“Please do not make it public”

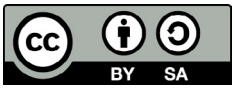
Vulnerabilities in Sogou Keyboard encryption expose keypresses to network eavesdropping

By Jeffrey Knockel, Zoë Reichert, and Mona Wang

AUGUST 9, 2023
RESEARCH REPORT #170

munk school
OF GLOBAL AFFAIRS & PUBLIC POLICY
UNIVERSITY OF TORONTO
THECITIZENLAB

Copyright



Electronic version first published by the Citizen Lab in 2023. This work can be accessed through https://citizenlab.ca/2023/08/vulnerabilities-in-sogou-keyboard-encryption/.

Document Version: 1.0

About the Citizen Lab, Munk School of Global Affairs & Public Policy, University of Toronto

The Citizen Lab is an interdisciplinary laboratory based at the Munk School of Global Affairs & Public Policy, University of Toronto, focusing on research, development, and high-level strategic policy and legal engagement at the intersection of information and communication technologies, human rights, and global security.

We use a "mixed methods" approach to research that combines methods from political science, law, computer science, and area studies. Our research includes investigating digital espionage against civil society, documenting Internet filtering and other technologies and practices that impact freedom of expression online, analyzing privacy, security, and information controls of popular applications, and examining transparency and accountability mechanisms relevant to the relationship between corporations and state agencies regarding personal data and other surveillance activities.

Acknowledgements

We would like to thank Jakub Dalek, Pellaeon Lin, Adam Senft, and Mari Zhou for valuable editing and peer review. Research for this project was supervised by Ron Deibert.

Suggested Citation

Jeffrey Knockel, Zoë Reichert, and Mona Wang. "'Please do not make it public': Vulnerabilities in Sogou Keyboard encryption expose keypresses to network eavesdropping." Citizen Lab Report No. 170, University of Toronto, August 2023. https://citizenlab.ca/2023/08/vulnerabilities-in-sogou-keyboard-encryption/.

Contents

We urge Sogou Input Method users to immediately update to the most recent version of the app (at least Windows version 13.7, Android version 11.26, or iOS version 11.25).

Key findings

- **We analyzed Tencent's Sogou Input Method, which, with over 450 million monthly active users, is the most popular Chinese input method in China.**
- **Analyzing the Windows, Android, and iOS versions of the software, we discovered troubling vulnerabilities in Sogou Input Method's custom-designed "EncryptWall" encryption system and in how it encrypts sensitive data.**
- **We found that network transmissions containing sensitive data such as those containing users' keystrokes are decipherable by a network eavesdropper, revealing what users are typing as they type.**
- **We disclosed these vulnerabilities to Sogou developers, who released fixed versions of the affected software as of July 20, 2023 (Windows version 13.7, Android version 11.26, and iOS version 11.25).**
- **These findings underscore the importance for software developers in China to use well-supported encryption implementations such as TLS instead of attempting to custom design their own.**

Introduction

Compared to typing alphabetic languages whose small number of letters can be represented uniquely by keys, typing logographic languages such as Chinese is more difficult. Chinese has tens of thousands of characters used in varying frequencies, far too many to fit on a single keyboard. There is no standard method of typing Chinese characters, but with the advent of modern technology a number of complementary approaches have emerged. The most popular is the pinyin method, based on the pinyin romanization of Chinese characters. Zhuyin is another popular phonetic input method, and shape or stroke-based input methods like Cangjie or Wubi are also commonly used. Modern input methods also support inputting characters via handwriting, voice recognition, and photograph or OCR (see Figure 1 for illustrations).

