A Bodyguard of Lies: The Use of Honey Objects in Information Security

Ari Juels
Cornell Tech
New York, NY, USA
juels@cornell.edu

ABSTRACT

Decoy objects, often labeled in computer security with the term *honey*, are a powerful tool for compromise detection and mitigation. There has been little exploration of overarching theories or set of principles or properties, however. This short paper (and accompanying keynote talk) briefly explore two properties of honey systems, *indistinguishability* and *secrecy*. The aim is to illuminate a broad design space that might encompass a wide array of areas in information security, including access control, the main topic of this symposium.

1. INTRODUCTION

From time immemorial, deception been a cornerstone of counterintelligence. In particular, the use of decoys, realistic but fake objects to divert or detect attacks, has proven a powerful technique. Like most clever ideas, decoys arose in the natural world before human beings stumbled upon them. For example, many insects and fish have false eyes known as "eye-spots." These prominent dark circles lure attackers away from more vulnerable body parts. Also like most clever ideas, decoys are the subject of a quip by Winston Churchill. He famously remarked, "In war-time, truth is so precious that she should always be attended by a bodyguard of lies."

In computer security, the term *honey* is often favored to denote decoys [2]. Honeypots, servers deployed to lure attackers for observation, are the best known example [6]. But there are many varieties of honey system, such as honeyfiles [10], honey documents [1], honeytokens [7], and honey encryption [3].

While honey objects are fairly widely used, there is little literature articulating overarching concepts in their use or design. This paper will briefly discuss two properties required for successful deployment of honey objects. While these properties are simple, intentional reflection on them can lead to better architectures and a richer design space for honey systems.

As a point of reference, let us consider a simple system in which $S = \{s_1, \ldots, s_n\}$ denotes a set of n objects of which one, $s^* = s_j$, for $j \in \{1, \ldots, n\}$ is the *true* object, while the other $n-1$ are *honey objects*. It's convenient to refer to the objects in S generically as *sweet* objects.

SACMAT'14, June 25–27, 2014, London, Ontario, Canada.
ACM 978-1-4503-2939-2/14/06.
http://dx.doi.org/10.1145/2613087.2613088 .

The two principles we'll explore then are:

1. *Indistinguishability:* To deceive and attacker, honey objects must be hard to distinguish from real objects. They should, in other words, be drawn from a probability distribution over possible objects similar to that from which a real object s^* was selected.

2. *Secrecy:* In a system with honey objects, j is a secret. Honey objects can, of course, only deceive an attacker that doesn't know j, so j cannot reside alongside S. Kerckhoffs's principle therefore comes into play: the security of the system must reside in the secret, i.e., the distinction between honey objects and real ones, not in the mere fact of using honey objects. The secret j may reside with either or both of two types of entity: An *actor* (generally a user) that selects objects from S and a *honeychecker*, a system that signals an attack when an object other than $s^* = s_j$ is selected and is distinct from the system that stores S.

This paper will illustrate these two concepts as they arise in a system called *honeywords* and how they influence the larger design space for honey systems. Then it will very briefly discuss how honey objects might be deployed in access-control systems.

2. AN EXAMPLE: HONEYWORDS

Honeywords are decoy passwords used to detect the breach of a password database [4].

A password database is an authentication system component that contains a set of passwords, one per account.[1] An attacker that breaches the system and obtains the database can impersonate a user by authenticating to the system using her password. A spate of recent such breaches has shown how pervasive this threat is.

In a honeywords system, for a given user, a *list* of n passwords $S = \{s_1, s_2, \ldots s_n\}$ is stored. In this list, $s^* = s_j$ is the user's real password, while s_i for $i \neq j$ is a honeyword, a fake password. The term "sweetwords" is applied to elements of S in general.

When a user tries authenticating to the system using a password s, it is checked against the list S. If it doesn't lie in the list, then the authentication attempt is rejected as in a normal password system. If $s = s_j$, then authentication proceeds normally.

