DecoyAuth: Protocol Design

Mathy Vanhoef

Abstract

This whitepaper presents the DecoyAuth protocol, a zero-knowledge authentication scheme that supports the use of decoy tokens. These tokens act as a reverse honeypot: if a stolen or leaked decoy is used, it signals a potential breach, in particular, that the server's password database has been compromised. This work was supported by NGI Sargasso.

1 Introduction

Zero-knowledge authentication protocols offer major security advantages. In particular, when using passwords or more general-purpose authentication keys, the counter-party only learns whether the correct key was used, without learning the value of the used key. Unfortunately, existing deployed zero-knowledge authentication protocols do not support decoy tokens or passwords, limiting their ability to detect credential compromise. To address this, we propose the DecoyAuth protocol: it adds support for decoy tokens to zero-knowledge authentication protocols. The decoy tokens function as a reverse honeypot: a decoy key is indistinguishable from a real key, and when an adversary steals or uses a decoy key, the counter-party can detect this and take appropriate security measures.

Concretely, this whitepapers describes the design of DecoyAuth. This protocol is inspired by a combination of O-PAKE and Dragonfly [9, 7]. It supports an arbitrary number of decoy passwords, where both the computation and data overhead scales linearly in function of the number of used decoy passwords. We also describe how to use DecoyAuth in a two-server architecture to detect credential compromise: the authentication server verifies logins without knowing which password is real, and a separate DecoyChecker component identifies decoy usage and raises alerts.

More broadly, we hope our work will serve as the basis for future research on extending zero-knowledge authentication. In particular, we believe it could inspire the integration of multi-password support into protocols like WPA3.

2 Background

This section introduces elliptic curves, the notation we use, and the Dragonfly protocol that we will modify.

2.1 Elliptic Curves

Elliptic curve cryptography (ECC) is a form of public key cryptography based on elliptic curves over finite fields. We work over a finite field $\mathbb{F}_p$, where p is a prime. This means that all arithmetic operations—such as addition, subtraction, multiplication, and division—are performed as usual, except that results are taken modulo p. An elliptic curve E over such a finite field $\mathbb{F}_p$ is defined by the equation:

$$E : y^2 = x^3 + ax + b \mod p$$

where $a, b \in \mathbb{F}_p$. The set of points (x, y) satisfying this equation, along with a special point at infinity $\mathcal{O}$, are called elliptic curve points and these points from the basis of various cryptographic operations. Several group operations are frequently used:

- **Point addition:** Given two distinct points P and Q, the point $P + Q$ is derived using a geometric rule defined over the curve. For more details on this operation, see [6].
- **Scalar multiplication:** Given a point P and scalar $k \in \mathbb{Z}$, the operation $k \cdot P$ denotes the repeated addition of P to itself k times.
- **Point negation:** For a point $P = (x, y)$, the negation is $-P = (x, -y \mod p)$.

2.2 Notation and Number-Theoretic Concepts

In this whitepaper we use several essential notations and concepts to keep descriptions concise. The most essential ones are the following:

Random Sampling Notation. The notation $x \leftarrow_\$ S$ means that the variable x is sampled uniformly at random from the set S. For example, writing $u \leftarrow_\$ \mathbb{F}_p$ denotes selecting a random element u from the finite field $\mathbb{F}_p$.

Quadratic Residues. Given a prime p, an element $a \in \mathbb{F}_p$ is called a *quadratic residue* if there exists an $x \in \mathbb{F}_p$ such that:

$$x^2 \equiv a \mod p.$$

In other words, a is a square modulo p. Not all elements in $\mathbb{F}_p$ are quadratic residues; for example, half of the nonzero elements in $\mathbb{F}_p$ are non-residues. Determining whether a given a is a quadratic residue can be done using Euler's criterion:

$$a^{(p-1)/2} \mod p = \begin{cases} 1 & \text{if } a \text{ is a quadratic residue,} \\ -1 & \text{if } a \text{ is a non-residue.} \end{cases}$$

Modular Inverses. The modular inverse of $a \in \mathbb{F}_p^*$ (nonzero elements of the field) is an element a^{-1} such that:

$$a \cdot a^{-1} \equiv 1 \mod p.$$

The inverse a^{-1} exists for all $a \neq 0$ in $\mathbb{F}_p$, and can be computed efficiently using the *Extended Euclidean Algorithm*. In code or pseudocode, this is often invoked when computing divisions modulo p. For example, computing $\lambda = \frac{a}{b} \mod p$ requires calculating $b^{-1} \mod p$ and then multiplying it by $a \mod p$.

