Security Overview of QR Codes

Kevin Peng, Harry Sanabria, Derek Wu, Charlotte Zhu
Massachusetts Institute of Technology
6.857 Computer and Network Security

Table of Contents

ABSTRACT

Quick Response codes are 2-dimensional barcodes that visually encode bits of information represented as black square dots placed on a white square grid. Because QR codes are a way to transmit information, this paper examines QR codes for security flaws that could pose a danger to the average user and presents potential fixes to these security flaws. We propose new security protocols for QR code creation that involve encoding bits that are encrypted or digitally signed via traditional security protocols. We then attempt to compare the efficacy of QR code phishing attempts to that of short URL phishing attempts using a social experiment. Finally, we perform a bug search on two existing QR code applications, the popular open-source QR code scanner ZXing and an open-source QR code payments library called SPayD.

We successfully applied security standards to QR codes and produced various forms of secure QR codes. However, each comes at a cost of either space or computation time, and therefore should be used only when needed. The social experiment was inconclusive because not enough people followed either the QR code or the shortened URL. Lastly, the bug search showed that both scanners had vulnerabilities involving code injection, unauthorized actions, and information leaking.

INTRODUCTION

Quick Response codes, commonly abbreviated as QR codes, started out as an extension of the standard UPC barcode commonly used in retail and production. Unlike a 1-D barcode, a QR code is a 2-D matrix code that conveys information by the arrangement of its dark and light elements in columns and rows [1]. The data in a QR code can be accessed by taking a picture of the QR code and processing it with a QR code reader.

The QR code itself is simply an array of bits to be identified by a scanner. Bits are reserved for the scanner to be able to identify and orient the image, as well as for version and format information (Figure 1). The remaining bits are used to encode the message, and the specific amount of available space leftover is dependent on the version of the QR code, which indicates the number of bits per row/column, and the level of error correction, which introduces redundancy. The most information dense QR codes used today can store just under 3,000 bytes of raw data [2].

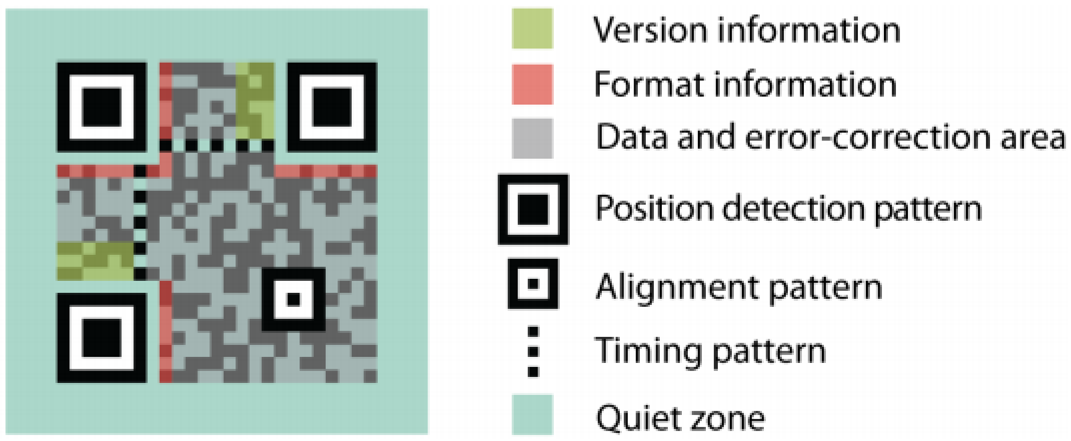


Figure 1. QR code structure [2]

Capable of encoding the same amount of data in about one-tenth the space of traditional 1-D barcodes, QR codes present a much more space-efficient way of presenting scannable data. Initially invented for use in automotive factories in Japan, QR codes were used to programmatically identify and track car parts quickly in order to speed up production. However, they soon began to see more widespread use as convenient methods of sharing and transmitting data. Popular commercial uses of QR codes now include URL redirection, payment information exchange, and electronic flight tickets [3].

With the increase in usage of QR codes in the general public, it is necessary to ensure that the data conveyed through the QR code is not harmful to the user. There are currently two major attack vectors for potential vulnerabilities: attacks on human interactions and automated attacks [2].

Attacks on human interactions rely on the fact that humans by themselves are unable to interpret what information is encoded in QR codes, and thus rely on QR code readers to decode the information. Since the information in the QR code is completely obfuscated, it is possible to trick and attack users via phishing, pharming, and other social engineering attacks by putting up fake QR codes. It is also possible to attack users by manipulating and exploiting existing QR code readers that users use via command injection or traditional buffer overflows [2].

Automated attacks often result from the assumption that the encoded information in QR codes is sanitized. However, it is known that QR codes themselves can easily be manipulated in order to change encoded information, potentially producing attacks on backend software. Without QR code input sanitation, it is possible to produce attacks such as SQL injection, command injection, and fraud.

In this paper, we propose potential solutions to these QR code attacks. We introduce security protocols for QR codes that allow the QR code creator to protect the information stored in the QR code by encrypting it and allow the user to verify that the message was not previously tampered with. We also attempt to compare the effectiveness of a QR code phishing attempt to that of a short URL phishing attempt in order to see whether a user is more likely to inherently trust a QR code or not. Finally, we search through existing QR code software to find bugs because user input is left unsanitized.

