Failures of Security APIs: A New Case

Abdalnaser Algwil and Jeff Yan

School of Computing and Communications, Lancaster University, UK
{a.algwil,jeff.yan}@lancaster.ac.uk

Abstract. We report novel API attacks on a Captcha web service, and discuss lessons that we have learned. In so doing, we expand the horizon of security APIs research by extending it to a new setting. We also show that system architecture analysis is useful both for identifying vulnerabilities in security APIs and for fixing them.

Keywords: API attacks, architecture analysis for security, Captcha, web security

1 Introduction

A security API is an Application Programming Interface that facilitates less trusted or even untrusted code to interact with a trusted computer. A classic example of security APIs are those that enable interactions between a banking computer (less trusted) and a cryptographic Hardware Security Module (trusted) attached to it. Security APIs differ from general programming APIs in that the former enforces a security policy, and typically the policy is about preventing some information flow while allowing tight and dynamic interactions between the untrusted computer and the trusted one. Even if the less trusted or untrusted code is malicious, ideally it should not be able to break the security of the trusted computer.

Security APIs started to get going as a research subject in 2000 with Ross Anderson's seminal paper [1]. A research community has formed, and interesting results emerged. However, most security API attacks published to date are about HSMs and cryptographic key management APIs, e.g. [2–4, 8, 9]. An exception is Robert Watson's work [12] that exploited concurrency vulnerabilities in system call wrappers to launch API attacks on operating systems.

In this paper, we report novel API attacks on CCaptcha (`http://crecaptcha.org`), an Internet service that is created to generate Captchas for any websites. A CCaptcha server (trusted) generates automated Turing tests using sensitive materials, and verifies each test result for the websites in the wild. The websites (untrusted) interact with the CCaptcha server via a set of APIs defined by the service provider. This gives us an opportunity for studying security APIs in a new setting, which expands the horizon of security APIs research.

We will show a number of API attacks. For example, one allows us to download all sensitive materials that the service uses for constructing Captchas; one allows us to launch an effective dictionary attack on the service; and the third

allows us to bypass this Captcha entirely. Two of the attacks defeat entirely or nearly so the purpose of deploying this service.

Our attacks work on both versions of the CCaptcha service, one released in 2010 and the other in the summer of 2014 (i.e. the latest version). It is clear that the service provider has invested some serious efforts in its design and implementation. But we argue that the designers did not seem to consider system architecture issues carefully, and this is a main reason that their security APIs fail. We discover our attacks by analysing the service's architecture, interactions among individual system components, as well as a limited amount of dynamic code available to a client.

This work also contributes to Captcha research. On the one hand, prior art did not examine Captcha security from the angle of security APIs, as conventional Captcha designs rarely provide the possibility to articulate or enforce a security policy.

On the other hand, text Captchas have been widely deployed, but many designs have been broken [5, 10, 11, 13, 14]. It is intellectually interesting and practically relevant to explore alternative designs, which are currently an active research topic. Initially disclosed in a USA patent application [7], CCaptcha is an interesting alternative scheme. The design is based on Chinese language, but due to its clever idea, it is universally usable, even to those who are illiterate in Chinese, which is quite counterintuitive. In a user study run by the inventors, foreigners without Chinese language knowledge achieved a high accuracy in solving this Captcha as native speakers did [6]. Our work is the first security analysis of CCaptcha.

2 CCaptcha

Concept. As shown in Fig.1, a CCaptcha challenge (or puzzle) is composed of 10 images. The bigger image at the top-left corner is a target Chinese character that can be decomposed into multiple elementary radicals. The nine smaller images on the right are candidate radicals. Some of them are real radicals from the target character, but others are faux ones. Thin lines and small dots in the background are not part of the character or radicals, but clutters. To pass a test, a user has to click typically three real radicals on the right panel. Selecting any faux radical will fail the test.

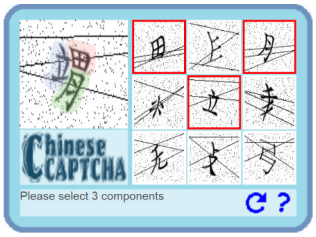

Fig. 1: CCaptcha: an example

Each time when a character or radical is used in a puzzle, it will undergo random distortions such as rotation, scaling and warping, and then random background clutters will be added. The inventors applied OCR software to recognize such distorted characters and radicals, but to no avail. Whether their recognition experiment is rigorous or not is beyond the scope of our paper. However, human users can easily solve such tests via pattern recognition, and they do not have to be literate in Chinese.

Random guess attacks. With random guessing, an attacker has about 1.19% chance (i.e., $1/C_3^9$) to break this scheme. If necessary, a user can be asked to solve 2 challenges or more in a row, and this will reduce the random guess success to $\sim 0.014\% = (1.19\%)^2$, or less. Alternatively, if the number of real radicals used, the number of all candidate radicals, or both, is slightly increased, it will reduce the random guess success, too. It will further decrease the success probability of random guess attacks by applying all these countermeasures together. Therefore, random guess attacks are not a serious issue for this Captcha.