Passwords and Hashes. We use π to represent a (decoy) password. The notation $H(\pi)$ represent a hash-to-curve method such as those in RFC 9380 [4]. In other words, the output of $H(\pi)$ is a point on an elliptic curve. In contrast, the notation $h(\pi)$ represents a traditional hashing method applied to the password such as SHA256.

2.3 Dragonfly

The Dragonfly handshake is a Password-Authenticated Key Exchange (PAKE) that establishes a shared secret between two parties over an insecure channel, using only a low-entropy shared password. It offers mutual authentication, resistance against offline dictionary attacks, and provides forward secrecy Dragonfly is used in real-world systems such as WPA3's authentication protocol, where it provides robust security even when users choose weak passwords. Its design supports both elliptic curves and MODP (modular exponentiation) groups. However, elliptic curve variants are more commonly used in practice, as they are required to be supported by WPA3.

Dragonfly consists of two main phases: the commit phase and the confirm phase. In the commit phase, each party generates a random scalar s and computes a corresponding elliptic curve element E. This point is calculated by multiplying the scalar with a point derived from the shared password. These values are exchanged between the parties. Using the received element and their own scalar, each party derives a shared secret key K. In the confirm phase, each party proves possession of this key by sending a confirmation value c, typically a hash-based message authentication code (HMAC) over the transcript of exchanged values. This two-phase structure ensures that authentication is tied to the password, while also confirming key agreement without revealing any useful information to potential eavesdroppers.

3 The DecoyAuth Protocol

This section presents the high-level design of the DecoyAuth protocol and outlines the cryptographic primitives and algorithms it builds upon.

3.1 High-Level Design

The design of DecoyAuth is illustrated in Figure 1 on page 5. The core idea is that the server will loop through all passwords π_i—of which one is the real password and the others are decoys—and generate an elliptic curve point $E_{B,i}$ for each (decoy) password. Similar to Kiefer and Manulis' O-PAKE protocol, these elliptic curve points are then encoded using a password hiding encoding algorithm [9]. The resulting encoding of these points, represented using the term $vals$, is then transmitted to the client.

The client can use its copy of the password to decode the matching elliptic curve point from the received $vals$ encoding. If the client's password is correct, meaning it is either the real password or a decoy password, this will result in the matching elliptic curve point $E_{B,i}$. If the client's password is incorrect, this will result in a random but valid elliptic curve point, after which the handshake will then not complete. In any case, the client uses the decoded E_B value to calculate the shared key K , and calculates a confirm value c_A. This confirm value is then sent to the server.

Upon receipt of the client's confirm value c_A, the server enters a for loop where it iterates over all passwords, i.e., over the real and all decoy passwords. For every password π_i, the corresponding shared key K and confirm value c'_A is calculated. If the computed c'_A matches the received confirm value c_A then the used (decoy) password was found. If there is no such match, authentication fails. After this, the server computes its own confirm value c_B, and sends it to the client.

Finally, the client verifies the received confirm value c_B by locally recomputing it, and seeing whether it matches the received value. If these values match, then authentication successfully completes.

An important remark is that the server should loop through all passwords in a random fashion upon receipt of the client's confirm value c_A. Otherwise, timing attacks may be possible, and a man-in-the-middle adversary would be able to determine the index of the used password, which can pose a privacy risk.

3.2 Secure Element Encoding

To ensure that an adversary cannot abuse the transmitted encoding *vals*, it is essential that the operation $decode(vals, h(\pi))$ outputs a valid elliptic curve point for every possible (decoy) password π. To guarantee this, we use the elliptic curve point encoding proposed by Tibouchi [11]. Tibouchi's encoding converts elliptic curve points into two numbers u and v, and supports a large set of different elliptic curves. More importantly, any combination of values for u and v encodes a valid elliptic curve point. As a result, any output of $decode(vals, h(\pi))$ would then represent a valid elliptic curve point.

When based on the Simplified Shallue–van de Woestijne–Ulas encoding, the pseudocode of Tibouchi's encoding algorithm is shown in Algorithm 1. Although the code is not constant time, the execution time does not depend on the elliptic curve point itself, and therefore does not leak secret information. This algorithm uses two subroutines called $f(u)$ and `calc_v`(Q, j) that are described in Algorithm 2 and 3, respectively.

