Attacking CAPTCHAs for Fun and Profit

Author:

Gursev Singh Kalra
Managing Consultant
Foundstone Professional Services

Table of Contents

Introduction

A "**C**ompletely **A**utomated **P**ublic **T**uring test to tell **C**omputers and **H**umans **A**part", or "CAPTCHA" is used to prevent automated software from performing actions which degrade the quality of service of a given system. CAPTCHAs aim to ensure the users of applications are human, which ultimately aid in preventing unauthorized access and abuse.

To analyze the strength of CAPTHA implementations on the internet, research was conducted covering several high traffic websites. During the research CAPTCHA protection on three types of forms were reviewed:

1. Registration pages
2. Forgot password functionality
3. User comment fields for blog posts, news articles etc…

The vulnerabilities identified during the research were classified into three broad categories: breaching client side trust, manipulating server side implementation, and attacking the CAPTCHA image. In this paper, we will look at the interesting and the most common vulnerabilities identified during the research.

A Strong CAPTCHA Implementation

To begin with, let us try to understand an example CAPTCHA implementation and various caveats that make this implementation strong. The image and description below explain the various steps of the CAPTCHA generation and verification process.

1. Client requests a CAPTCHA from the server with or without a valid SESSIONID.
2. If the client does not provide a valid SESSIONID, a new SESSIONID is generated and corresponding session store is instantiated.
3. The server side code creates a new CAPTCHA with random text.
4. CAPTCHA solution is stored in the HTTP session store.
5. CAPTCHA image is sent to the client. If client request did not provide a valid SESSIONID, the newly generated SESSIONID in step 2 is also returned.
6. Client sends CAPTCHA solution along with SESSIONID for verification.
7. Server side code retrieves CAPTCHA solution from the HTTP Session and verifies it against the solution provided by the client.

8. Server side CAPTCHA state is cleared (we will see why).
9. If verification is successful, client is sent to next logical step. If not, client is forced to request a new CAPTCHA (step 1).