System architecture. CCaptcha is implemented as a web service, and it provides both Captcha creation and validation services to web sites, where a CCaptcha challenge (or puzzle) can be easily embedded in each web page by installing a PHP library.

The service provider does not explicitly describe the system architecture for CCaptcha. By studying the limited amount of documents available online and by experimenting with the service, we reconstruct its system architecture diagram. Figure. 2 attempts to capture the workflow envisaged by the designers. Interactions among an end user, a web server and the CCaptcha service can be summarized as follows:

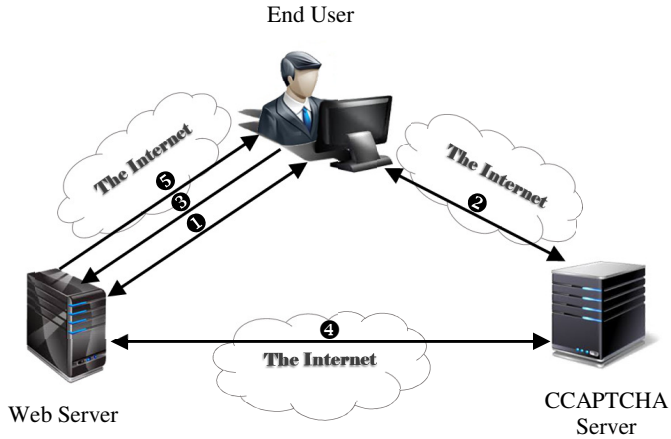

Fig. 2: CCaptcha web service: a reconstructed system architecture

1. A user requests to fetch a page from the web server, and the server sends her the page in which a CCaptcha script is embedded;

2. Driven by the script, her browser retrieves a puzzle from the CCaptcha server;
3. The user submits her puzzle solution to the web server;
4. The web server forwards this solution to the CCaptcha Server, and the CCaptcha Server verifies it and sends back a verification result;
5. Following the result, the web server accepts or denies the user's request.

3 API attacks on CCaptcha

Here we present API attacks that we have uncovered. They include information leakages, a dictionary attack and several oracle attacks. Some of the attacks are built upon each other, but some are standalone attacks on their own.

3.1 Information leakage

The CCaptcha service provides a PHP library `labcrecaptcha.php`, which wraps its APIs and provides a simple mechanism to embed a Ccaptcha puzzle on any web page. Our analysis starts with this library and follows up with leads exposed.

Paths. At the beginning of the library, as shown in Fig. 3, we find the location and path where the CCaptcha APIs are served:

`https://crecaptcha.org/crecaptcha/api/`

```
define("CRECAPTCHA_API_SERVER", "https://crecaptcha.org");
define("CRECAPTCHA_API_PATH", "/crecaptcha/api");
define("CRECAPTCHA_VERIFY_SERVER", "crecaptcha.org");
define("CRECAPTCHA_VERIFY_PATH", "/crecaptcha/api");
```

Fig. 3: Server and path information for the CCaptcha service

Puzzle generation. The library also reveals that a script `puzzle.php` is the interface responsible for Captcha generation. We note that once this script is called, a puzzle is created as an array of 10 integers (see Fig. 4).

```
var CrecaptchaConfiguration = {
server : 'https://www.crecaptcha.org/crecaptcha/api',
ishint : 3,
hardlevel : 1,
skin : 1,
puzzle : new Array('10089','18429','253','136','18591',
                   '20469','19901','17288','271','20461')};
document.write('<s' + 'cript type="text/javascript" src="' +
 CrecaptchaConfiguration.server+'/crecaptcha.js"></s'+'cript>');
```

Fig. 4: Captcha generation

Hidden field. When integrating the library with our test web page, we discover a hidden field `crecaptcha_puzzle`. This field can be obtained by using a PHP super-global method such as GET and POST during the Captcha verification process.

We note that this hidden field stores 10 integers separated by comma, for example,

```
crecaptcha_puzzle = "10089, 18429, 253, 136, 18591, 20469,
                     19901, 17288, 271, 20461"
```

We also note that these numbers are the same as those stored in the `puzzle` array.

It turns out that these numbers are distinct IDs of the images used to compose a CCaptcha puzzle, with the first number identifying the target character and the others identifying nine candidate radicals respectively. As illustrated in Fig. 5, a puzzle is generated by fetching 10 images via their IDs, and then composing the images accordingly.