QR CODE SECURITY PROTOCOLS

Background

A QR code is simply a string of bits visually encoded as black and white squares on a grid. In most uses of QR codes where the intent is to have maximum accessibility, such as the common URL redirection, there are no security standards. Therefore, anybody can read or write QR code messages with impunity. However, certain applications may require restricted access or verification of QR codes, and thus there is a need to design QR codes that meet various security specifications.

In this section of the paper, we shall discuss the implementation of a few security standards for QR codes, their costs, and some examples of use cases for each. The first is a standard that allows the creator of a QR code to encrypt the message for a specific reader. The second is a standard that allows the reader to verify the origin of the QR code. Lastly, we shall discuss a few other attempts we made at security protocols, where they fall short, and what future work could be made in those areas.

Encryption

The first security standard for QR codes is Encrypted QR codes, or EQRs. We will propose two kinds of EQRs: Symmetric EQRs (SEQR) and Public Key EQRs (PKEQR). In SEQRs we use a symmetric encryption scheme where both the reader and the writer of the EQR share a secret key. The encryption scheme is extremely straightforward: encrypt the bits of the message using AES block cipher with the shared secret key. In PKEQRs we use the RSA public key encryption scheme combined with AES, using a public RSA key to encrypt the AES key and including the encrypted key in the message. The only thing to note with these two methods is that the error correction bits should correct errors on the *encrypted* message, not the message itself, in order to avoid leaking information of the original message.

This is not the first time SEQRs have been done; Google did an experiment involving SEQRs being used for login in 2011 [4]. Their method is the same as our first use case, namely using a QR code reader on a "verified" mobile device to verify another device. The idea is as follows. Assume a user has a mobile device on which they are logged in, which shares some secret key (e.g. a password) with Google, or any other service requiring login credentials. The user wishes to login on a new device, one on which he or she has not logged in before. Instead of typing in their password on the new device, which can put them at risk to keyloggers if they are doing this on a public computer, they are presented with an SEQR which can be read using only the secret key. The verified mobile device can then decrypt this SEQR, which contains a URL that, when hit by a request from the mobile device, logs in the user on the new device.

This URL is uniquely generated for the new device and expires within a few minutes. In this way, the user does not need to remember their password or type it in on untrusted devices.

PKEQRs have not, as far as we have seen, been put into practice. This is probably because they have few applications, as commercial bar codes are mostly intended to get some data to a lot of people easily, not to encrypt data for only a single person. There are, however, non-commercial applications which can be found by going back to the QR code's roots as a identifier for parts in manufacturing plants. Some companies, such as Apple or Google, wish to keep their operations secret. When they receive a shipment of parts for a new, secret product, they are able to see if the packaging has been tampered with and inspected, but cannot stop intermediaries such as the shipping company to inspect the labeling on the shipment. Therefore, they are unable to keep their supply of parts a secret. The solution is to publish a public key, and have the part manufacturer label the shipment using PKEQRs, thereby hiding the information from third parties while still allowing the recipient to read the label efficiently, with their same QR code reading technology.

As with any layer of security, these modifications to the QR code standard come at a cost. The encoded messages with SEQRs are of the same length of the original messages (as long as the message does not need to be padded), and therefore the only cost involved is the extra computation time when creating or reading the codes. Most of the uses of SEQRs don't need to be fast to write, but users usually desire the readers to produce a result quickly. Therefore, by using cipher modes like CBC where decryption is parallelizable, the decryption time can be minimized. Since these messages are usually only URLs a couple of bytes long, the decryption time should be negligible in most cases.

Codes encrypted with PKEQRs, however, come at a greater cost, as encrypting messages this way adds additional data, namely the bits of the encrypted secret key, increasing the length of the sent message. Message length is extremely important in QR codes; longer messages require denser QR codes, which can make readers more error-prone. Even the smallest AES key size is 128 bits, or 16 QR code blocks. Therefore, a message that could be encrypted with a version 1 QR code (the most common QR code size, 21x21) must be encrypted with a version 3 PKQR code (29x29) [2], even at the lowest level of error correction! Therefore, in order to achieve the same level of readability, the new PKQR code would need to have nearly twice the area of the unencrypted QR code. This cost is infeasible for QR codes printed in magazines or newspapers where the publisher of the code is charged per square inch. Therefore, the use of the PKQR code must be limited to situations in which size is not a factor, such as the labelling of shipments of top secret parts.

Signing

The next security standard we have devised is Signed QR codes, or SQRs. The purpose of this encoding is to allow the reader to verify the source of the SQR before any action is performed. If the verified source is trusted, the user can proceed to open the URL or perform any other action the QR code initiates without fear of a security breach. The SQR standard requires more modification than the previous encryption methods, as the code must contain the message, the signature, and a way to identify the public key of the signer. The new standard is described in the image below:

