To Whom It May Concern:

The Citizen Lab is an academic research group based at the Munk School of Global Affairs & Public Policy at the University of Toronto in Toronto, Canada.

We analyzed Sogou Pinyin Method on Windows, Android, and iOS as part of our ongoing work analyzing popular mobile and desktop apps for security and privacy issues. We found that the Windows and Android versions of Sogou Pinyin Method contain vulnerabilities, including a vulnerability to a CBC padding oracle attack, which allows network eavesdroppers to recover the plaintext of encrypted network transmissions, revealing sensitive information including what users have typed. In the case of the Android version, we are also able to recover the second halves of the symmetric encryption keys used to encrypt traffic. We also found vulnerabilities affecting the encryption implemented in the iOS version, but we are not presently aware of methods to exploit these vulnerabilities in the version which we analyzed. For further details, please see the attached document.

Background

The Citizen Lab is committed to research transparency and will publish details regarding the security vulnerabilities it discovers in the context of its research activities, absent exceptional circumstances, on its website: https://citizenlab.ca/.

If *no response is received* to this disclosure, the Citizen Lab will publish details regarding the security vulnerability on its website after **15 calendar days** from the date of this communication. In other words, where there is no response from you, Citizen Lab will publish details regarding the vulnerability after June 15 2023.

If *a substantive response is received* (which excludes, for example, an auto reply) to this disclosure **within 15 calendar days** from the date of this communication, the Citizen Lab will provide you with **45 calendar days** from the date of this communication to fix (whether in whole or in part) the vulnerability before publicly disclosing the issue. In other words, where we do receive a substantive response from you, the Citizen Lab will publish details regarding the vulnerability after July 15 2023.

We reserve the right to publish details regarding the vulnerability to the general public before the expiry of the 45 calendar days set out above in the following situations: (1) you have disclosed the vulnerability to the general public, (2) you have patched the vulnerability, (3) you have taken the position that there is no security vulnerability, or (4) the Citizen Lab observes the vulnerability is under active exploitation.

All communications associated with this disclosure may be included in the Citizen Lab's public disclosure of this vulnerability.

Next steps

Please communicate what steps you will take to address the vulnerability that we have described, and please provide the timeline you decide upon for the implementation of fixes.

Finally, upon implementation of any fixes, we ask that you communicate the full extent of the vulnerability to

the Citizen Lab.

Should you have any questions about our findings please let us know. We can be reached at this email address: disclosure@citizenlab.ca.

Sincerely,

The Citizen Lab

Summary

The Citizen Lab is an academic research group based at the Munk School of Global Affairs & Public Policy at the University of Toronto in Toronto, Canada.

We analyzed Sogou Pinyin Method on Windows, Android, and iOS as part of our ongoing work analyzing popular mobile and desktop apps for security and privacy issues. We found that the Windows and Android versions of Sogou Pinyin Method contain vulnerabilities, including a vulnerability to a CBC padding oracle attack, which allow network eavesdroppers to recover the plaintext of encrypted network transmissions, revealing sensitive information including what users have typed (see Table 1 for a breakdown of versions analyzed). In the case of the Android version, we are also able to recover the second halves of the symmetric encryption keys used to encrypt traffic. We also found vulnerabilities affecting the encryption implemented in the iOS version, but we are not presently aware of methods to exploit these vulnerabilities in the version which we analyzed.

Platform	**Version analyzed**	**Exploitable?**
Windows	13.4	Yes
Android	11.20	Yes
iOS	11.21	No known exploit

Table 1: Summary of versions of Sogou Pinyin Method affected.

Findings

In this section we discuss our attacks on Sogou's "EncryptWall" encryption system. We begin by giving background on the encryption system, then detailing our attack on it, and finally we break down how the attack applies to the three platforms which we analyzed, adapting our attack for all deviations in the implementation of the EncryptWall system across platforms.