If, however, s is a honeyword, meaning $s = s_i$ for some $i \neq j$, an alarm is triggered in the system. It is improbable that a user will accidentally submit a honeyword rather than her correct password when authenticating. So submission of a honeyword is good evidence (although not necessary proof positive) of a breach that disclosed S to an attacker.

[1]The passwords should be hashed and salted, but exactly how they are stored and checked isn't important for our purposes here.

The two principles of indistinguishability and secrecy are addressed in this system as follows:

1. *Indistinguishability:* Suppose that we model a true password $s^* = s_j$ as coming from a probability distribution U, and honeywords as selected from a distribution G. Suppose an attacker breaches the system and learns S, and that it knows U and G. It can be shown that the attacker that tries to impersonate a user by proffering a sweetword s_i as the user's password achieves the highest probability of success by guessing the sweetword $s_i \in S$ that maximizes $U(s_i)/G(s_i)$.

   Thus, indistinguishability in this setting means that the distributions U and G are close. There are a number of strategies for achieving this property and generating good honeywords. For example, in a large system, a given user might be assigned the passwords of other users as honeywords.

2. *Secrecy:* If j is stored alongside S, then an attacker that breaches the system can immediately identify $s^* = s_j$. The architecture of the honeywords system proposed in [4] is therefore distributed. A honeychecker is deployed as a system distinct from the computer system that stores S. This honeychecker stores the index j for every user. The computer system transmits to the honeychecker the index of any submitted sweetword, which the honeychecker verifies. Given its minimal functionality—it can be designed as a input-only device—the honeychecker may be harder for an attacker to breach than the computer system.

   The actor in this case is the user. The user stores $s^* = s_j$ and thus j in her head.

3. THE DESIGN SPACE

There are many other ways in which indistinguishability and secrecy can be enforced. A few other examples deserve mention to illustrate the breadth of the design space for honey systems.

3.1 Indistinguishability

There are some cases in which the true object s^* comes from a mathematically well defined distribution, e.g., credit card numbers, RSA private keys, and so forth. A number of such examples are given in [3]. In these cases, generation of good honey objects may be relatively straightforward. Alternatively, s^* might come from a distribution that can be well characterized using public sources of data, such as user-selected passwords. Generation of valid honey objects may nonetheless remain somewhat challenging [4].

When the true object s^* comes from a complicated distribution, constructing good honey objects may seem nearly impossible. Honey documents—e-mail messages, for example—could in principle require the generation of fake but semantically and contextually realistic natural language, an intractable problem today.

On the other hand, to be effective, honey objects need not necessarily bear up under intensive manual scrutiny by an attacker. If manual analysis of honey objects is costly, then it is sufficient for samples from the probability distributions U and G to be difficult to distinguish by means of machine classification. In [5], a project history is simulated for a piece of bogus software is produced by introducing obfuscated software variations. Similarly, in [9], the creation of documents in foreign languages is proposed to deceive attackers without the ability to apply translation tools on the fly.

A good design principle is to minimize the complexity of U by creating a honey system that presents a highly constrained distinguishability problem. For example, a decoy document system is proposed in [8] whose efficacy relies on attackers searching documents differently than legitimate users. The documents themselves need not look realistic, though, only their names and directory placement.

Measuring indistinguishability can be a serious challenge when U is complex—particularly when an attack may be manual. An additional challenge, when honey objects are visible to users, is ensuring that valid users can themselves distinguish effortlessly between true and honey objects, or otherwise do not trigger false positives by frequently touching honey objects. Legitimate users should preferably encounter honey objects only rarely.

3.2 Secrecy

A honeychecker can in principle be a simple, dedicated service, as in the honeywords system, and the secret j the index of a valid object. But a honeychecker can in fact assume a wide variety of forms, as can the secret.

In the Decoy Document Distributor (D^3) system [1], several types of honey and thus honeychecker are deployed. Documents contain watermarks in their associated binaries that can be detected upon document exfiltration from a network, code that beacons to a server upon opening of a document, and bogus credentials whose use signals document exfiltration to a relying party.