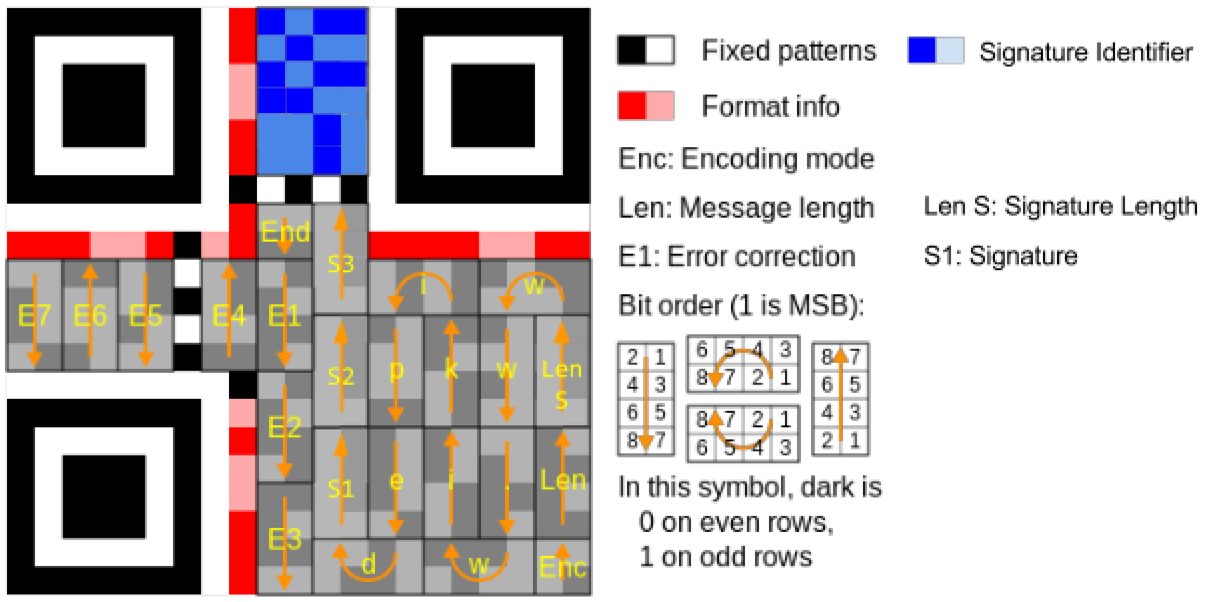


Figure 2. SQR code standard for version 1.

We have made several changes. First, there are now double the number of blocks devoted to relaying the message length, in order to also provide the signature length. Second, the signature bits are appended to the end of the message, and the error correction bits correct these signature bits as well. (Note that the length of the signature in this image is completely unrealistic, and only serves a demonstrative purpose.) Lastly, regardless of the QR code size, 24 bits adjacent to the existing formatting bits are devoted to a unique identifier for the signer, which can be used to retrieve the public key used to verify the signature. 24 bits allows for the unique identification of over 16 million signers; if this number is insufficient in the future the design can be scaled up and additional identification bits added.

The scheme depends on a trusted third party that can be accessed at a known URL which can provide the public keys used to verify the SQRs. Like any certificate authority, this TTP assumes the burden of performing due diligence on those that it provides public keys for. A party wishing to produce an SQR needs to first consult the TTP and, after it is verified, publishing a RSA-PSS public key on the trusted server. Then, using their RSA-PSS secret key, they sign the message they wish to send, append the signature to the message, and produce the QR code, adding the lengths of the message and signature as well as their unique identifier provided by the TTP. The reader for this QR code then requests the public key from the TTP using their identifier, and uses it to verify the message with traditional the RSA-PSS verification algorithm.

This cryptosystem clearly has many practical applications. Companies, clubs, or groups of any kind can use the scheme to send trusted QR codes to their members. For example, a company can impose the rule that all QR codes in their offices must be SQRs signed by them, thereby enabling them to filter all the QR codes and avoiding problems caused by arbitrary URLs being opened on the company network. The user also has a lot to gain, as URLs and other data read from promotional posters or other public sources can be certified before being read, resulting in less phishing scams.

As with SEQRs and PKEQRs, there are costs associated with SQRs. There is a significant overhead in terms of space: from 1 byte for the bits specifying the length of the signature, 3 bytes for the signature identifier, and usually 128 bytes for the signature. Therefore, as many as 261 additional bytes are needed beyond the message length, meaning a message that would use a version 1 QR would need a version 7 SQR code, which is over 4 times as large [2]. The size difference is large enough that it is probably preferable to not increase the size and sacrifice some readability. This is reasonable if using the highest level of error correction, as any loss in readability should be capable of being corrected in this case.

Other Attempts

We dedicated quite a bit of time thinking about alternative security protocols for QR codes that utilize the nature of the QR code itself instead of performing encryption operations on the bits. The schemes above are more viable simply because the algorithms involved are known, proven to work, and have been optimized over many years. However, our other efforts are, still worth noting in case future researchers wish to investigate further.

One attempt made was for a "proof of work" QR code. In this scheme, which takes its idea from Bitcoin, a QR code contains a message, a series of random bits, and the hash of the concatenation of those two. The catch is that the bits containing the hash are located in regions usually reserved for fixed orientation patterns, namely the squares at the corners of the QR code. If the hash does not form the proper orientation patterns, the code will be unreadable. Much like how Bitcoin miners must compute for a long time to produce a hash ending in multiple 0's, creators of the "proof of work" QR code must compute for a long time to produce a hash which is both valid for the encoded message and contains the correct orientation bits.

This is more space efficient than forcing the creator to produce a hash ending in 0's and having them use the message space to contain the hash, as our method frees up more room for the message by moving much of the hash into normally reserved bits. We could not, however, come up with a scenario in which such a scheme would be useful.