Background

The attacks which we discuss in this report concern vulnerabilities which we found in Sogou's "EncryptWall" encryption system which is intended for securely tunneling sensitive traffic to unencrypted Sogou HTTP API endpoints via encrypted fields in HTTP POST requests. In this report we call the outer HTTP request the *EncryptWall* request and the tunneled HTTP request the *tunneled* request. Although there were differences in the implementation across the three platforms that we analyzed, we found that the system generally works as follows:

- An EncryptWall request is sent as an HTTP POST request to a Sogou EncryptWall API endpoint containing at least five HTTP form fields specifying cryptographic parameters used to encrypt the tunneled request as well as the encrypted tunneled data. Two form fields relate to specifying the key and initialization vector (IV) used to encrypt other fields in the EncryptWall request:
    - *K* – the base64 encoding of the encryption of a 256-bit AES key *k* with a 1024-bit public RSA key using PKCS#v1.5 padding; *k* is generated randomly for each request
    - *V* – the base64 encoding of a 128-bit initialization vector *v*; *v* is generated randomly for each request
- Three of the form fields are individually zlib compressed, encrypted using *k* and *v*, and base64-encoded according to the following pseudo-code:

encrypt(*data*) = base64_encode(AES_cbc_encrypt(zlib_compress(*data*, wbits=-15), *k*, *v*))

The three form fields we consistently observed encrypted in this manner are as follows:

- *U* – encrypt(the URL of the tunneled HTTP request)
- *G* – encrypt(any GET parameters for the tunneled HTTP request in the form of a query string)
- *P* – encrypt(the raw POST data for the tunneled HTTP request, if any)

Depending on the platform analyzed and the type of request being made, the EncryptWall request may be sent over encrypted HTTPS or plain HTTP. In cases where EncryptWall requests were made over HTTPS, we believe that the requests are secure against network eavesdropping, despite any defects which might exist in the underlying cryptography of the EncryptWall request on account of the HTTPS's TLS cryptography additionally protecting it. Thus, our findings in the remainder of this section only concern EncryptWall requests which we observed being made over plain HTTP which do not benefit from the additional protection of HTTPS.

Attack

We found that the EncryptWall system is vulnerable to a CBC padding oracle attack, a type of chosen ciphertext attack originally published in 2002 impacting block ciphers using cipher block chaining (CBC) block cipher mode and PKCS#7 padding. In such an attack, the plaintext of a message can be recovered one byte at a time, using at most 256 messages per byte. While we do not intend to fully reiterate how this attack works here, the attack relies on the existence of a certain kind of side channel called a padding oracle which reveals unambiguously whether the received ciphertext, when decrypted, is correctly padded. We identified such an oracle in the EncryptWall system: we found that a ciphertext sent in the *U* form field returns an HTTP 400 status code when it contains incorrect padding, whereas, when correctly padded, it returns either a 200 status or 500 status code depending on whether the decrypted URL is a valid URL or not, respectively. By performing a CBC padding oracle attack, this padding oracle allows us to

not only reveal the entire plaintext of *U* but also *G* and *P*, since they use the same key and initialization vectors. Thus, by using this padding oracle we can decrypt the contents of the entire EncryptWall request.

In the remainder of this section, we adapt this attack for all deviations in the implementation of the EncryptWall system on the Windows and Android platforms. Although they do not presently appear exploitable, we also detail defects in the EncryptWall system on iOS.

Windows version 13.4

The EncryptWall system implemented in the Windows version which we analyzed deviated from the standard implementation described above in one detail, namely that the IV *v*, instead of being public, was encrypted in the same manner as the AES key *k*. Due to this discrepancy, *v* is not immediately known, which is potentially problematic for our attack for two reasons: first, in the CBC padding oracle attack, the IV must be known in order to decrypt the first block of plaintext. Second, since the data tunneled in the EncryptWall requests is compressed before being encrypted, the first block of plaintext is important for decompressing the remaining blocks.