Algorithm 1 Encode curve point P as (u, v)

```
Require: Curve point $P$, curve parameters $a, b \in \mathbb{F}_p$, prime $p$
Ensure: Encoded values $(u, v)$ or failure
1: Sample $u \xleftarrow{\$} \mathbb{F}_p \setminus \{-1, 0, 1\}$
2: $Q \leftarrow P - f(u)$
3: if $Q = \mathcal{O}$ then goto 1
4: Sample $j \xleftarrow{\$} \{0, 1, 2, 3\}$
5: $v \leftarrow$ calc_v$(Q, j)$
6: if $v =$ None then goto 1
7: return $(u, v)$
```

Algorithm 2 Compute $f(u)$ — Maps $u \in \mathbb{F}_p$ to an elliptic curve point

```
Require: $u \in \mathbb{F}_p$, $u \notin \{-1, 0, 1\}$, elliptic curve parameters $a, b \in \mathbb{F}_p$, prime $p$
Ensure: A point $P = (x, y)$ on the curve or a placeholder for the point at infinity
1: $d \leftarrow u^2 \cdot (u^2 - 1) \mod p$
2: $X_0 \leftarrow -b \cdot a^{-1} \cdot (1 + d^{-1}) \mod p$
3: $y \leftarrow \sqrt{X_0^3 + aX_0 + b} \mod p$          ▷ Use modular square root
4: if $y$ exists then return $(X_0, y)$
5: $X_1 \leftarrow -u^2 \cdot X_0 \mod p$
6: $y \leftarrow \sqrt{X_1^3 + aX_1 + b} \mod p$          ▷ Guaranteed to exist
7: return $(X_1, -y \mod p)$
```

DecoyAuth Protocol

```
Client                                                  Server
$r_A \leftarrow\!\!\$\ \mathbb{F}_p$
$m_A \leftarrow\!\!\$\ \mathbb{F}_p$
$s_A = (r_A + m_A)$
$E_A = -m_A \cdot H(\pi)$
                              $\xrightarrow{\text{send } s_A, E_A}$
                                                        $s_B \leftarrow\!\!\$\ \mathbb{F}_p$
                                                        for $i = 0..n$ do
                                                            $m_{B,i} \leftarrow\!\!\$\ \mathbb{F}_p$
                                                            $r_{B,i} = (s_B - m_{B,i})$
                                                            $E_{B,i} = -m_{B,i} \cdot H(\pi_i)$
                                                            $D = D \cup \{(h(\pi_i), E_{B,i})\}$
                                                        $vals = encode(D)$
                              $\xleftarrow{\text{send } s_B, vals}$
$E_B = decode(vals, h(\pi))$
$K = r_A \cdot (s_B \cdot H(\pi) + E_B)$
$c_A = \mathrm{HMAC}(h(K), (s_A, E_A, s_B, E_B))$
                              $\xrightarrow{\text{send } c_A}$
                                                        $\pi = \text{None}$
                                                        for $i = 0..n$ do
                                                            $K = r_{B,i} \cdot (s_A \cdot H(\pi_i) + E_A)$
                                                            $c'_A = \mathrm{HMAC}(h(K), (s_A, E_A, s_B, E_{B,i}))$
                                                            if $c_A = c'_A$ then $\pi = \pi_i$
                                                        if $\pi = \text{None}$ then abort
                                                        $c_B = \mathrm{HMAC}(h(K), (s_B, E_B, s_A, E_A))$
                              $\xleftarrow{\text{send } c_B}$
$c'_B = \mathrm{HMAC}(h(K), (s_B, E_B, s_A, E_A))$
if $c_B \neq c'_B$ then abort
```

Figure 1: Design of the DecoyAuth protocol.

Algorithm 3 Compute $v = \texttt{calc\_v}(Q, j)$ from a curve point $Q = (x, y)$ and index j

Require: $q = (x, y)$ on curve, index $j \in \{0, 1, 2, 3\}$, curve parameters $a, b \in \mathbb{F}_p$, prime p
Ensure: A field element $v \in \mathbb{F}_p$ or failure

```
1:  $\omega \leftarrow (a \cdot b^{-1} \cdot x + 1) \mod p$
2:  $s \leftarrow \sqrt{\omega^2 - 4 \cdot \omega} \mod p$
3:  if $s$ does not exist then return None
4:  if $j = 2$ or $j = 3$ then $s \leftarrow -s \mod p$
5:  if $y$ is a quadratic residue in $\mathbb{F}_p$ then
6:      $m \leftarrow (2 \cdot \omega)^{-1} \mod p$
7:  else
8:      $m \leftarrow 2^{-1} \mod p$
9:  end if
10: $r \leftarrow \sqrt{(\omega + s) \cdot m} \mod p$
11: if $r$ does not exist then return None
12: if $j = 1$ or $j = 3$ then $r \leftarrow -r \mod p$
13: return $r$
```

4 DecoyChecker

This section describes an example two-server architecture that uses the DecoyAuth protocol to implement decoy-capable authentication.