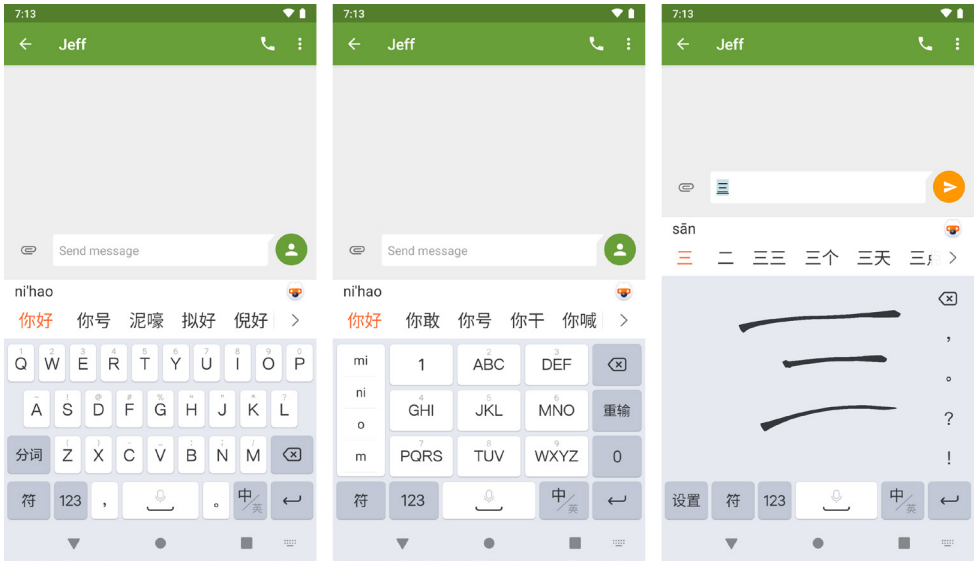


Figure 1: Example of three of the different Chinese input methods supported by the Android version of Sogou Input Method. The first two are pinyin-based inputs, whereas the third is based on handwriting or drawing characters. Sogou also supports wubi-based input, photograph input, and a "rare characters" keyboard which is based on inputting the pinyin for characters' individual components or radicals.

While alphabetic keyboards typically provide autocomplete features for more expedient typing, predictive features in Chinese input methods are more crucial when using input methods such as pinyin where hundreds of characters might match an inputted pinyin syllable. For longer strings of syllables, an IME will commonly reach out over the network to a cloud-based service for suggestions if suitable suggestions are not available in the input method's local database.

In this report, we analyze Tencent's Sogou Input Method, the most popular Chinese input method with over 455 million monthly active users and versions of the app for multiple platforms, including Windows, Android, and iOS. Sogou Input Method accounts for 70% of Chinese input method users, with products by iFlytek and Baidu taking second and third place, respectively. McAfee's 2015 analysis previously observed that the Windows version of the app transmitted device identifiers in the clear without any encryption, but it did not analyze the safety of data transmitted by the app's encryption system.

We analyzed Sogou Input Method on three operating system platforms, finding that the app has troubling vulnerabilities in its custom-designed encryption system which render sensitive data such as the keystrokes that users type decipherable to network eavesdroppers. The vulnerabilities which we discover are not limited to Chinese writers in China, as market research estimates concerning visitation to the app's website put United States users as comprising over 3.3% of visits, Taiwan as nearly 1.8%, and Japan as over 1.5%.

The remainder of this report is structured as follows. In the "Methodology" section, we outline the reverse engineering tools and techniques we used to analyze Sogou Input

Method. In "Findings", we describe how Sogou's custom-designed encryption system works, the vulnerabilities which we discovered in it, as well as examples of impacted data transmissions. In "Mitigation" and "Coordinated disclosure", we discuss how Sogou can fix the vulnerabilities that we reported to them and how we reported the vulnerabilities to them. Finally, in "Discussion" we reflect on how these vulnerabilities speak to systemic issues in the larger Chinese app ecosystem.

Methodology

We analyzed the Windows, Android, and iOS versions of Sogou Input Method. To procure the versions we analyzed, in May 2023, we downloaded the latest versions of the Windows and Android versions from the product website (the Android version of Sogou Input Method, while available as recently as June 3, 2021, is presently not available in the Google Play Store). We procured the iOS version from Apple's App Store (see Table 1 for a breakdown of versions analyzed).

Platform	Sogou Input Method Version	Device
Windows 7 SP1	13.4	Virtual machine
Android 9	11.20	Google Pixel 2
iOS 14.8	11.21	iPhone SE 2nd generation

Table 1: Breakdown of versions of Sogou Input Method analyzed and the environments in which they were analyzed.

We analyzed these versions of Sogou Input Method using both static and dynamic analysis methods. We used jadx to statically analyze and decompile Dalvik bytecode and IDA Pro to statically analyze and decompile native machine code. We used frida to dynamically analyze the Android and iOS versions and IDA Pro to dynamically analyze the Windows version. Finally, we used Wireshark and mitmproxy to perform network traffic capture and analysis.