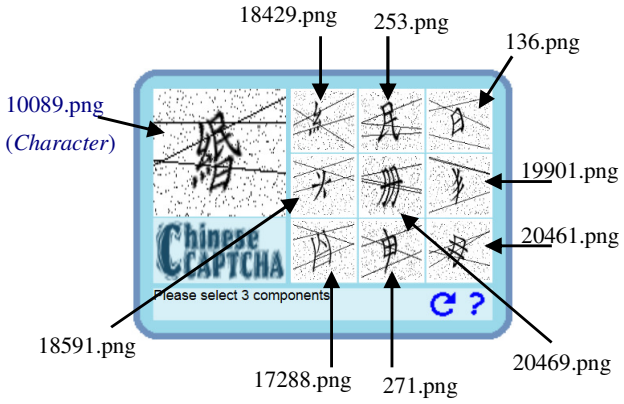

Fig. 5: Images used in a puzzle (1 character + 9 radicals) and their numeric IDs

Each time when a character or radical is used to compose a puzzle, it will undergo different distortions and be rendered as a different image. That is, the generator will not reuse any image, but create a different image for the same character or radical each time. However, the numeric ID always remains the same for all the different image versions rendered for the same character or radical. That is to say, there is a fixed one-to-one relationship between a character/radical and its image ID. This means that with the knowledge of an image's ID, we will know the image's content without applying any computer vision or pattern recognition algorithms.

Database leakage. A further analysis of `puzzle.php` source code (Fig. 4) reveals a JavaScript file `crecaptcha.js`. By examining this JS script (see Fig. 6), we discover that a PHP script `image.php` is responsible for retrieving image files from the CCaptcha server.

```
_create_image_tag: function (i, c, width, height) {
  output = '<img src="' + CrecaptchaConfiguration.server +
           '/image.php?c=' + c + '" alt="" style="width:'+
           width + ';height:' + height + ';border: 0px;" />';
  if (i > 0 && i < 10){
      output = '<a href="javascript:Crecaptcha.click_option('
               + i.toString() + ');">' + output + '</a>';
  }
  return output;
},
```

Fig. 6: Source code snippet from `crecaptcha.js`: `image.php` fetches images from the CCaptcha server

The script `image.php` enables us to fetch any radical and character by sending their numeric ID to the server. For example, as shown in Fig. 7, `https://crecaptcha.org/crecaptcha/api/image.php?c=1` returns us the character/radical of ID 1.

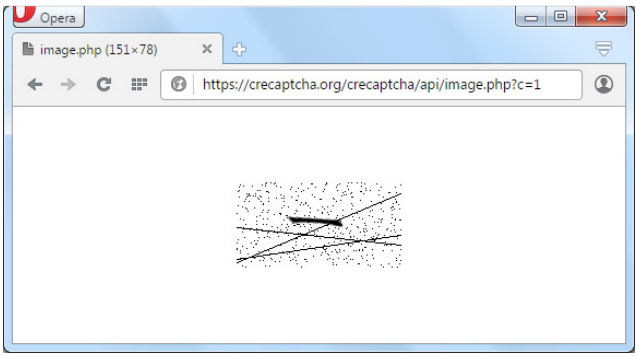

Fig. 7: Retrieving an image from the CCaptcha database

By sending a sequence of numbers starting from 1 to 72154, we manage to fetch 66,111 unique characters/radicals from the server – all the components used for CCaptcha generation. Note that a majority of numbers between 13060 and 19783 are not assigned to any images.

Note: all the scripts that we have analysed are available on the client side, and they are readily accessible merely via a browser.

3.2 A dictionary attack

The information leakages discussed above can be exploited to launch two attacks: 1) a machine learning attack that trains an automatic Captcha solving algorithm with the leaked database, and 2) an effective dictionary attack that solves the CCaptcha tests with a high success rate.

The first attack is beyond the scope of this paper, and we discuss only the dictionary attack here. The idea is as follows. We build a dictionary with entries being each character along with its valid radicals, and we store their image IDs in the dictionary. To solve a new puzzle, we simply pick up its target character's ID from the traffic, use that ID to look up the dictionary, and then identify valid radicals among nine candidates in the puzzle.

We have a simple but effective method for dictionary construction. By exploiting the ID leakage vulnerability, we know which character/radical is which, with the knowledge of their ID alone. If we analyse multiple puzzles generated for the same target character, we will know that if a radical occurs every time or most of the time in the puzzles, it will be a valid one with a high probability.

A general description of our dictionary construction is given as follows. First we launch a large number of requests to the server to collect θ different puzzles for each character. For a certain character, we can sort all candidate radicals by their occurrence frequency in the θ puzzles. If a radical occurs at least k times, we keep it in the dictionary as a real radical for the character. This process will be applied to all characters until our dictionary is stabilized. More details are given in Algorithm 1.

Next we discuss how to determine θ (the number of tables) and k (occurrence threshold) with the following analysis.