The second attempt we made was to take advantage of the optical nature of the QR code to produce a more space-efficient security encoding. Since QR codes are always read by an optical sensor, if given a high fidelity sensor, there is no reason to restrict the code to black or white binary squares. Shades of gray (or other colors) could conceivably be used, allowing each square to contain more than one bit of data. For example, assume two parties share a secret key and wish to share a secret message via QR code. The secret key could be used to generate a series of chromatic ranges, with each ranged assigned either 0 or 1, such that every color identifiable to the optical sensor can be identified as a 0 or a 1 (if the reader also knows the secret key). The writer of the QR code then encrypts the message by randomly selecting an appropriate color or shade for each bit. The reader, knowing the color assignments, just reads each color in and converts it to its assigned bit value. Any third party reading the QR code cannot distinguish which colors are a 1 and which are a 0.

We did not follow through with this scheme for two reasons. First, it is difficult to get a value for the number of distinguishable colors in optical sensors, and the variation between sensors on different reader devices is enormous. Second, providing a cryptographically secure method for producing the color to bit assignments is troublesome, and there is no reason to believe that this method could be more secure than known algorithms such as RSA or AES. It could be interesting, however, for those interested in the topic to delve deeper and come up with new ways to take advantage of the optical nature of QR codes.

QR CODE SOCIAL EXPERIMENT

Background

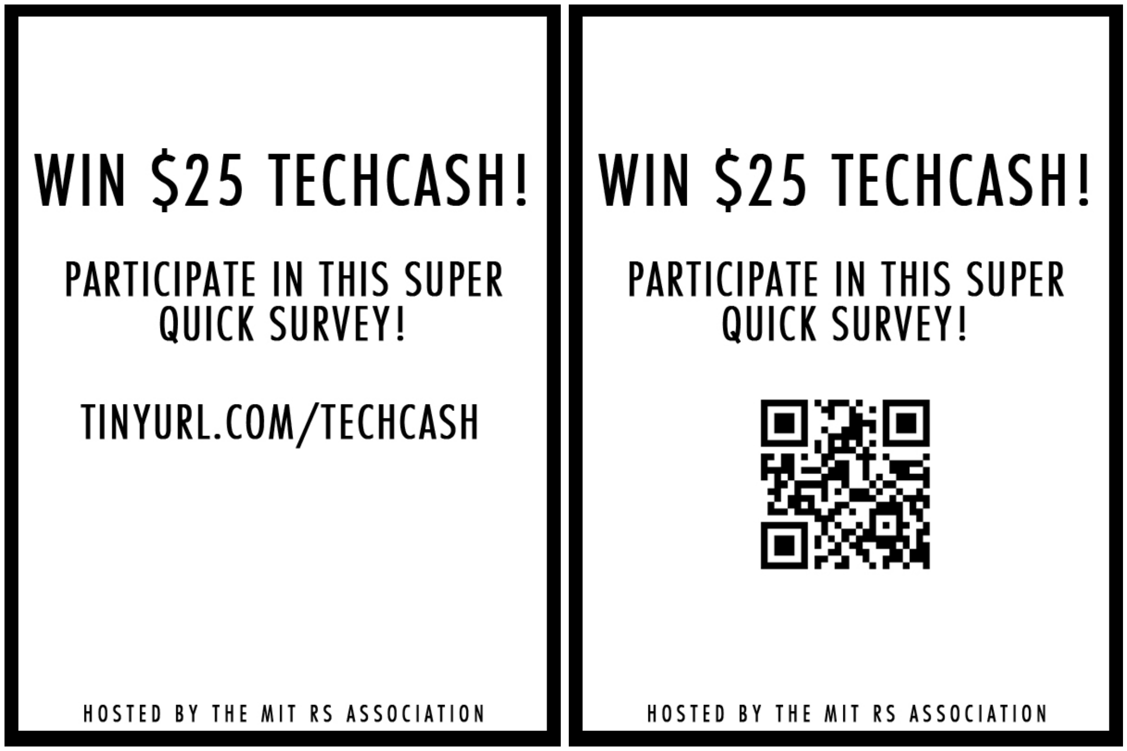
Because QR codes are designed to be read by a machine, humans cannot easily read the encoded message with the naked eye. Thus, QR codes can often be used to hide their contents from the user. This makes them potent as a vector to redirect unsuspecting users to malicious websites, among other forms of attacks.

In April 2012, GreenSQL CTO David Maman performed an experiment at the Infosec UK Security Conference to observe how many people would scan a QR code that could potentially redirect users to a malicious website [5]. He created a poster with a QR code and a logo of a real security company that advertised to "just scan to win an iPad". Scanning the QR code redirected users to a webpage that contained just a harmless smiley face, but in theory, could have had more malicious intentions. Maman found that over three days, there were a total of 455 scans of the QR code: 142 via iPhone, 211 via Android, 61 via Blackberrys, and 41 via unknown devices. Despite the security conference setting, Maman has demonstrate that a significant number of users fall prey to this attack.

Building on Maman's results, we were interested to see how effective a malicious QR code redirection is compared to a malicious shortened URL misdirection. Both methods obscure the final malicious landing page, with QR codes using visual bits to encode the URL and shortened URLs using a different URL. However, the methods to reach the malicious landing page are different: a QR code requires a QR code scanner, a shortened URL requires the user to enter in that URL into a browser. We conducted this experiment to gain insight into which attack vector was more effective.

