-Based Character Recognition via Web Security Measures", *Science*, **321**, 2008. pp. 1465 – 1468.