However, we developed a method to recover *v* which exploits the fact that *v* is reused to encrypt multiple plaintexts. Specifically, since the URL *U* is easily predictable and is ever only one of a small number of possible endpoints, we are able to recover *v* by performing a CBC padding oracle attack on the first ciphertext block of *U*, assuming an all zero IV. The result of this attack will be the first plaintext block of the URL XORed with *v*. We then XOR this result with our prediction for the first plaintext block of the URL, yielding *v* alone. With *v* recovered, we can perform the CBC padding oracle attack on *G* and *P* as usual.

```
1 {
  1: 1
  2 {
    1 {
      2: "1111_sogou_pinyin_guanwang_13.4e_1111"
      3: "13.4.0.7561"
      5: 3
      7: 1
      8: "13.4.0.7561"
    }
    7: "nihaohaohaohaohaohaozdaasdfffaahellocanyoureadthis"
    16: 11
    17 {
      3 {
        1: 2
        2: 1
      }
      9: 1
      10: 1
    }
    19 {
      4: "0"
    }
```

Figure 1: Example of the recovered protobuf data; line 11 contains the typed text.

As one example of the kind of transmitted data vulnerable to this attack, we found that for EncryptWall requests sent to http://get.sogou.com/q, when *U* was http://master-proxy.shouji.sogou.com/swc.php, *G* contained version information pertaining to Sogou's software, and *P* was a protobuf buffer containing the keystrokes which had been recently typed in (see Figure 1 for an example). We believe that these transmissions are related to a cloud-based implementation of an autocompletion service. Since these transmissions are vulnerable to our attack, the keystrokes of Sogou Pinyin Method users are an example of what a network eavesdropper could decrypt, informing the eavesdropper of what these users are typing as they type.

Android version 11.20

The Android version which we analyzed adopts the standard implementation of EncryptWall but with the inclusion of four additional form fields: *R*, *S*, *E*, and *F*. The field *R* transmits another 32-byte key *r*. Notably, however, each byte of *r* is randomly chosen from the 36-character set of ASCII uppercase letters and numbers. Therefore, instead of $256^{32} = 2^{256}$ bits of entropy, the key only has $36^{32} < 2^{166}$ bits of entropy. Furthermore, unlike *k*, *r* is not generated randomly for each request and is only generated once per application lifetime. The field *R* is then transmitted as the base64 encoding of $k \oplus r$. Note that due to this transmission, *k*'s entropy is also reduced to $36^{32} < 2^{166}$ bits of entropy. The parameters *k*, *r*, and *v* are used to encode *S*, *E*, and *F* according

to the following pseudo-code:

encryptAndroid(*data*) = base64_encode(*k* ⊕ AES_cbc_encrypt(*data*, , *r*, “EscowDorisCarlos”))

Note that unlike the typical encrypt() function, encryptAndroid() features a hard-coded IV “EscowDorisCarlos” and no zlib compression. Additionally, although encryptAndroid() uses *r* instead of *k* as an AES key, *k* is additionally XORed with the result of the AES encryption. Each of the fields *S*, *E* and *F* are individually encrypted and encoded according to the encryptAndroid() function.

Despite the use of this modified cryptography, we were still able to successfully attack the encryption of these fields. We were able to apply the CBC padding oracle attack, using Sogou’s processing of the *E* form field instead of the *U* form field that we typically would use, except we had to make the following two accommodations.

First, since the key *k* is 32 bytes but AES blocks are 16 bytes, when the output of the AES block cipher is XORed with *k*, we can think of the output being XORed with two keys k_1 *and* k_2, where k_1 is XORed with odd-numbered blocks (1, 3, …) and k_2 is XORed with even-numbered blocks (2, 4, …) (see Figure 2 for an illustration). Thus, when performing the CBC padding oracle attack, we had to ensure that the block that we were attacking was in an even-numbered position if it was originally even-numbered or in an odd-numbered position if it was originally odd-numbered. In other words, we had to preserve the parity of the block’s position.