Findings

We found that each version of Sogou Input Method encrypts sensitive data using an encryption system that is internally referred to as the "EncryptWall" encryption system. We found that the Windows and Android versions of Sogou Input Method contain vulnerabilities in this encryption system, 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 2 for a breakdown of versions affected). 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	Exploitable?
Windows	Yes
Android	Yes
iOS	No known exploit

Table 2: Summary of versions of Sogou Input Method affected.

In the remainder of this section we detail 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, or whether, the attack applies to the three platforms which we analyzed, adapting our attack for deviations in the implementation of the EncryptWall system across platforms.

Sogou's EncryptWall

The attacks which we discuss in this report concern vulnerabilities that we found in Sogou's "EncryptWall" encryption system, which appears intended for securely tunneling sensitive traffic to unencrypted Sogou HTTP API endpoints via encrypted fields in plain HTTP POST requests. In this report we call the outer, plain HTTP request the *EncryptWall* request and the single tunneled HTTP request each EncryptWall request encapsulates 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:
  a. "K" – the base64 encoding of the encryption of a 256-bit AES key *k* with a hard-coded 1024-bit public RSA key using PKCS#v1.5 padding; *k* is generated randomly for each request
  b. "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 that 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 that we analyzed deviated from the basic 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* that 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 {
2   1: 1
3   2 {
4     1 {
5       2: "1111_sogou_pinyin_guanwang_13.4e_1111"
6       3: "13.4.0.7561"
7       5: 3
8       7: 1
9       8: "13.4.0.7561"
10    }
11    7: "nihaohaohaohaohaohaozdaasdfffaahellocanyoureadthis"
12    16: 11
13    17 {
14      3 {
15        1: 2
16        2: 1
17      }
18      9: 1
19      10: 1
20    }
21    19 {
22      4: "0"
23    }
```

Figure 2: Example excerpt of 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 that had been recently typed in (see Figure 2 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 Input Method users can be decrypted by a network eavesdropper, informing the eavesdropper of what users are typing as they type**.

Android version 11.20

The Android version which we analyzed adopts the basic 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 as it is cached in C static memory. 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:

```
encryptSEF(data) = base64Encode(k ⊕ AES_cbc_encrypt(data, r,
"EscowDorisCarlos"))
```

Note that unlike the typical encrypt() function, encryptSEF() features a hard-coded IV "EscowDorisCarlos" and no zlib compression. Additionally, although encryptSEF() 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 encryptSEF() 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, with the exception of 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 3 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.

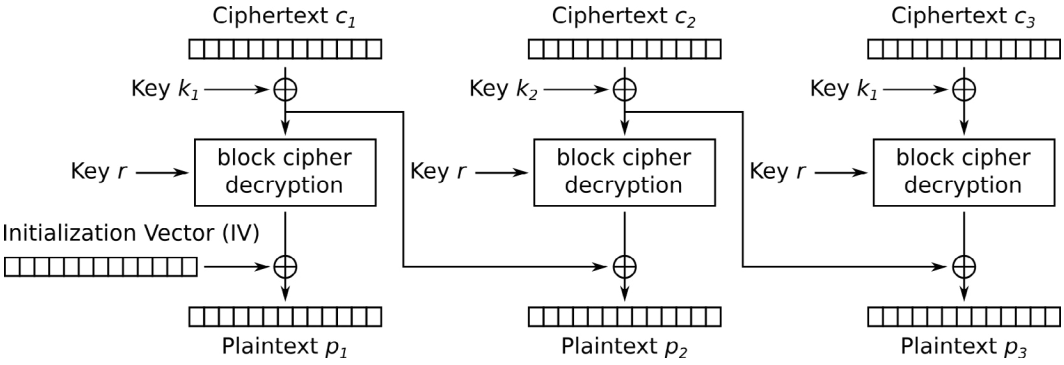


Figure 3: A modified version of CBC in which a 32-byte key $k = k_1 \| k_2$ *composed of two 16-byte keys* k_1 *and* k_2 *is XORed with ciphertext blocks before being decrypted by the block cipher such that* k_1 *is XORed with odd-numbered blocks (1, 3, …) and* k_2 *is XORed with even-numbered blocks (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 for form fields "S", "E", and "F" via the following procedure:

1. We treat the fixed IV, "EscowDorisCarlos", as a ciphertext block c_0 preceding the first ciphertext block c_1 and send it to the oracle. Since c_1 must be in an odd-numbered position, we ensure that 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 c_1.
2. Resultantly, decryption of c_1 produces p_1', which is equal to $p_1 \oplus$ "EscowDorisCarlos" $\oplus\, c_0 \oplus k_2$.
3. Since (per step 1) *c0* = "EscowDorisCarlos", p_1' is merely $p_1 \oplus k_2$. Therefore, by applying steps 1–3, we recover $p_1 \oplus k_2$ for each of fields "S", "E", and "F".
4. Moreover, we also found that the 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 clear as an HTTP header of the EncryptWall request and thus would be available to any network eavesdropper. Thus, in the case of form field "S", we know p_1. In step 3, we recovered $p_1 \oplus k_2$ for form field "S". Since we know p_1 and $p_1 \oplus k_2$, we have therefore recovered k_2.
5. Once we know k_2, which is the same value for fields "S", "E", and "F", since (per step 3) we know $p_1 \oplus k_2$ for fields "E" and "F", we can recover p_1 for "E" and "F" as well.

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 the "R" field'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 {
2   1: "com.android.messaging"
3   2: "11.20"
4   4: 1
5   6: "android_sweb"
6   8: "Google"
7   10: "android_sweb"
8   11: "11.20"
9   14: "30"
10  18: "-1"
11  22: "5682b3aa4fa7bd40d776c93a35a77c6d"
12 }
13 2 {
14   1: 0xbff0000000000000
15   2: 0xbff0000000000000
16   3: "-1"
17 }
18 3: 1
19 4: "canyoureadthis"
20 11 {
21   1: "onekeyimageenable"
22   2: "1"
23 }
```