Parameter configurations. We know that a puzzle typically has 3 real radicals and 6 fake ones. It is reasonable to assume that they are randomly selected by the system from the real radical set (of a certain character) and the fake radical set, respectively. Also assume that a certain character has m real radicals (e.g. $m = 4$ for character 42328 and $m = 3$ for character 30646 in Fig. 10). Therefore, for θ trails, the probability that a real radical has been selected k times, denoted by P_1, follows the binomial distribution:

$$P_1 = \mathrm{C}_r^n \rho_1^k (1-\rho_1)^{\theta-k},$$

where $\rho_1 = 3/m$ is the probability of a certain real radical having been selected in a trial. Then the probability that a real radical has been selected less than k times is:

$$P_R = P(\#real < k) = \sum_{i=0}^{k-1} \mathrm{C}_i^\theta \rho_1^i (1-\rho_1)^{\theta-i} \tag{1}$$

Algorithm 1: Dictionary Construction

input : Puzzles from Chinese Characters Database in CCaptcha Server
output: Dictionary contains each Chinese character along with its correct radicals

DB_{CCs} : *Chinese Characters Database in CCaptcha Server*;
P: *Puzzle* (1 *character* + 9 $radicals_{\{3real+6faux\}}$);
P[*char*] : *Target Character ID* (*i.e. Big image*);
P[R_j] : *Radical ID* (*i.e. small Image*), *where* $j = 1, 2, 3, \ldots, 9$;
T_n : *Table to store puzzles* (P), *where* $n = 1, 2, 3, \ldots, \theta$;
θ : *Number of tables* (*e.g. 5*);
k : *Threshold of a radical occurrence*;
NoR : *Number of table Records*;
P_{T_n} : *Stored Puzzle in table n* (T_n);
C_{T_n} : *Target Character ID in stored puzzle in table n* (T_n);
R_{T_n} : *Radical IDs in stored puzzle in table n* (T_n);
All_Radicals : $\mathrm{P}_{T_1}[R] + \mathrm{P}_{T_2}[R] + \mathrm{P}_{T_3}[R] + \cdots + \mathrm{P}_{T_\theta}[R]$;
candidate_radicals : *IDs of correct radicals that compose the Target character*;

do
  P ← *send a request to CCaptcha Server and fetch a new puzzle*;
  for $n \leftarrow 1$ **to** θ **do**
    if (`IsFound`(P[*char*], T_n) ≠ *true*) **then**
      insert P *into* T_n ;
      break; // *for*
    end
  end
while ($NoR(\mathrm{T}_\theta) < NoR(\mathrm{DB}_{CCs})$);

Dictionary ← { };

for *all records in* T_1 **do**
  $\mathrm{C}_{T_1} \leftarrow \mathrm{P}_{T_1}[char]$;
  All_Radicals ← $\mathrm{P}_{T_1}[R]$;
  for $n \leftarrow 2$ **to** θ **do**
    $\mathrm{R}_{T_n} \leftarrow$ `GetRadicals` (T_n , C_{T_1});
    Merge R_{T_n} *with* All_Radicals;
  end
  count all of the matching values in All_Radicals;
  candidate_radicals ← All_Radicals[R] *where their frequency* > k;
  insert(C_{T_1}, candidate_radicals) *into* Dictionary;
end

Similarly, we can model the probably distribution of the fake radicals. Assume that there are M fake radicals that can be chosen from, where $M = z - m$, and $z = 1366$ (the total number of radicals used in the system). After θ trials, the probability that a fake radical has been selected at least k times is

$$P_F = P(\#fake \geq k) = 1 - \sum_{i=0}^{k-1} \mathrm{C}_i^{\theta} \rho_2^i (1 - \rho_2)^{\theta - i} \tag{2}$$

where $\rho_2 = 6/M$.

In building our dictionary, we repeat the random selecting process θ times, and then select radicals occurred at least k times as the real ones. Note that P_R and P_F are two important metrics determining the effectiveness of the dictionary. By definition, it is preferable to set parameters that yield small P_R and P_F.

We have empirical evidence that ρ_2 is relatively small but ρ_1 relatively large: by sending millions of requests to grab puzzles from the CCaptcha server, we establish that not all 66111 characters, but only 50313 of them, are used as a target character in a puzzle, where there are at least 3 valid radicals. We also establish that for all the 50313 characters, $m \in \{3, 4, 5, 6, 7\}$.

Therefore, according to Eq. 2, and Eq. 1, P_R and P_F can be very small if we choose a large θ and an appropriate k. However, a large θ would result in a high computational cost. To balance the trade-off between accuracy and efficiency, we decide to set $\theta = 5$ and $k = 3$, which as shown later will yield a reasonable accuracy with a low computational cost.