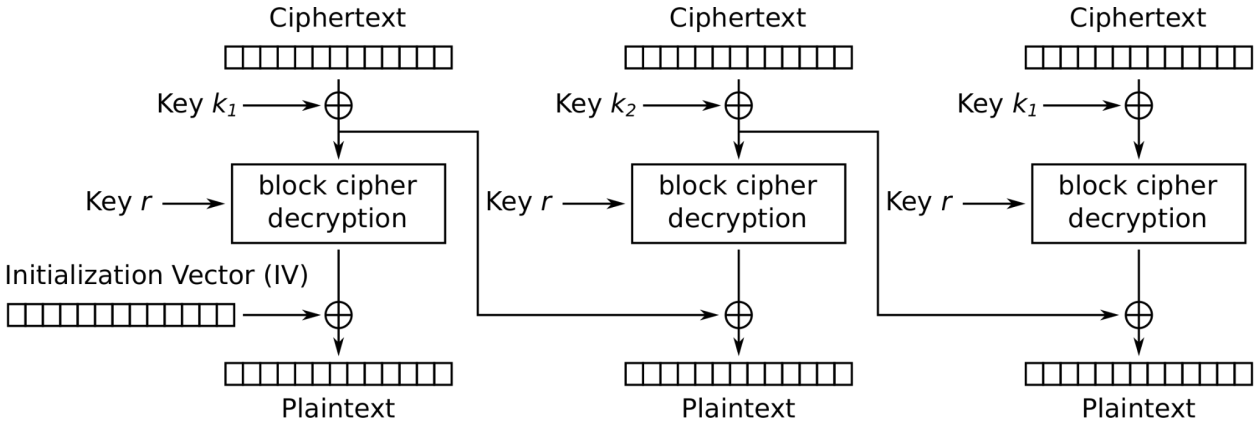

Figure 2: A modified version of CBC in which a 32-byte key $k = k_1 \| k_2$ *composed of two 16-byte keys* k_1 *and* k_2 *is XORed with ciphertext blocks before being decrypted by the block cipher such that* k_1 *is XORed with odd-numbered blocks (1, 3, …) and* k_2 *is XORed with even-numbered blocks (2, 4, …).*

Second, since the IV is hard-coded, we cannot modify it and thus, similar to the Windows version, the CBC padding oracle attack cannot recover the first block of plaintext p_1 without an adaptation. Namely, we found that p_1 was still recoverable by first treating the IV, “EscowDorisCarlos”, as a ciphertext block c_0 preceding the first ciphertext block c_1 during the

attack. Since c_1 must be in an odd-numbered position, c_0 is in an even-numbered position. Thus, during the attack, the oracle first XORs c_0 with k_2 when decrypting the first ciphertext block. Resultantly, our attack produces $p_1 \oplus k_2$ instead of merely p_1. However, we also found that the plaintext contents of the first plaintext block of the form field *S* were highly predictable. Namely, they contained the version of Sogou being used, which was already transmitted in the plain as an HTTP header of the EncryptWall request and thus would be available to any network eavesdropper. Knowing this plaintext, then, by applying the CBC padding oracle attack to *S*, we can recover *S*'s first plaintext block XORed with k_2, and since we can predict that first plaintext block itself, we can XOR it with the result, yielding k_2 alone. Once we know k_2, for decrypting both *E* and *F*, we can now perform the CBC padding oracle attack using c_0 = "EscowDorisCarlos" $\oplus$ k_2 to recover their entire plaintext including the first block.

Additionally, we can now also recover the second half of *r*, r_2, which is beneficial to an attacker in that our knowledge of r_2 can be used to more easily recover k_2 in subsequent requests. Recall that the form field *R* encodes $k \oplus r$. Thus, after recovering k_2 we can recover r_2 by XORing the second half of *R*'s encoded contents with k_2. Once r_2 is recovered, since *r*, unlike *k*, is generated once per application lifetime, we can more easily recover k_2 in future requests by simply XORing the second half of *R* with r_2, making the attack even easier to perform in the future. Furthermore, this reduces the entropy of *r*, and thus, also *k*, to $36^{16} < 2^{83}$ bits.

```
1 {
  1: "com.android.messaging"
  2: "11.20"
  4: 1
  6: "android_sweb"
  8: "Google"
  10: "android_sweb"
  11: "11.20"
  14: "30"
  18: "-1"
  22: "5682b3aa4fa7bd40d776c93a35a77c6d"
}
2 {
  1: 0xbff0000000000000
  2: 0xbff0000000000000
  3: "-1"
}
3: 1
4: "canyoureadthis"
11 {
  1: "onekeyimageenable"
  2: "1"
}
```

Figure 3: Example of the recovered protobuf data; line 19 contains the typed text and line 2 contains the package name of the app in which the text is being typed.

