

Zero-Knowledge and You: A Beginner's Guide to SPHiNX & Challenge-Response Protocols

Joshua J. Hollenbeck, Ryan J. Lewis, Patrick F. Wilbur
{hollejj,lewisrj,wilburpf}@clarkson.edu

December 12, 2008

1 Abstract

We present a method for zero-knowledge, hash-based challenge-response network authentication in lieu of transmitting a password across the network. Zero-knowledge protocols, like our method, offer authentication alternatives to prevent a third-party from discovering a password after intercepting network data. Our method makes use of one-way hash functions to generate a response from a randomly-created challenge code supplied by an identity verifier. Our method also makes use of dynamic engagement for choosing which hash functions are used on a per-case basis, in order to thwart reversal of our method in the event of future discovery of weaknesses in any of the deployed hash functions.

2 Introduction

Can you think of a question that would tell you whether a computer or a human being answered it? Or maybe a question that a computer cannot answer that a human would have no problems answering? Certain tests that are used now can determine whether the user is a human or a computer. These tests are considered turing tests.

Turing test; a test given by someone to subjects A and B to determine which is a computer and which is a human. The questioner is limited to using the responses to written questions in order to make the determination. A newer test in this category is the CAPTCHA (Completely Automated Public Turing test to tell Computers and Humans Apart). This is a very efficient and effective way to tell if a human is answering the question, and is frequently used at many sites. These tests follow this basic concept; an image is shown, then through image recognition (which is not easy for computers) the answer must be inserted in ASCII. Ultimately, given a challenge, a response can be created [2].

This non-cryptographic authentication was in use long before it became standard for Internet communications and security. [1] One reason for this was to make sure the system asking for the password was not a human trying to gain access. Channels were also a worry, due to listener's being able to get the password. To address the insecure channel problem, a more sophisticated approach is necessary.

2.1 Authentication Over a Network

One solution to this is to use two-way authentication. In such a system, both the user and the system must each convince the other that they know the key, without this secret ever being transmitted over the line. Once this is accomplished, then a connection will become established and information will be sent. This approach is evident in SPHiNX, which uses the challenge response protocol to authenticate that there is a human with the correct key on the other end.

Method	Strengths	Weaknesses
Transmission of a password	-	Susceptible to man-in-the-middle and easy identity theft
Encrypted transmission of a password	Harder to intercept	Still susceptible to identity theft during man-in-the-middle attacks or key compromise
Hash-based challenge-response	Identity cannot easily be stolen for future impersonations	Session hijacking can still take place if not performed in a secure medium, and a compromise of either party reveals original shared secret
Zero knowledge	Identity cannot easily be stolen for impersonation, and a compromise of the verification authority does not provide original shared secret	-

2.2 State of the Art

Modern implementations of two-way authentication use the idea of a cryptographic nonce, a number used once, to try to prevent reply attacks. A reply attack is executed by an eavesdropper by listening for and storing a requested challenge, and then listening for and storing the response. With both of those pieces of information, the eavesdropper can act like the server sent it the challenge it has stored, and send the appropriate response. A nonce attempts to prevent this by always using unique challenges, so anyone that would be stored by an eavesdropper would never be able to be used again.

Unfortunately, an eavesdropper could launch a more sophisticated attack, simulating the server by forwarding session keys, which is how the server would be able to keep track of what to expect for this instance of a challenge request. The client would think that the packet it just received was actually from the server, reply with the response, and the eavesdropper would intercept that as well.

Therefore, this usage of nonces should be used in conjunction with an asymmetric cryptosystem, such as Transport Layer Security.

3 SPHiNX Design

There exist two primary parties in our SPHiNX identity verification scheme—The Sphinx and Oedipus. The Sphinx is the authoritative party that wishes to verify the identity of a passerby. Oedipus is the passerby that wishes to be authenticated by The Sphinx. Both Oedipus and The Sphinx have previously and securely shared a secret, and Oedipus needs to somehow prove his identity to The Sphinx without revealing the shared secret out loud.

3.1 Algorithm Overview

Given a previously-negotiated secret that is shared between The Sphinx and Oedipus, The Sphinx chooses a random riddle that it will use for verifying the identity of Oedipus. This riddle is communicated over the air to Oedipus, who passes it and the shared secret to the main `solve_the_riddle` procedure.

```
1 function solve_the_riddle(secret , riddle) {
2   len1 := length(secret);
3   len2 := length(riddle);
4
5   if (len1 >= len2) {
6     riddle := secret;
7     len2 += len1;
8   }
9
10  part1, part2 := obscure(secret , riddle);
11  enc_part1 := the_nose_fell_off(part1);
12  enc_part2 := the_nose_fell_off(part2);
13
14  merged := obscure(enc_part1 , enc_part2);
15
16  return trim(the_nose_fell_off(merged));
17 }
```

Figure 1: solve_the_riddle Procedure

To generate Oedipus’s response to the riddle, the algorithm, as implemented in the `solve_the_riddle` procedure, performs the following steps:

1. Pads the riddle to ensure sane length
2. OBSCURE+SPLIT: Obscurely integrates components from both the key and the riddle, then obscurely splits the deck of merged components into two stacks
3. THE NOSE FELL OFF: Selects a hashing algorithm to perform on each stack based upon the values of each stack, then performs hashing on each stack

4. **OBSCURE**: Obscurely integrates the two hashed stacks into a single deck
5. **THE NOSE FELL OFF**: Selects a hashing algorithm to perform on the merged deck based upon the value of the deck, then performs hashing on the deck
6. Trims the hashed deck for uniform length independent of hashing algorithm used

3.1.1 Padding the Riddle

Our method begins by repeatedly padding the riddle with the secret key until the length of the riddle is equal to or greater than the length of the key.

```

1 | len1 := length(secret);
2 | len2 := length(riddle);
3 |
4 | if (len1 >= len2) {
5 |   riddle .= secret;
6 |   len2 += len1;
7 | }

```

Figure 2: Padding

In practice, if the shared secret and a riddle shorter than our shared secret are passed to our `obscure` function, `obscure` will not merge the two in a way that prevents the easy recovery of the shared secret. These potential problems with this case are alleviated by padding the riddle with sufficient repetitions of the secret key until it is of equal or greater length than the secret key.