Methodology

We created two posters, one containing a TinyURL^{TM} and one containing a QR code (Figure 3). The two posters advertised a chance to win a free 25 dollars in TechCASH, which can be used to purchase many goods and services across the MIT campus. Like the Maman poster, a bogus association (MIT RS Association) was mentioned on the poster to create more credibility.


WIN $25 TECHCASH!

PARTICIPATE IN THIS SUPER QUICK SURVEY!

TINYURL.COM/TECHCASH

HOSTED BY THE MIT RS ASSOCIATION

WIN $25 TECHCASH!

PARTICIPATE IN THIS SUPER QUICK SURVEY!

HOSTED BY THE MIT RS ASSOCIATION

Figure 3: TinyURL^{TM} and QR code posters

The TinyURL^{TM} and QR code redirected to different URL's that visually looked identical, and displayed an apologetic message explaining that the 25 dollar TechCASH poster was a social experiment, and the only user data collected was the number of hits for each of the two posters. (A PHP script tallied the number of users that accessed the landing page via TinyURL^{TM} and QR code.)

The posters themselves were placed on the MIT campus at various high-traffic locations: Student Center (W20), Building 56, Building 66, and the Stata Center (32) (Figure 4). The TinyURL^{TM} poster was sufficiently distanced from its corresponding QR code poster. To remain consistent with Maman's duration of the experiment, the number of accesses to the landing page was monitored for a period of 3 days.

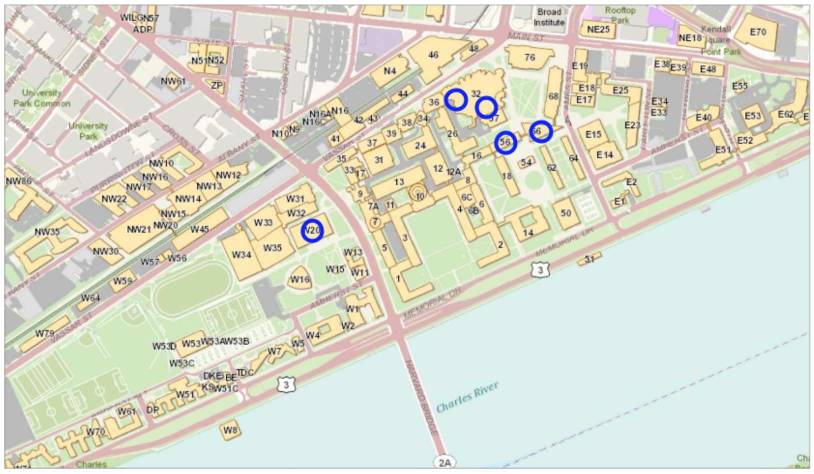


Figure 4: Locations of Posters on MIT Campus:
Each blue circle represents a corresponding pair of
TinyURL™ and QR code posters, placed sufficiently far apart.

Results

After three days, 5 users accessed our "malicious" landing page via TinyURL™ and 3 users accessed the landing page via the QR code. Because the numbers were so low, no significant conclusion can be made to relate the two different vectors of attack.

Discussion

Although our data is not significant enough to draw any conclusions, we can postulate a few reasons why our numbers were so low. Our methodology did not allow us to collect the number of individuals that noticed the poster but did not elect to follow through with the TinyURL™ or QR code. Thus, we have no sense of the proportion of users that fell prey to the posters. Our low numbers could be due to low number of users noticing the posters or due to low "participation rate".

Assuming that our numbers are due to a low participation rate, there are several possible reasons why individuals did not choose to follow through the link. It is possible that the posters did not seem credible or enticing enough to persuade the user to use the TinyURL™ or QR code. It is also possible that the MIT community in general is more tech savvy and therefore more likely to avoid accessing websites via TinyURL™ or a QR code.

There is also the possibility that more people would have scanned the QR code, if only more people had QR code scanners previously installed on their devices.

Ultimately, the purpose of this experiment was to attempt to determine which attack vector was more effective in luring unsuspecting individuals. Because we collected such low numbers, we cannot make a conclusion that one vector is more effective than the other.

For future experiments, researchers should devise a way to collect the number of people who noticed the posters but did not follow through to the website. One possible way to do this is to gain permission to have a camera recording the number of people that actually see the posters. Thus, researchers can acquire the proportion of individuals that fell victim to such attacks. Furthermore, we could also collect more information of the demographic of the people who do fall prey, and observe which groups of individuals are more susceptible. In order for this collected data to have any significance, more samples need to be collected. This can be remedied by extending the experiment duration or finding higher traffic areas to display the posters.

QR CODE READER BUG SEARCH

Background

Because QR codes are translated via a QR code reader, security vulnerabilities may exist in the QR code software. A malicious QR code can take advantage of a poorly coded scanner via some sort of code injection.

We performed a code review of two QR code readers: ZXing and an open-source Short Payment Descriptor (SPayD) reader. ZXing is a popular, generic QR code reader [6]. The SPayD code reader that we investigated provides a mechanism for exchanging payment information [7].

In most cases, readers go through three steps. First, they read in the raw bits in the correct order from the code, applying error correction as needed. Second, they (attempt to) interpret the contents, and put the raw data into a properly formatted object. Lastly, the reader performs an action related to the data, such as opening a URL. Usually the last step requires user confirmation before the action is performed.