Figure 4: Example excerpt of 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 observed 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 4 for an illustration). These transmissions occurred when pressing the magnifying glass icon, and we believe that these transmissions are related to an image search 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 Input 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 observed 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. While one can imagine knowing

which app a user is presently using may be useful for providing better typing suggestions in that app, it is difficult to imagine how knowing every app that a user has installed can provide better typing suggestions, even apps which users do not intend to use with Sogou Input Method.

iOS version 11.21

The iOS version which we analyzed had no major deviations from the basic 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 that 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 5:

```
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 5: 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 Input 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 Input Method should not transmit data unnecessary for the functionality of the program.

Coordinated disclosure

On May 31, 2023, we disclosed our findings to Tencent in a letter attached here, following our security disclosure vulnerability policy. Below in Table 3 is our disclosure timeline:

Date	Contact
May 31 2023	Vulnerability disclosed to IMETS@tencent.com.
June 16 2023	Vulnerability disclosed again via Tencent Security Response Centre (TSRC) web portal.
June 25 2023	We received the following response via the TSRC portal: *"Thank you for your interest in Tencent security. There is no low or low security risk for this issue. We look forward to your next more exciting report."*

Date	Contact
June 25 2023	Eighteen hours later, we received the following response via the TSRC portal: *"Sorry, my previous reply was wrong, we are dealing with this vulnerability, please do not make it public, thank you very much for your report."* Tencent's initial rejection of our disclosure and subsequent about-face served as inspiration for the title of this report.
June 26 2023	We sent the following message via the TSRC portal: *"Thank you for the update. We will publicly disclose the vulnerability after July 31, 2023."*
June 28 2023	We received the following response via the TSRC portal: *"Thank you very much for your report, repair plan and repair time, which have been replied to disclosure@citizenlab.ca by email."*
June 28 2023	We sent the following message via the TSRC portal: *"We have not received such an email at that address. However, it has come to our attention that our domain (citizenlab.ca) may not be accessible from China, and therefore emails from China may not be deliverable to it. Could you send a copy of the email you sent to disclosure@citizenlab.ca to another email address of mine, [redacted]@utoronto.ca ? I believe that there should be no issue delivering emails from China to this utoronto.ca address. Thank you."*
June 29 2023	We received the following response via the TSRC portal: *"The email we sent is security@tencent.com, the subject line is: Reply Sogou Pinyin Method vulnerabilities,which may have been classified as junk mail?"*
June 29 2023	We sent the following message via the TSRC portal: *"Unfortunately we have not received such an email at that address, not even in our spam folder. Would you be able to try sending a copy of the email to another email address of mine, [redacted]@utoronto.ca ? Thank you."*
July 4 2023	We received the following response via the TSRC portal: *"Can you use disclosure@citizenlab.ca to send an unsolicited email to security@tencent.com? Then I'll send the fix details to [redacted]@utoronto.ca."*
July 4 2023	We sent the following message via the TSRC portal: *"Yes, we have now sent such an email and are awaiting your response."*

Date	Contact
July 4 2023	We received the response attached here at the [redacted]@utoronto.ca email address. In the email response, Sogou Input Method developers outline a partial mitigation which they had already deployed by the date of the email as well as a timeline to migrate all platforms to use TLS encryption by July 31, 2023.
July 18 2023	We found that Sogou Input Method developers had released versions of the app for each platform which they had identified in previous correspondence as being the versions to fix the issues we identified. Finding that the Windows and iOS versions addressed the issues we reported but not the Android version, we sent the following message via the TSRC portal: *"Hello again. In the email you sent us you indicated that version 11.25 of the Android app would be upgraded to send EncryptWall requests using HTTPS. We analyzed version 11.25 (SogouInput_11.25_android_sweb.apk) and found that it still does not use HTTPS to transmit all EncryptWall requests, including the ones that we identified in our disclosure. Is version 11.25 still the version of the Android app that should contain these fixes, or will it be in a future release?"*