In [8], inside attackers are detected by merit of the fact that they click on documents that a well behaving user would not access. Thus there are cases in which the secret in a honey system is not an index j, but rather a behavioral profile that might be stored by a honeychecking service and reside in the mind of an actor, i.e., a user, as a pattern of behavior.

Honey encryption [3] operates in a rather different setting. It permits the construction of a ciphertext that decrypts under any key to a plausible looking plaintext. An attacker therefore cannot tell when the ciphertext has been correctly decrypted. For example, given a password vault encrypted under a master password using honey encryption, an attacker that tries to guess master passwords cannot determine when decryption has been successful. The attacker therefore does not know when passwords extracted from the vault are safe for use in impersonating the owner of the vault. There is no explicit honeychecker associated with a honey encryption system.

A service that consumes and verifies the correctness of the plaintext in a honey encryption system, however, may act in effect as a honeychecker. For example, if a password vault is encrypted using honey encryption, then any server that consumes a password in the vault effectively acts as a honeychecker.

4. HONEY IN ACCESS CONTROL

Honey objects can be used in an access-control system against a range of possible adversaries: rogue administrators, misbehaving users or other subjects, outside attackers, and so forth—as well as for audit. Similarly, access-control systems can conveniently support general honey systems. Here we give a couple of examples.

Consider the case of a user seeking to access resources illegitimately. One might imagine a role-based access control (RBAC) system in which roles include "honey permissions," that is, permissions that exceed the organic responsibilities associated with the role (e.g., access to financial spreadsheets for an IT administrator). Indistinguishability in this case means the inability of an attacker to determine whether a permission was assigned legitimately to a role or as bait. The secret is the set of honey permissions; a honeychecker might monitor for use of these permissions. In a real-world system, accommodation would need to be made for (inevitable) false positives, and a policy decision would have to be made as to

whether resource-access granted by honey permissions would be allowed by the system or blocked upon alert by the honeychecker.

Access-control systems can be a convenient point at which to insert honeychecking capabilities for a honey system. In particular, a reference monitor may be used to store secrets for honey systems—watchlists of honey objects. Access-control systems can also support real-time response to suspected breaches. If permission is requested to touch a honey object, the reference monitor might alert an administrator, isolate critical systems by revoking permissions, etc.

There are no doubt many other opportunities to combine honey objects, with their rich design space, and access-control systems, with their widespread use in real-world system.

5. REFERENCES

[1] B.M. Bowen, S. Hershkop, A. D. Keromytis, and S. J. Stolfo. Baiting inside attackers using decoy documents. In *Security and Privacy in Communication Networks*, pages 51–70, 2009.

[2] F. Cohen. The use of deception techniques: Honeypots and decoys. In H. Bidgoli, editor, *Handbook of Information Security*, volume 3, pages 646–655. Wiley and Sons, 2006.

[3] A. Juels and T. Ristenpart. Honey encryption: Beyond the brute-force barrier. In *Eurocrypt*, 2014. To appear.

[4] A. Juels and R. Rivest. Honeywords: Making password-cracking detectable. In *ACM CCS*, pages 145–160, 2013.

[5] Y. Park and S. J. Stolfo. Software decoys for insider threat. In *ASIACCS*, pages 93–94, New York, NY, USA, 2012.

[6] L. Spitzner. *Honeypots: Tracking Hackers*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[7] L. Spitzner. Honeytokens: The other honeypot. Symantec SecurityFocus, July 2003.

[8] M. B. Salem S. J. Stolfo. Modeling user search behavior for masquerade detection. In *RAID*, pages 181–200, 2011.

[9] J. Voris, N. Boggs, and S.J. Stolfo. Lost in translation: Improving decoy documents via automated translation. In *IEEE Symposium on Security and Privacy Workshops (SPW)*, pages 129–133, 2012.

[10] J. Yuill, M. Zappe, D. Denning, and F. Feer. Honeyfiles: deceptive files for intrusion detection. In *IAW*, pages 116–122, 2004.