Given the distribution of character number with respect to m, we can also calculate $P_R^{Overall}$ and $P_F^{Overall}$, defined as follows:

$$P_R^{Overall} = \sum_{m=3}^{m=7} \frac{\#character(m)}{N} P_R(m),$$

$$P_F^{Overall} = \sum_{m=3}^{m=7} \frac{\#character(m)}{N} P_F(m),$$

where $N = \sum_3^7 \#character(m) = 50313$.

Table 1 shows various probabilities as calculated, and it clearly indicates that with $\theta = 5$ and $k = 3$, an effective dictionary can be built, since $P_R^{Overall}$ and $P_F^{Overall}$ are low, which suggests that real radicals tend to occur more than k times while fake radicals tend to occur less than k times.

Table 1: the probabilities of P_R & P_F ($\theta = 5$; $k = 3$)

-	# character(m)	$P_R(m)$	$P_F(m)$
$m = 3$	25088	0.0 %	0.00008474 %
$m = 4$	18780	10.35 %	0.00008493 %
$m = 5$	5907	31.74 %	0.00008511 %
$m = 6$	525	50.00 %	0.00008530 %
$m = 7$	13	63.21 %	0.00008549 %
$P_{RorF}^{Overall}$		**8.13 %**	**0.00008486 %**

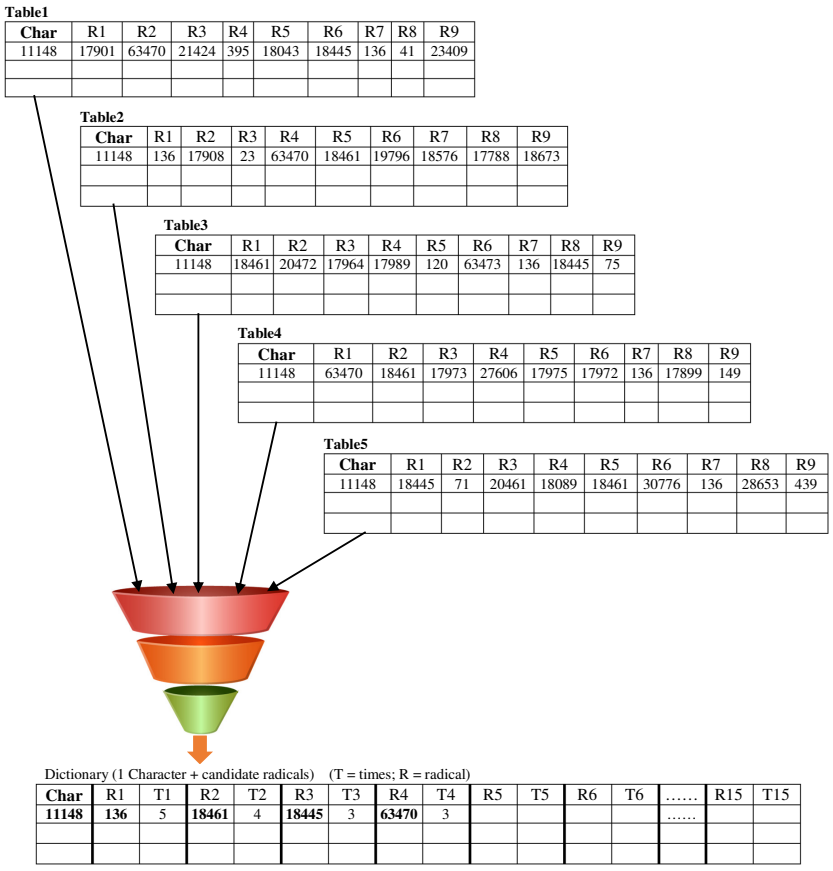

Fig. 8: The dictionary construction process for a character

An example of dictionary building is given in Fig. 8, where we use the character 11148 and $\theta = 5$. Each puzzle is stored in a different table that contains the character alongside with 9 candidate radicals. We put together the 5 puzzles and count the occurrences of each candidate radical. Radical 136 occurs 5 times, 18461 occurs 4 times, and both 18445 and 63470 occur 3 times, and therefore all of them are determined as real radicals for the character. Figure. 9 shows all the corresponding images identified by their IDs, confirming the correctness of our algorithm.