July 20 2023	We found that Sogou Input Method developers had released version 11.26 of the Android app. We found that this version addressed all of the issues that we reported.
July 21 2023	The TSRC portal prompted the following message: *"The vulnerability has been repaired, please review and check if it still exists. If it has been repaired, please click 'Repaired'; if it has not been repaired, please click 'Unrepaired'."* We clicked "Repaired".
July 22 2023	We received the following response via the TSRC portal: "Thank you for your feedback. We'll look into it internally."
July 24 2023	We received the following response via the TSRC portal: *"Thank you very much for your feedback, our latest repaired version is 11.26 (SogouInput_11.26_android_sweb.apk, you can download it from our official website: https://shurufa.sogou.com/). If you have any other questions, please let us know.thanks."*
July 27 2023	We received the attached email at the [redacted]@utoronto.ca email address. In the email, Sogou Input Method developers provide us with the versions containing the fixes and inquire about "the exact time, website and specific content" of our public disclosure.

Date	Contact
July 27 2023	We sent from [redacted]@utoronto.ca the following response: *"We can confirm that you have fixed the vulnerabilities that we reported. We will not publicly disclose the vulnerabilities until after July 31, 2023. We will publish details regarding the security vulnerabilities in a report that will be available on our website: https://citizenlab.ca/ ."*
July 29 2023	We received the attached email at the [redacted]@utoronto.ca email address. In the email, Sogou Input Method state their commitment to privacy and security, explain their original motivation for the EncryptWall system, and remind us of their speedy resolution of the reported vulnerabilities.

Table 3: Vulnerability disclosure timeline.

On July 4, 2023, we evaluated the partial mitigation which the Sogou Input Method developers stated they applied on June 30, 2023, in which, in the case of error, Sogou servers always return the same HTTP status code — 400 — instead of 400 or 500 depending on whether there is a padding error or some higher level application layer, respectively. While this mitigated our attack on the Windows version of Sogou Input Method as well as our attack on the "U", "G", and "P" fields on the Android version, our attack on Android's "S", "E", and "F" fields still worked since it relied on distinguishing between HTTP status codes 400 and 200, 200 being a success code and not an error code, and the mitigation only modified the servers to unconditionally return status code 400 in the case of an error.

Platform	Fixed Version
Windows	13.7
Android	11.26
iOS	11.25

Table 4: Fixed versions of Sogou Input Method.

In the Sogou Input Method developers' July 4 correspondence, they stated that version 13.7 of the Windows version of the app and version 11.25 of the Android and iOS versions of the app would address the issues that we reported. On July 18, 2023, we found that these versions of the app had been released. Note that these updates were released ahead of the July 31 deadline which we imposed. Analyzing the updated Windows version, we found that all EncryptWall traffic was encrypted using the TLS implementation provided by the operating system's WinHTTP service, satisfyingly fixing the vulnerabilities we reported in the Windows version. Recall that we were unaware of any way to exploit the issue which we discovered in the iOS version of the app. Nevertheless, we found via static analysis that the updated version of the iOS version addressed the issue that we reported. Despite version 11.25 being originally identified by the Tencent developers as resolving the vulnerabilities we reported, we found that on July 20, 2023, the Sogou Input Method developers released version 11.26 of the Android app and that this version used TLS to

encrypt all EncryptWall traffic, satisfyingly fixing the vulnerabilities we reported in the Android version. Thus, by July 20, 2023, all issues that we reported were fixed (see Table 4 for a summary of fixed versions).