Methodology

To search for vulnerabilities in the QR code readers, we focused on the critical points in this three step process, namely the second and third steps. In general, there are three common kinds of vulnerabilities possible, shown below. The first of these vulnerabilities can occur on the second step of the reading process, whereas the other two can occur on the third step.

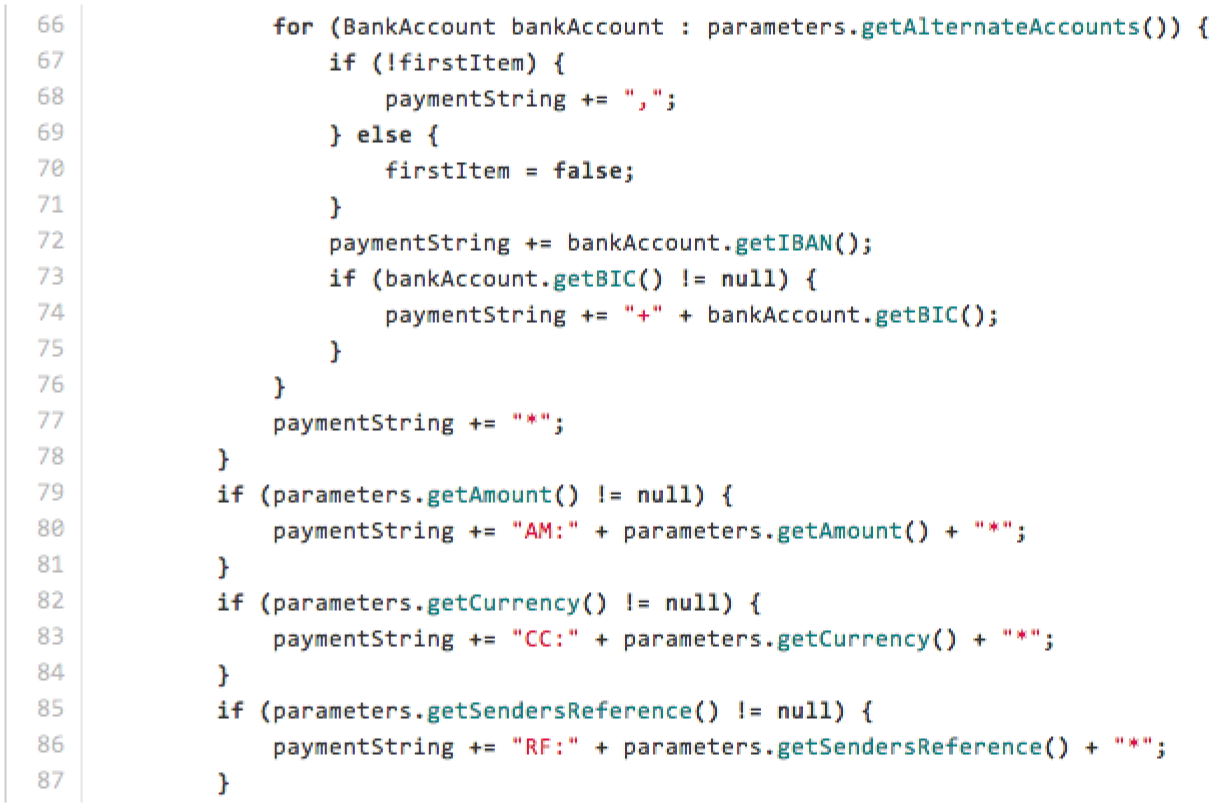
1. Code Injection: Much like SQL injections that occur when SQL queries are made with user input text inserted into the query string, QR code readers are subject to data injection into their structured objects when they attempt to interpret the data of a QR code. If the contents of a QR code are inserted into a data structure with no sanitation done beforehand, a malicious party can create a QR code that injects arbitrary strings into a user's data structures, potentially causing harm to the user. We will see an example of this in the open source SPayD reader.

2. Unauthorized actions: If a reader incorrectly interprets the contents of a QR code and attempts to perform an action based on its wrong interpretation, there can be resulting security vulnerabilities. Since the device is performing what it interprets as the correct action and asks the user permission to perform that, the actual action performed can go unnoticed to the user. For example, a reader can misinterpret some plaintext data as a URI and issue a

system call to “open” the URI. The user will be asked permission to open a URL, but since the URI is just plaintext the system might actually do something else with it and not do anything else with the browser at all. We will see an example of when this can occur in the ZXing reader.

3. Information leaking: If a reader is incapable of performing actions on the QR code data itself, it often delegates this responsibility to other applications. This can lead to problems in the common publisher-subscriber model, because if the reader is not careful, it can leak the contents of the QR code to sniffers on the device. Of the three common problems, this one is the least severe, and is very platform based. We will see an example of this with ZXing where careless use of the Android publisher-subscriber model (Intents) leaks information on the device.