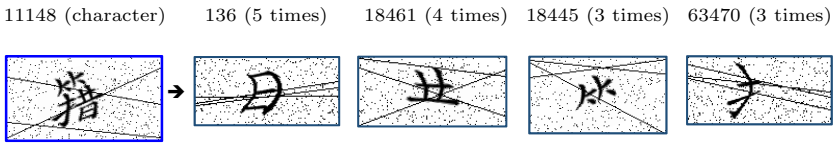

Fig. 9: An example on a character with the classified real radicals

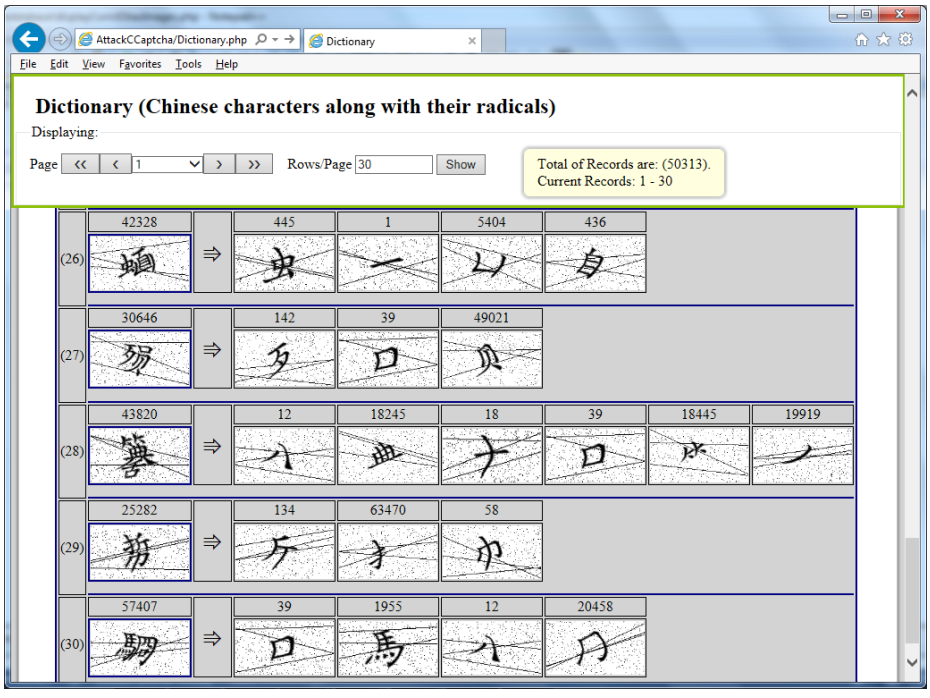

Fig. 10: Dictionary visualization

Experiment results. We spent about two weeks building a dictionary with $\theta = 5$ tables, i.e. 5 different puzzles for each character. Figure. 10 visualizes a small part of the dictionary. With this dictionary, we tested on 2000 new puzzles that were randomly generated by the service, and completely solved 1833 of them, achieving a success rate of 91.65%. For 152 failed cases, 2 out of the 3 real radicals were successfully identified by our dictionary attack. It takes on average about 0.093 seconds to solve a puzzle on a standard desktop computer.

To increase the success of our dictionary attack, we can further suppress $P_R^{Overall}$ and $P_F^{Overall}$ by increasing θ (the number of tables) and choosing an appropriate threshold k. For example, when we set $\theta = 10$ and $k = 5$, $P_R^{Overall}$ becomes 3.10% and $P_F^{Overall}$ becomes 0.000000041%, which are both much lower than the previous configurations. The dictionary attack's success will be improved accordingly.

3.3 Verification Abuse

Further investigation into the CCaptcha library "*labcrecaptcha.php*"reveals that a script named "*verify.php*"is responsible for Captcha verification. The key API is defined as follows:

```
function check_crecaptcha_answer($remoteaddr, $puzzle, $answer,
                                 $useragent, $userlanguage, $setting)
```

It returns '*Success*' if the user's answer is correct, and '*Failure*' otherwise. Two parameters of the API are defined but not really used in the verification process, and they are *remote address* and *user agent*. We find that adversaries can abuse this API for the following oracle attacks.

Bypass the service. For any puzzle generated by the service, an attacker can ask the server to do a brute force search for the correct answer, and then the attacker uses the answer to pass the test. The pseudocode in Algorithm 2 shows our attack. We enumerate each combination of 3 candidate radicals, and then send it to the service one by one. It takes at most 84 trials (i.e., C_3^9) to know which combination is correct. In our experiment, our success rate is 100% and it takes less than 30 seconds on average to get the correct answer. This attack enables adversaries to entirely bypass the CCaptcha test.

Improve dictionary quality. We can also improve our dictionary (constructed in Section 3.2) by abusing the API to ensure the correctness of dictionary entries, e.g. by filtering out fake radicals.

We first run Algorithm 1 to build our dictionary, which mainly keeps track of radicals appearing more than 3 times in five puzzles. Since only 5 tables are used for dictionary construction, we also keep track of each radical that appears only twice. Next, for each dictionary entry, we sent a series of requests to the verification script, each request including a character together with three candidate radicals, and the **check_crecaptcha_answer** API will tell us whether they are the right combination.

