Chapter 1
Protecting Financial Institutions from Brute-Force Attacks

Cormac Herley and Dinei Florêncio

Abstract *We examine the problem of protecting online banking accounts from password brute-forcing attacks. Our method is to create a large number of honeypot userID-password pairs. Presentation of any of these honeypot credentials causes the attacker to be logged into a honeypot account with fictitious attributes. For the attacker to tell the difference between a honeypot and a real account he must attempt to transfer money out. We show that is simple to ensure that a brute-force attacker will encounter hundreds or even thousands of honeypot accounts for every real break-in. His activity in the honeypots provides the data by which the bank learns the attackers attempts to tell real from honeypot accounts, and his cash out strategy.*

1.1 Introduction and Related Work

The majority of banking and financial institutions in the US authenticate users with a simple userID-password pair. The main encouragement for brute-force attackers is the notorious weakness of user-chosen passwords, first observed three decades ago by Morris and Thompson [7]. A more recent study of web password habits by Florêncio and Herley [1] showed that weak passwords are still very common. Much of the work on password attacks has focussed on off-line attacks. Instead of focussing exclusively on keeping the attacker out we propose to let him in and, as Provos and Holz put it [6], "look at the bright side of break-ins." Break-ins are a problem because it can be hard to tell the fraudulent activity in an account resulting from a break-in from the legitimate activity of the account owner. By allowing the attacker into many honeypots for every one real account we will learn detailed information on his strategy both for cash-out and to tell honeypots from real accounts. In addition we slow him down more effectively than a lockout rule, without the Denial of Service hole that lockouts create. Pinkas and Sander [5] examine a question close to our problem: their motivation is to prevent brute-force attacks without resorting to a lockout policy. The lockout policy opens a Denial of Service vulnerability, and, as [5] points out, this can be a very serious issue for some classes of accounts: *e.g.* the possibility of eliminating a rival

Microsoft Research
One Microsoft Way
Redmond, WA

from an online auction by locking their account would be unacceptable to eBay for example. Pinkas and Sander's raise the attacker's cost by requiring that a Human Interactive Proof (HIP) be solved after a threshold number of attempts has been exceeded. Van Oorschoot and Stubblebine [4] enhance the scheme by better use of the recent account login history; they are able to improve the protection and simultaneously reduce the number of HIP's that legitimate users will be asked to solve.

1.2 Attacks

Brute-force and guessing attacks In a brute-force attack repeated credential pairs are tried in an attempt to gain access to an account. The simplest is directed against a single account: the attacker tries all possible passwords for one userID until one succeeds. For non-numeric passwords he might increase his yield by trying passwords in order of likelihood. Simple brute-force attacks like this are generally stopped by a "three strikes" type lockout policy (but such policies open up a Denial of Service attack as below). Far more likely to succeed is a bulk guessing attack against a large number of accounts [5, 3]. Instead of trying different passwords for a single userID the attacker tries different userID-password pairs. Since only a small number of unsuccessful logins are attempted at any individual userID the attack is far harder to detect. Observe that a bulk guessing attacker knows very little about his victims. For example an attacker who forces entry will generally not know the name, address or any other information about the victim until he logs in.
Cash-out strategies: Once an attacker forces entry, the server has few means to distinguish him from the legitimate user. It might seem that all is now lost, but in fact the attacker's task is just beginning. The assets in the compromised account is all potential gain, but to turn the potential into reality he must move the assets to a "safe place," by which we mean cash or an account the attacker controls that is beyond the reach of the bank or law enforcement and cannot be frozen. It is important that any transfers he performs cannot be reversed when the break-in is detected; it is also important that none of the intermediate accounts can be used to identify the attacker. For example, wiring money from a compromised account at BigBank to an account at AnotherBigBank will naturally draw scrutiny to the holder of the second account. This task is far from trivial. Thomas and Martin [8] describe a complex ecosystem that has developed around harvesting stolen credentials. Cashiers and drop men are used to pick up money moved from the compromised accounts.
Do we care about brute-force anymore? The existing approach to brute-force attacks is a combination of:

- password strength policies,
- "three strike" type lockout rules and

- fraud and anomaly detection at the backend server.

Passwords strength policies are unpopular and users demonstrate considerable preference for short passwords. Three strikes type lockout rules suffice to slow down attacks on a single account, but do little against the bulk guessing attacker. Further, in some cases any lockout on an account can be very undesirable. Three well-timed failed logins can deny access a rival in an auction, for example [5]. Equally, an attacker who gains access to the list of userID's at a bank can halt all online access at a cost three times as many form submits as their are accounts. And this attack can be repeated. Details on the backend fraud detection employed by banks is understandably not made public.

The prevalence of brute-force attacks is itself hard to estimate. One might argue that in the age of better attacks such as phishing and keylogging brute-force is no longer an issue. However, banks are unable to lower the defences against it. And these defences result in burdensome password policies for users, and lockout policies, which generate the Denial of Service (DoS) vulnerability. It is argued by Florêncio *et al.* [2] that bulk guessing brute-force attacks are the main reason for strong password policies on online accounts. The approach we introduce in the next section makes brute forcing a great deal harder. Thus, we claim it carries two great advantages. First, it removes the need to encumber users with strong password policies. Second, it removes the need for a lockout rule and thereby eliminates a lockout based DoS threat.