SPayD



```
66              for (BankAccount bankAccount : parameters.getAlternateAccounts()) {
67                  if (!firstItem) {
68                      paymentString += ",";
69                  } else {
70                      firstItem = false;
71                  }
72                  paymentString += bankAccount.getIBAN();
73                  if (bankAccount.getBIC() != null) {
74                      paymentString += "+" + bankAccount.getBIC();
75                  }
76              }
77              paymentString += "*";
78          }
79          if (parameters.getAmount() != null) {
80              paymentString += "AM:" + parameters.getAmount() + "*";
81          }
82          if (parameters.getCurrency() != null) {
83              paymentString += "CC:" + parameters.getCurrency() + "*";
84          }
85          if (parameters.getSendersReference() != null) {
86              paymentString += "RF:" + parameters.getSendersReference() + "*";
87          }
```

Figure 5. In line 83, lack of sanitization in SpayD code reader allows the user to overwrite the AM field that was already set in line 80.

The SPayD reader that we analyzed does not have any form of sanitation for the inputs that are provided. This, in essence, allows the user to enter variables as inputs and inject code to modify values. In the example shown in figure 5, the parameters variable is inputted by the user. Note that the asterisk symbol is a delimiter for SPayDs, and does not denote multiplication.

In line 80, the payment amount is set by retrieving metadata from the parameters variable. For example, let's assume the user originally set the amount to be 5. If the user also passes in a currency parameter equal to "USD * AM: 1,000,000", the currency will be set equal to USD, as intended, but the user will also be able to overwrite the value of the amount to be 1,000,000 [7]. Here, code has been injected to modify fields in the payment string because there was no sanitation to prevent arbitrary values from being entered into the metadata of the parameters.

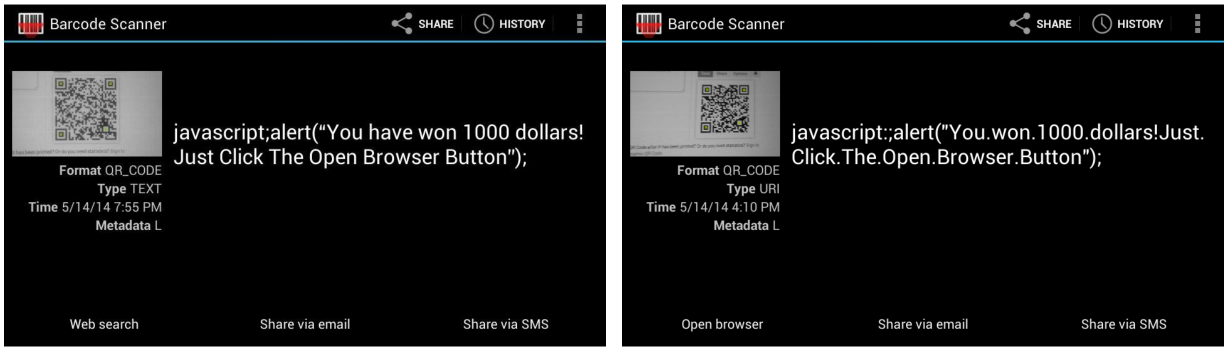
ZXing

ZXing does very little checking of the URI type that is provided through the QR code. Any exploit that can be provided through a browser can be directly passed through the QR code as a result. This, in the context of a mobile application, can lead to launching arbitrary phone applications or injecting Javascript in browsers ZXing attempts to filter out certain malicious attacks via regular expressions by requiring the URI to have either:

1. A period followed by an extension of at least length two
2. A transport protocol of length at least two followed by a colon in the expression

ZXing also requires that the URI have no whitespace. If these conditions are not upheld, it will assume that the data presented in the QR code is plaintext rather than a URI. After After passing this stage, ZXing takes the given URI and only checks if the URI begins with http:// or https://. It assumes that all other transport protocols are legitimate and performs no sanitation on them.

This mechanism bans simple attacks such as javascript;alert("You have won 1000 dollars! Just Click The Open Browser Button");, but by simply making a few changes to the code, ZXing will think that the Javascript call is a normal URI, and expose a vulnerability to any browser that doesn't also have its own defense mechanisms against URI attacks (Figure 6). Note additionally that ZXing doesn't immediately open a URI, but rather gives the user the option of looking at the exact text before proceeding. This allows provides experienced users with some security when receiving links from QR codes, but nevertheless, presents a vulnerability to novice users.


Barcode Scanner

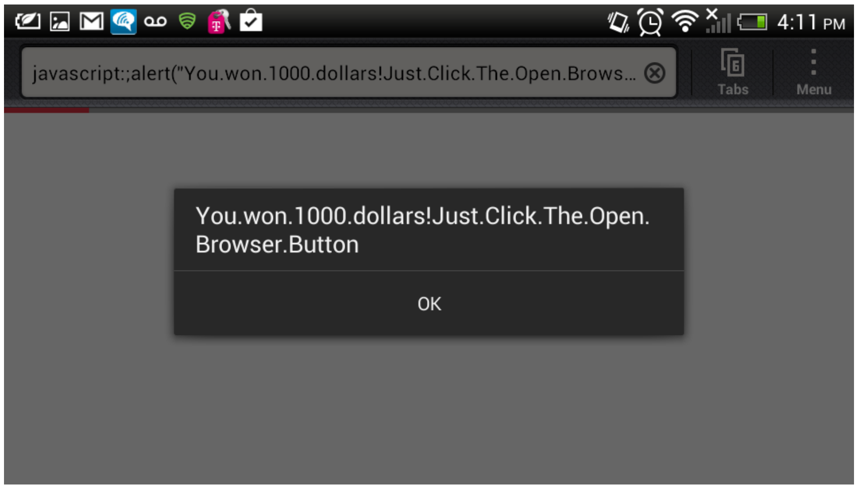
javascript;alert(“You have won 1000 dollars! Just Click The Open Browser Button”);