A fundamental philosophy for promoting one-wayness from layer to layer, which we adhere to throughout SPHiNX, is the assumption that all subsequent layers could be compromised, and that it is the responsibility of each layer to thwart recovery of the input to that layer. Protecting the input to each layer is paramount for thwarting layer-by-layer defeat, which can help thwart eventual recovery of the shared secret.

3.1.2 Obscure and Split

Next, our method integrates the contents of the shared secret and the riddle into a single string, and then splits this string into two parts.

```

1  function obscure(str1 , str2) {
2    len1 := length(str1);
3    len2 := length(str2);
4
5    if (len1 > len2) {
6      temp := str2;
7      str2 := str1;
8      str1 := temp;
9      temp := len2;
10   len2 := len1;
11   len1 := temp;
12 }
13
14 merged := "";
15 newchar := "";
16 sum := 0;
17 j := 0;
18
19 for i := 0 to len2 {
20   j += i + ( str1[i % len1] * str2[i] ) + str2[i];
21   j %= len1;
22   newchar := str1[j] ^ str2[i];
23   sum += newchar;
24   merged .= newchar;
25 }
26
27 offset := sum % i;
28 part1 := merged[0..offset];
29 part2 := merged[offset..length(merged)];
30 return part1 , part2 , merged;
31 }

```

Figure 3: Obscure Procedure

This step can be seen as being analogous to taking two halves of a deck of playing cards, riffing them to reform an entire deck, and then cutting the deck while shuffling—the primary differences being that characters in the strings are not interwoven, but exclusively-ored, with one another, and that the riffing pattern and cutting position are not chosen pseudorandomly, but are calculated based upon the values of the shared secret and riddle strings.

3.1.3 The Nose Fell Off

The next step in our method is to perform hashing of each string part using a dynamic selection of available one-way functions based upon the values of each part.

```

1 | function the_nose_fell_off(msg) {
2 |     // ;D
3 |     hashes := {MD5, SHA1, SHA256};
4 |
5 |     sum := 0;
6 |     for i := 0 to length(msg) {
7 |         sum += msg[i];
8 |     }
9 |
10 |    which_hash := sum % length(hashes);
11 |    return hash(hashes[which_hash], msg);
12 | }

```

Figure 4: The Nose Fell Off Procedure

The `the_nose_fell_off` procedure contains a list of one-way functions to potentially be used on strings passed to the procedure, and selects the one-way function based upon the computed sum of all the string's characters. This procedure represents an ideally-irreversible state progression, much like the irreversible nature of the nose falling off of the actual Sphinx. Dynamically selecting a one-way function for hashing in this step thwarts later recovery of the pre-hashed string passed to this procedure, once further steps complete and truncate the response string to a uniform length that does not immediately reveal the one-way function used.

3.1.4 Finalization

For the remainder of our method, each hashed string part is then passed to the `obscure` procedure for integration into a merged string, which is not split into two parts like before but is instead passed in the merged state to the `the_nose_fell_off` procedure for a final round of hashing. The final hashed string is then trimmed to uniform length, to prevent an attacker's discernment as to which one-way function was used for hashing, and that trimmed value is returned as the response.

Oedipus then transmits the response to The Sphinx for approval. The Sphinx performs the same process on the shared secret and the riddle as Oedipus and, if both responses match, then it grants Oedipus authorization; otherwise, Oedipus's response is rejected as invalid, and Oedipus is assumed to be an imposter. This is analogous to the reaction of the actual Sphinx, with the exception that the actual Sphinx would not only deny passing but would also eat Oedipus.

4 Conclusion

Reversal of our method is extremely difficult. By using one-way hash functions, we can use this dynamic engagement to prevent anyone from intercepting the

data across the network and having access to the system. Challenge response protocols have an important role to fill when trying to surpass password vulnerabilities. These issues include keeping a dynamic challenge present, therefore even if somehow they saw the answer, they would have to get the same challenge; this is impossible to re-enact. So, is there a question that you can ask a human and be positive the one answering isn't a computer? Yes.

4.1 Improvements

One improvement that can be made to this cryptosystem is to use an incremental type of secret key generation. The idea is that the two parties exchange a key generator securely, just as they would the individual key normally. The prover requests a challenge from the verifier and can then choose to increase their secret key by some known amount, specified by the shared generator, and then create a response based on this new key. The new hash based on the increased key is sent and the verifier checks the response against it's key, at which point it would fail. The verifier then increases it's secret key and the generated hash would now match the received. This now provides users with a systematic way of increasing their keys and not having to use a secure medium more than once.

Another improvement is one that we implemented in SPHiNX: using a dynamic method of choosing a hashing algorithm. The key benefit being that this system is not vulnerable to weaknesses in a given hashing algorithm itself.

5 Disclaimer of Warranty

The methods described herein are presented WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

References

- [1] Sara Robinson. Human or computer? take this test, 2002. Retrieved on December 11, 2008 from <http://query.nytimes.com/gst/fullpage.html?sec=technology&res=9907E5DF163AF933A25751C1A9649C8B63>.
- [2] Carnegie Mellon University. Telling humans and computers apart automatically, 2007. Retrieved on December 11, 2008 from <http://www.captcha.net/>.