Algorithm 2: Brute Force Search

input : Puzzle (1 character + 9 radicals$_{\{3\,\text{real}+6\,\text{faux}\}}$)
output: Correct solution (IDs of correct radicals that compose the Target character)

P: *Puzzle* (1 *character* + 9 *radicals*$_{\{3real+6faux\}}$);
P[*char*] : *Target Character ID* (*i.e. Big image*);
P[R_n] : *Radical ID* (*i.e. small Image*), *where* $n = 1, 2, 3, \ldots, 9$;

P $\leftarrow$ *grab a new puzzle needed to solve*;
for $i \leftarrow 1$ **to** 7 **do**
  for $j \leftarrow i + 1$ **to** 8 **do**
    for $k \leftarrow j + 1$ **to** 9 **do**
      Possible_Solution = {P[R_i], P[R_j], P[R_k]} ;
      Result = `check_crecaptcha_answer` (P, Possible_Solution,...);
      if (Result = *success*) **then**
        Correct_Solution = Possible_Solution;
        Exit;
      end
    end
  end
end

This way, we have successfully eliminated each radical that accidentally repeats at least 3 times but is not part of the correct combination. On the other hand, we have also found that some radicals appear only twice, but are real roots from target characters.

However, this enhancement method hardly produces a perfect dictionary for a simple reason: when we choose small numbers for k and θ, some real radicals will never be observed in the say 5 tables.

Build an optimal dictionary. We can also build an attack dictionary by abusing the verification API alone. To do so, we ask the service to perform a brute force search for each character. That is, at most 84 requests will identify 3 real radicals for a character, creating a dictionary entry. The dictionary constructed this way will be optimal, but it will take a long time, much longer than required for building the improved dictionary described earlier.

4 Countermeasures

The vulnerabilities we have discussed are mostly due to architecture flaws in the system design, and they can be addressed by carefully thinking about architecture issues.

All the information leakage vulnerabilities can be prevented by significantly restructuring the division of labour between the CCaptcha server and the code executed on an end user's computer (i.e. the client). Specifically, generating

Captchas can be entirely done by the server without interacting with the client. The server randomly picks a target character, and assembles it with nine candidate radicals into a big image like the one shown in Fig 1. Then, the image is sent to the client, which will display the image to the end user, collect her inputs in the form of a series of coordinate pairs describing where she has clicked on the image, and then send the inputs to the server. Next, the server interprets which candidate radicals the user has clicked, and determines whether the clicked radicals are real ones or not.

The one-to-one relationship between a character (or a radical) and its image ID is a devastating vulnerability. It could be fixed by hashing the ID with a random number to give each image a completely different name each time. However, this quick hack solution is not needed anymore if the new architecture discussed above is in place.

Our dictionary attack was built on both the leaked database and the one-to-one mapping between characters (or radicals) and their IDs. With both of the problems being fixed, the dictionary attack will no longer work.

Verification abuse has not only contributed to improve our dictionary attack, but it can be used as a standalone attack by itself. The root causes of verification abuse are two serious flaws in the architecture design:

1. No mechanism is in place to identity a web server that interacts with the CCaptcha server, and thus any third party can request the CCaptcha server for Captcha generation and verification.

2. No mechanism is in place to vouch and verify the authenticity of each Captcha puzzle, and the CCaptcha server fails to tell whether a puzzle is issued by itself or not.

Such critical flaws have allowed us to launch various oracle attacks. Even when we sent millions of requests to the server, we have experienced little obstruction.

To fix these problems, we recommend an improved system architecture, which is shown in Fig. 11.

First, the CCaptcha service should provide each website a unique pair of API keys: one is a site key that uniquely identifies a website, the other a secret key that is known only to the website and the CCaptcha server. An end user will get the site key from the web server (step 1 in Fig. 11), and use the key to retrieve a Captcha from the CCaptcha server (step 2-A). The secret key is used for authenticated communication between the web server and the CCaptcha server in the verification stage (step 4-A) to prevent the verification abuse.

The CCaptcha service can introduce an enrolment process, in which the pair of API keys is generated upon a website's registration with the service. Second, a unique ID (token) should accompany each new puzzle issued by the Captcha generator (step 2-B). Each token should be used only once. This token should also accompany the solution when the latter is sent from the web server to the CCaptcha server in the verification process (step 4-A). Using this token, the CCaptcha server can determine whether a puzzle is authentic or issued by a party

impersonating the Captcha generator. Additionally, the end of the puzzle's life should be associated with the end of the token's life in the verification process.