1.3 Method

A Simple Honeypot account: A honeypot account at a financial institution is an account that appears exactly like a real account, except of course there is no real money there. In every other respect it is indistinguishable from a real account. It has all the attributes that a real account would have: name, address, email account of record, beneficiary information, account history, balance, holdings *etc.* When logging in to any account a user generally expects to be able to:

- Change account information (*e.g.* name, address, beneficiaries *etc*)
- Buy or sell instruments (*e.g.* move money from checking to savings, buy or sell stock *etc*).
- Send money to previously used accounts (*e.g.* utilities, mortgage)
- Send money to a new recipient

An attacker who enters a honeypot account will have access to the full range of services, with the exception of course that the bank will not actually remit any money to anyone. It will however pretend that it has done so. Only attackers will enter a honeypot, and the goal of the account is to create the illusion of reality. Thus the bank will do everything possible to perpetuate the illusion except part with money.

Generation of Honeypot accounts: To protect against brute-force attackers a bank may need thousands, or even millions of honeypot accounts. Hence it is important to be be able to generate such accounts at will. This is actually simple. Our solution is to copy attributes from a the pool of real accounts, but enter fictional attributes for name, address, beneficiaries *etc.* This guarantees that the honeypot contains valid account history and transaction details. For example an attacker would see real online bill pay details, together with the amounts and dates, but for a fictionally generated user. Accounts are generated on-demand at the time of first entry, but account information persists through successive logins.

Distribution of Honeypot accounts: Consider an institution BigBank which assigns users a b_u bit userID and enforces that passwords are at least b_p bits. Together the userID-password pair form the credential that grants access to a user account. In general entry of a wrong userID or wrong password or both results in a "login failed" message to the user.

In our solution the bank in addition to the credentials of real users allows access to a honeypot account when any of a number of honeypot credentials is presented. The size of the combined userID-password search space, at $2^{b_u+b_p}$, is far larger than the number of real user accounts the number of honeypot accounts can outnumber real accounts. For example, for a given userID if a single 8-digit PIN results in login there is one correct password, and almost one million incorrect possibilities. Instead of having each of these incorrect passwords results in a "login failed" page we have the bank assign 10000 of the incorrect passwords to honeypot accounts. Thus an attacker who mounts a brute-force attack on the password of that userID is 10000 times more likely to gain entry to a honeypot account than a real account. With an 8-digit PIN we can still ensure that a legitimate user who types two digits of his PIN incorrectly (there are $\binom{8}{2} \cdot 10^2 = 2800$ such possibilities) will still never enter a honeypot.

Honeypots associated with a userID assigned at setup: To ensure that a user who types the userID correctly, and gets no more two characters of the password incorrect, never enters a honeypot we must ensure that honeypot passwords, for that userID, are a sufficient distance from the true password. Since the salted hash is all that is required for authentication, passwords are generally not stored on the server. Thus the only time the server can determine which passwords are close to the true password for a particular userID is when the account is being set up. At this time, when it salts and hashes the password, it should also generate as many honeypot passwords as required and also salt and hash them. In this way there is no change in the existing arrangements for storing credentials. The details associated with a honeypot need not be generated only when an attacker enters the account.

Cashing-out must be done on real accounts only: If an attacker forces entry he must first determine if he is in a real or a honeypot account and then cash-out. He would be very foolish to attempt cash-out on all accounts: since stepping stone accounts and the services of cashiers are expensive [8], he

cannot risk a cash-out attempt on a honeypot. Should he do so he identifies his cashier, and the bank will suspect any attempt to transfer to the cashier from a real account.

Telling them Apart Without Getting Caught: Suppose the attacker has compromised N accounts, of which $M \ll N$ are real and the remainder honeypot. The attacker and the bank each have partial and different information about the break-ins. The attacker knows all N of the accounts he has broken, but does not know which M are real. The bank knows only that the $N-M$ logins to the honeypot accounts must be the work of an attacker, but does not know which M real accounts are also compromised.

To distinguish the real from the honeypots, but with the constraint that his activity must vary from account to account. Suppose the attacker sends funds for a small purchase (from an innocent retailer) from each of the N accounts and arranges to have mail sent to a hotmail account when the item ships. This can be done, but is cumbersome. The attacker must choose a retailer who will accept a transfer or check and then arrange for funds to be sent from the compromised account. This would enable him to identify the M real accounts that are active. But it also notifies the bank that these M real accounts have something in common with the $N-M$ honeypot accounts. Clearly a one-size-fits all strategy will not work to distinguish accounts if he is to remain undiscovered. To avoid linking a real account, when he finds it, to the known-bad work of an attacker he must employ N different retailers to probe his collection of accounts. While he may be able to write a script that programmatically arranges a transfer from each of the N accounts, it is not possible to set up purchases at N different retailers this way. This is expensive: the attacker must expend individual effort for a retailer on each of the N accounts. Thus the attacker's effort increases with N, while the bank can generate honeypot accounts at will. For large enough N the attacker is faced with great effort for minimal return.

References

1. D. Florêncio and C. Herley. A Large-Scale Study of Web Password Habits. *WWW 2007, Ban®.*
2. D. Florêncio, C. Herley, and B. Coskun. Do Strong Web Passwords Accomplish Anything? *Proc. Usenix Hot Topics in Security*, 2007.
3. K. J. Hole and V. Moen and T. Tjostheim. Case Study: Online banking Security. *IEEE Security & Privacy Magazine*, 2006.
4. P.C. van Oorschot, S. Stubblebine. On Countering Online Dictionary Attacks with Login Histories and Humans-in-the-Loop. *ACM TISSEC vol.9 issue 3*, 2006.
5. B. Pinkas and T. Sander. Securing Passwords Against Dictionary Attacks. *ACM CCS*, 2002.
6. N. Provos and T. Holz. *Virtual Honeypots*. Addison Wesley, 2007.
7. R. Morris and K. Thompson. Password Security: A Case History. *Comm. ACM*, 1979.
8. R. Thomas and J. Martin. The Underground Economy: Priceless. *Usenix ;login:*, 2006.