As one example of the kind of transmitted data vulnerable to this attack, we found that for EncryptWall requests sent to http://v2.get.sogou.com/q, when *U* was http://swc.pinyin.sogou.com/swc.php, *P* was a protobuf buffer containing all of the text currently present in the input field in which the user is currently typing as well as the package name of the app in which the text was being typed (see Figure 3 for an illustration). These transmissions occurred when pressing the magnifying glass icon, and we believe that these transmissions are related to a Giphy-like feature in which typed text is searched against a database of animations and memes which can be inserted into the typed message. Since these transmissions are vulnerable to our attack, the keystrokes of Sogou Pinyin Method users are an example of what a network eavesdropper could decrypt, informing the eavesdropper of what these users are typing as they are typing.

As one other example of the kind of transmitted data vulnerable to this attack, we found that for EncryptWall requests sent to http://v2.get.sogou.com/q, when *U* was http://update.ping.android.shouji.sogou.com/update.gif, *P* was a query string containing a list of every app installed on the Android device. We are unaware of what feature this data transmission is intended to implement.

iOS version 11.21

The iOS version which we analyzed had no major deviations from the standard EncryptWall implementation. However, unlike on some platforms where we saw some EncryptWall requests sent over encrypted HTTPS and others over plain HTTP, all EncryptWall requests which we observed transmitted by the iOS version which we analyzed were transmitted over HTTPS and thus we believe them to be secure against network eavesdropping. However, we note that without the additional protection of HTTPS, the iOS version would have been the most vulnerable due to the existence of an additional defect in the implementation of EncryptWall. Namely, we found that the iOS version randomly chooses the key *k* and IV *v* according to the following code in Figure 4:

```
void __cdecl +[DataEncryptor randomizeAesKeyIv:keyLen:iv:ivLen:](
        id a1,
        SEL a2,
        unsigned __int8 *key,
        ssize_t key_len,
        unsigned __int8 *iv,
        ssize_t iv_len)
{
  unsigned __int8 *iv_; // x20
  unsigned __int8 *key_; // x22
  unsigned int key_seed; // w0
  unsigned int iv_seed; // w0

  if ( key )
  {
    iv_ = iv;
    if ( iv )
    {
      key_ = key;
      key_seed = time(0LL);
      srand(key_seed);
      if ( key_len >= 1 )
      {
        do
        {
          *key_++ = rand();
          --key_len;
        }
        while ( key_len );
      }
      iv_seed = time(0LL);
      srand(iv_seed);
      if ( iv_len >= 1 )
      {
        do
        {
          *iv_++ = rand();
          --iv_len;
        }
        while ( iv_len );
      }
    }
  }
}
```

Figure 4: Decompiled code for generating AES key and IV. Note that the random number generator is seeded with the current time, rounded down to a whole second, before generating the key and again before generating the IV.

Note that before randomly generating the key and again before randomly generating the IV the random number generator is seeded with the current time as seconds since the Unix epoch, rounded down to a whole second. There are two consequences to this behavior: first, the only information needed to derive the AES key *k* is the time which the request was sent, which any network eavesdropper would be able to easily record. Second, since the random number

generator is re-seeded before generating the IV *v* with what will almost always be the same time in seconds after rounding, *v* is almost always the first 128 bits of *k*. Since *v* is public, all EncryptWall messages reveal the first half of *k* in *v*, despite the fact that *k* is encrypted with a public RSA key.

However, we note again that this defect is not currently exploitable since EncryptWall requests on iOS appear to always be additionally wrapped in HTTPS. However, due to the severity of the defect, we are nevertheless compelled to mention it on account of the fact that previous versions of the iOS version may be impacted and because this code may be reused in other apps which may be vulnerable.

Mitigation

In order to address the reported issues, Sogou Pinyin Method should secure all transmissions using a popular, up-to-date implementation of HTTPS or, more generally, TLS instead of relying on custom-designed cryptography to secure the transmission of sensitive user data. Moreover, Sogou Pinyin Method should not transmit data unnecessary for the functionality of the program.