Our difficulties receiving Tencent's email response to our disclosure highlight unexpected challenges in disclosing vulnerabilities to companies in certain jurisdictions. After disclosing the vulnerabilities to Tencent, we measured that our email domain (citizenlab.ca) is blocked in China. Specifically, we found that China's national firewall injected anomalous DNS replies in response to queries for this domain, including MX record lookups. The injected DNS replies contain an A record with a seemingly arbitrary IP address, even when the lookup was for an MX record, not an A record. When a client making an A record lookup receives one of these injected responses, it will erroneously use the bogus IP address in the injected response. However, for MX records, these injected responses are likely to be interpreted as errors by DNS clients due to receiving an A record in response to an MX lookup, and a DNS client's MX lookup for an injected domain is likely to simply fail rather than erroneously using a bogus record as in the case of A lookups. While this injection behavior may have been intended to block Chinese users from accessing our website, it also hampers the ability for users in China to email us, even if such an email has been solicited.

We cannot be certain that China's blocking of our domain is why Tencent's email was not delivered to an email server on our domain, but we received some late evidence that further strengthened this hypothesis. The July 27 email that we received at [redacted]@utoronto.ca was also addressed to disclosure@citizenlab.ca. The disclosure@citizenlab.ca address ultimately received the email on July 28, just over 24 hours later. By inspecting the email's headers, we found that the delivery of the email stalled between one of Tencent's mail servers and Google's MX servers. As Google is our mail provider in the citizenlab.ca MX records, this finding strengthens the hypothesis that Tencent's mail servers were struggling to look up our domain's MX records. The email may have eventually been delivered over 24 hours later due to an intermittent failure in China's firewall or due to packet loss dropping the firewall's injected DNS responses, allowing the MX lookup on our domain to finally succeed. Therefore, we have chosen to communicate all future disclosures from a different domain that, to our best knowledge, is not blocked in any country, to ensure that we do not fail to receive crucial communication during a coordinated disclosure. Simultaneously, we ask firewall operators to consider how blocking domains may have unintended consequences such as contributing to continued vulnerabilities in the software developed by those behind their firewalls who may be hampered in participating in important dialog during coordinated disclosures.

Limitations

In this report we detail vulnerabilities in Sogou's EncryptWall encryption system as used in Sogou Input Method. However, in this work we did not perform a full audit of Sogou Input Method or make any attempt to exhaustively find every security vulnerability in the software. Our report concerns a single set of related vulnerabilities that we discovered, and the absence of our reporting of other vulnerabilities should not be considered evidence of their absence.

Discussion

Over the last eight years we have dedicated immense effort analyzing, documenting, and responsibly disclosing vulnerabilities concerning the insecure transmission of sensitive data in Chinese-developed apps. While we have had some success in coordinating with developers to resolve these issues, the ecosystem remains problematic, as here we are, again, reporting on how an unimaginably popular Chinese-developed app fails to adopt even simple best practices to secure the sensitive data which it transmits. In the present case, Sogou Input Method, an app with over 450 million users, failed to properly secure the transmission of sensitive data, including the very keypresses which its users were typing, allowing such data to be recovered by any network eavesdropper. This vulnerability could have been easily avoided by, instead of using "homebrew" cryptography, adopting TLS, a common and mature cryptographic protocol with ubiquitous availability and up-to-date support. While no cryptographic protocol is perfect, TLS implementations had already ameliorated vulnerability to CBC padding oracle attacks in 2003, two decades prior to the time of this writing. We have come to believe that coordinated security disclosures are sorely inadequate to protect the data of users transmitted by Chinese apps. We believe that holistic change in the software development ecosystem is required to resolve these systemic issues.

Even with the reported vulnerabilities now resolved, the Sogou app relies on transmitting typed content to Sogou's servers as part of its ordinary functionality. Keystrokes coming from users anywhere in the world are transmitted to servers in mainland China, which are operating under the legal jurisdiction of the Chinese government. High risk users of Sogou should be cautious, as typed material could include sensitive or personal information. The attacks outlined in this report demonstrate how network eavesdroppers can decipher such data in transit. However, even with the vulnerabilities resolved, such data will still be accessible by Sogou's operators and by anyone with whom they share the data.