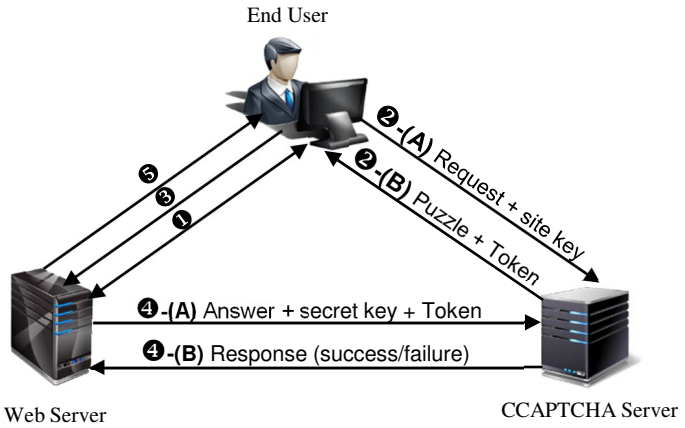

Fig. 11: A revised system architecture

When adversaries have a website and an end user both under their control, they will have legitimate access to the API keys and can misuse them. For example, the adversaries can act first as a legitimate web server and register for free access to an API key pair, and they can act thereafter as a legitimate end-user. A simple, cheap but imperfect solution we suggest is for the CCaptcha server to monitor the number of requests from each website, and to apply rate control when traffic anomalies are detected.

5 Lessons

It is notoriously hard to design security APIs. With nearly thirty years of research in cryptographic protocols, it is still a challenge to get a novel design right. Security APIs are much harder to design than cryptographic protocols. Therefore, it is crucial to understand failures of security APIs, and learn from them.

We conclude this paper by summarising lessons that we have learned both from identifying the API attacks on CCaptcha, and from fixing the vulnerabilities.

Lesson 1: Security policies were not articulated by the designers. Otherwise, entirely bypassing the service should probably have never happened in the first place.

In this specific context, at least three security policies are relevant:

1. Do not trust client;
2. No leakage of sensitive materials;

3. No bypass of the service without solving an automated Turing test.

Lesson 2: System architecture is highly relevant to security APIs, but it was not carefully considered, and not articulated either.

Probably the most important lesson we have learned is the following. A careful analysis of system architecture is useful and effective for identifying vulnerabilities in security APIs and for figuring out suitable countermeasures. To the best of our knowledge, this insight was never spelled out in the literature.

To extrapolate it a little further, we believe that for any complex system (including a system of systems) where multiple components interact with each other, an architecture analysis will prove an effective method both for identifying security vulnerabilities in the system and for fixing them. This analysis can be applied in many stages of the system's life circle such as design, implementation and testing.

This method of 'architecture analysis for security' deserves further study and is our ongoing research.

6 Acknowledgement

We thank Butler Lampson for inspiring conversations, Yu Guan for assistances, and anonymous reviewers for helpful comments.

References

1. Anderson, R.: The correctness of crypto transaction sets. In: 8th International Workshop on Security Protocols. Cambridge, UK (2000)
2. Berkman, O., Ostrovsky, O.: The unbearable lightness of pin cracking. In: Financial Cryptography and Data Security, pp. 224–238 (2007)
3. Bond, M.: Understanding Security APIs. Ph.D. thesis, University of Cambridge (2004)
4. Bond, M., Anderson, R.: Api level attacks on embedded systems. In: IEEE Computer Magazine. pp. 67–75 (2001)
5. Bursztein, E., Martin, M., Mitchell, J.C.: Text-based captcha strengths and weaknesses. In: Proceedings of the 18th ACM conference on Computer and communications security. ACM (2011)
6. Chen, L.: Personal Communications (2014)
7. Chen, L., Juang, D., Zhu, W., Yu, H., Chen, F.: CAPTCHA AND reCAPTCHA WITH SINOGRAPHS. Patent US20120023549 A1- (2012)
8. Clulow, J.: On the security of pkcs# 11. In: Cryptographic Hardware and Embedded Systems - CHES 2003, pp. 411–425 (2003)
9. Cortier, V., Steel, G.: A generic security api for symmetric key management on cryptographic devices. In: Information and ComputationComputer SecurityESORICS 2009, pp. 605–620 (2009)
10. Gao, H., Wang, W., Qi, J., Wang, X., Liu, X., Yan, J.: The robustness of hollow captchas. In: Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security - CCS '13. pp. 1075–1086. New York, USA (2013)

11. Gao, H., Yan, J., et al: A simple generic attack on text captchas. In: Proc. Network and Distributed System Security Symposium (NDSS). San Diego, USA (2016)
12. Watson, R.N.M.: Exploiting concurrency vulnerabilities in system call wrappers. In: First USENIX Workshop on Offensive Technologies (WOOT 07) (2007)
13. Yan, J., El Ahmad, A.S.: Breaking visual captchas with naïve pattern recognition algorithms. In: 23rd annual Computer Security Applications Conference - ACSAC '07. USA (2007)
14. Yan, J., El Ahmad, A.S.: A low-cost attack on a microsoft captcha. In: Proceedings of the 15th ACM conference on Computer and communications security - CCS '08. pp. 543–554. New York, NY, USA (2008)