Format QR_CODE
Type TEXT
Time 5/14/14 7:55 PM
Metadata L

javascript:;alert("You.won.1000.dollars!Just.Click.The.Open.Browser.Button");

Format QR_CODE
Type URI
Time 5/14/14 4:10 PM
Metadata L

Figure 6: The left image depicts ZXing’s attempt to filter out simple Javascript attacks, but with some slight modification, as seen on the right, vulnerabilities are still present.


javascript:;alert("You.won.1000.dollars!Just.Click.The.Open.Brows…

You.won.1000.dollars!Just.Click.The.Open.Browser.Button

Figure 7: Opening the URI from the right half of figure in the browser.

ZXing also is vulnerable to intent sniffing. When processing this URI, ZXing constructs a new implicit intent using a global action, Intent.ACTION_VIEW [6]. Other applications that filter for the same implicit intent can retrieve the same data from this intent. Moreover, Intent.ACTION_VIEW is a common action that is used by many Android applications. Since any other application that has an event listener for this action can obtain information about the URI that is read from the QR code, ZXing should first send an a global implicit intent requesting which application can open the URI without actually providing the data, and then send the data to the correct application with an explicit Intent. Other applications cannot eavesdrop on explicit intents, so this mechanism will prevent intent sniffing.

CONCLUSIONS

QR codes are by no means a secure standard for data encoding in their current state. There is, however, great room for improvement. New security standards such as Symmetric Encrypted QR codes, Public Key Encrypted QR codes, and Signed QR codes can be implemented within the existing QR standard. SEQRs allow for services such as login by scanning a QR code on a verified mobile device. PKEQRs would allow secure data to be distributed to a select group of people easily. SQRs allow users to check whether they trust the creators of QR codes before opening their potentially dangerous contents. All these methods come at some cost, either in space or time, and therefore should be used only when required, such as when using QR codes to share sensitive data within a company, and not in commercial use.

We attempted to gain more insight into whether QR codes or shortened URLs was a more effective attack vector to lure unsuspecting individuals to malicious websites. We conducted a social experiment with two sets posters, one with a TinyURL^{TM} and the other with a QR code, counting the number of individuals that assessed each. However, we found that the numbers observed were too low to make a significant conclusion. Future experiments could improve this by also measuring the number of individuals who viewed the posters but did not choose to navigate to the website, thereby giving the more useful proportion of people who accessed each instead of the raw number of people who accessed each.

We searched for implementation vulnerabilities in two QR code readers: Zebra Crossing (ZXing) and a Short Payment Descriptor (SPayD). We discovered three different kinds of vulnerabilities: code injection, unauthorized actions, and information leaking. Code injection allows the users to attack the system and other users by manipulating data through invalid inputs. SPayD is vulnerable to code injection because it does not filter for escape characters. Unauthorized actions occur when the reader misinterprets one form of data as another. This occurs in ZXing due to incorrect implementation of the regex that takes in a string and determines the type of data provided by the QR code. Information leaking also occurs in ZXing, since it uses implicit intents, meaning any application can acquire the data that ZXing broadcasts in the intent. We proposed that ZXing use a mixture of implicit and explicit intents to tackle this issue and prevent information leakage.

QR codes have various advantages as an information sharing tool. They allow fast and easy distribution of various forms of structured data and have many applications in manufacturing and business. As long as their users are aware of the vulnerabilities inherent in the standard, in their particular reader, and the ease of social manipulation attacks, QR codes can be used safely. The key is having awareness of the various attacks presented in this paper and implementing preventative measures by use of the standards also presented. If this is done, QR codes can be used globally as a secure and efficient standard for many different purposes.

REFERENCES

[1] *QR Code Essentials*. DENSO ADC. 2011. Web. 14 May 2014. <http://www.nacs.org/LinkClick.aspx?fileticket=D1FpVAvvJuo%3D&tabid=1426&mid=4802>.

[2] Kieseberg, Peter, M. Leithner, M. Mulazzani, L. Munroe, S. Schrittwieser, M. Sinha, and E. Weippl. (2010). QR Code Security. Proceedings of the 8th International Conference on Advances in Mobile Computing and Multi-media, MoMM '10. New York, NY, USA. ACM, pp 430-435.

[3] *History of QR Code*. DENSO WAVE Inc. Web. 14 May 2014. <http://www.qrcode.com/en/history/>.

[4] Dante D'Orazlo. *Google experiments with new QR-based secure login.* The Verge. 17 Jan 2012. Web. 14 May 2014. <http://www.theverge.com/2012/1/17/2714263/google-experiment-qr-code-secure-login-sesame>.

[5] William Jackson. *With QR codes, even security pros play the fool*. GCN. 12 Sept 2012. Web. 14 May 2014. <http://gcn.com/articles/2012/09/12/cybereye-qr-codes-security-pros-fooled.aspx>.

[6] *Official ZXing ("Zebra Crossing") Project Home*. Github repository. 14 May 2014. <https://github.com/zxing/zxing/>.

[7] *SmartPaymentDescriptor Generator*. Github repository. 14 May 2014. <https://github.com/spayd/spayd-java>.