4.1 Authentication Server

Similar to protocols such as Honeywords [8], HoneyPAKE [10], and SweetPAKE [1], we assume a system architecture consisting of two servers: an authentication server and a DecoyChecker. The authentication server is responsible for performing the actual user authentication via the DecoyAuth protocol. For each user, it stores the real password alongside an arbitrary number of decoy passwords. Note that the authentication server only stores a list of passwords: it does not know which password is the real password and which ones are the decoys. Upon a successful authentication attempt, the server determines which stored password was used and forwards the *index* of this password to the DecoyChecker.

4.2 Role of the DecoyChecker

The DecoyChecker is a separate component that ideally is running on a different operating system, with a minimal trusted computing base, and ideally isolated from the external network. It maintains only the information necessary to identify the correct password index for each user. Its sole responsibility is to verify whether the password used corresponds to the genuine credential or a decoy.

This separation of roles enhances security: if the authentication server is compromised and its password file exposed, an attacker cannot distinguish real from decoy passwords. Furthermore, any login attempt using a decoy password—whether through manual guessing or automated attacks—will be detected by the DecoyChecker, allowing timely alerts of potential breaches. This architecture ensures that if an adversary compromises the authentication server, they do not learn which passwords are real or decoys, meaning subsequent usage of a decoy can then be detected by the DecoyChecker.

4.3 Decoy Token Generation

It is essential that decoys are drawn from the same distribution as real secrets. If decoys are distinguishable from real passwords based on their format, frequency, or any structural bias, then an adversary may trivially exclude decoys and narrow down the set of valid candidates. This would effectively reduce the entropy of the secret space and undermine the intended security guarantees, such as resistance to offline dictionary attacks.

To ensure security, each decoy token must be computationally indistinguishable from an actual password under the assumed threat model. This often means sampling decoys from the same empirical password distribution or applying the same encoding and hashing processes. Even subtle biases in encoding or entropy can lead to distinguishability under statistical analysis. Therefore, careful attention must be paid to how decoys are generated and represented, which typically depends on the exact scenario in which the decoys are used and requires a (probabilistic) model of real passwords.

Concrete examples for algorithms on how to best generate honey passwords that are indistinguishable from real ones, and possible attacks otherwise, are given by Jules et al. [8], Chatterjee et al. [2], and Golla et al. [5].

5 Conclusion

The proposed DecoyAuth protocol supports decoy passwords with still ensuring strong security properties such as mutual authentication, forward secrecy, and resistance against off-line dictionary attacks. Regarding overhead, the server must loop through all passwords twice, and the data overhead is linear in function of the number of support decoy passwords. Future work includes further optimizing the protocol and providing a formal analysis and proof.

Acknowledgements

This work was funded by NGI Sargasso, the Research Fund KU Leuven, and the Cybersecurity Research Programme Flanders.

References

[1] A. Arriaga, P. Y. Ryan, and M. Skrobot. Sweetpake: Key exchange with decoy passwords. In *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security*, pages 1017–1033, 2024.

[2] R. Chatterjee, J. Bonneau, A. Juels, and T. Ristenpart. Cracking-resistant password vaults using natural language encoders. In *2015 IEEE Symposium on Security and Privacy*, pages 481–498. IEEE, 2015.

[3] H. Cheng, Z. Zheng, W. Li, P. Wang, and C.-H. Chu. Probability model transforming encoders against encoding attacks. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1573–1590, 2019.

[4] A. Faz-Hernandez, S. Scott, N. Sullivan, R. S. Wahby, and C. A. Wood. Hashing to Elliptic Curves. RFC 9380, 2023.

[5] M. Golla, B. Beuscher, and M. Dürmuth. On the security of cracking-resistant password vaults. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 1230–1241, 2016.

[6] D. Hankerson, A. J. Menezes, and S. Vanstone. *Guide to elliptic curve cryptography.* Springer Science & Business Media, 2004.

[7] D. Harkins. Simultaneous authentication of equals: A secure, password-based key exchange for mesh networks. In *2008 Second International Conference on Sensor Technologies and Applications (sensorcomm 2008)*, pages 839–844. IEEE, 2008.

[8] A. Juels and R. L. Rivest. Honeywords: Making password-cracking detectable. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 145–160, 2013.

[9] F. Kiefer and M. Manulis. Oblivious pake: Efficient handling of password trials. In *International Conference on Information Security*, pages 191–208. Springer, 2015.

[10] J. M. Lopez Becerra, P. Roenne, P. Ryan, and P. Sala. Honeypakes. In *Security Protocols XXVI: 26th International Workshop*. Springer International Publishing, 2018.

[11] M. Tibouchi. Uniform points on elliptic curves of prime order as uniform random strings. In *International Conference on Financial Cryptography and Data Security*, 2014.