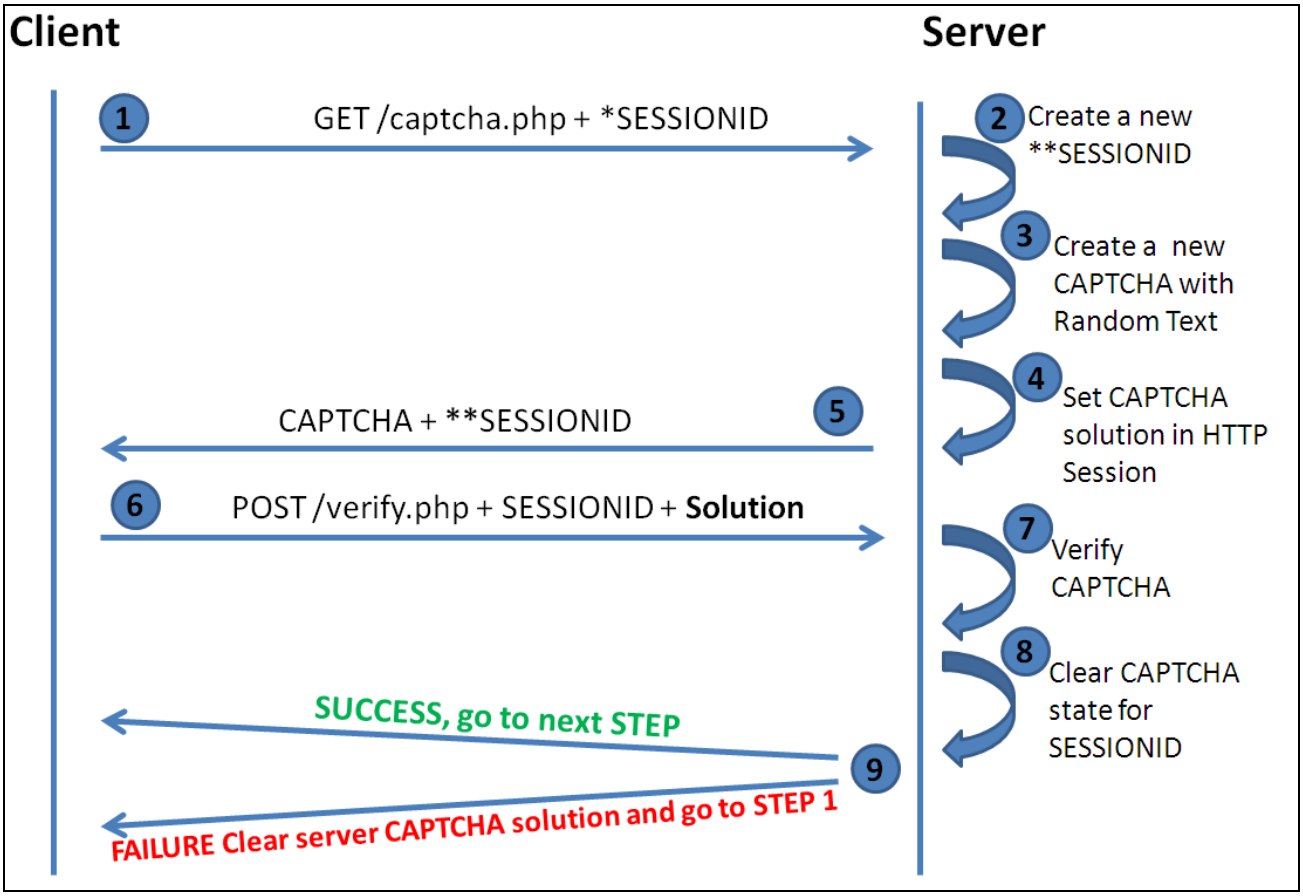


Figure 1: Image shows a sample strong CAPTCHA implementation

Additional considerations for strong CAPTCHA implementation:

1. The client should not have any "influence on" or "knowledge about" the CAPTCHA content.
2. The CAPTCHA text must be randomly generated and should have a large sample space.
3. CAPTCHA image should be so created that it deters automated extraction of text by increasing the complexity to perform image preprocessing, segmentation and classification.
4. The client should not have direct access to the CAPTCHA solution.
5. CAPTCHAs should not be reused.

Let us now look at various vulnerabilities identified during the research.

Breaching Client Side Trust

Secure design principles for web applications and distributed systems recommend not trusting the client for performing security checks. During the research it was observed that many developers relied on the clients to perform CAPTCHA validation, generation, and storage. This allowed the client to directly access CAPTCHA solution, bypass the verification process, and generate CAPTCHAs of its own choice. Client side flaws identified during analysis of CAPTCHA implementations are discussed below.

Hidden Fields and Client Side Storage

Hidden fields have long been used as an insecure means to pass sensitive information between client and server. Against popular wisdom, CAPTCHA implementations were found to rely on hidden fields to relay CAPTCHA solutions between client and server. These implementations completely relied on a client provided value for both the CAPTCHA solution and the user entered CAPTCHA value. An attacker could provide values of his choice and the server has no means of performing meaningful validation as it does not have access to original CAPTCHA solution. This particular implementation requires minimum effort to break and does not offer any protection.

Occasionally, it was observed that some implementations relied on JavaScript code and hidden fields to verify the CAPTCHA on the client side with no validation on server side.

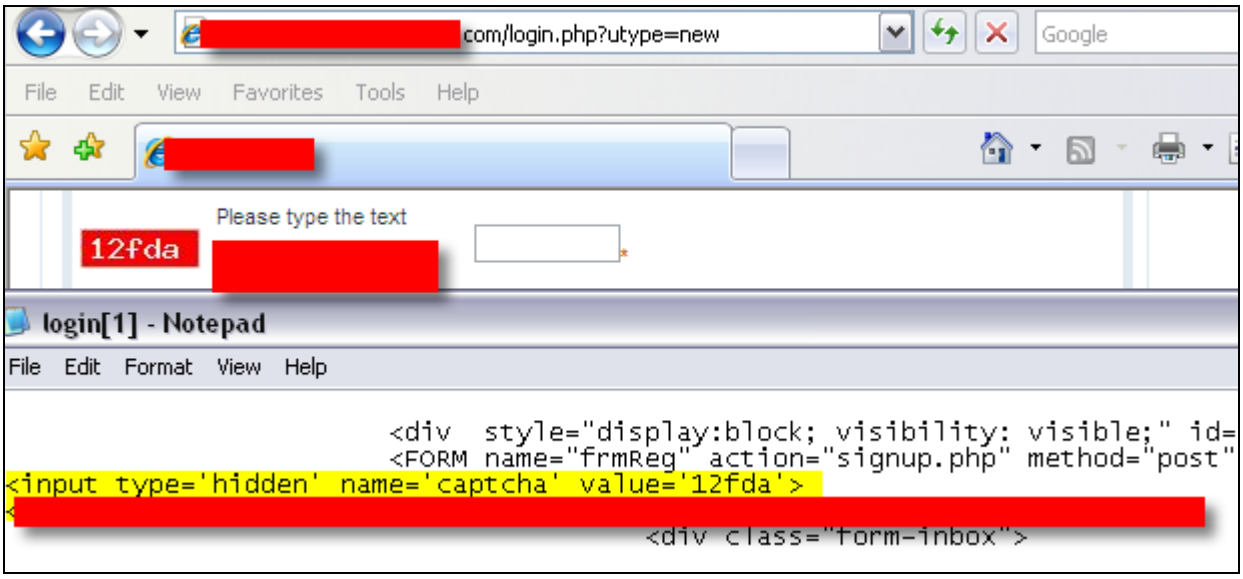


Figure 2: Image shows hidden field being used to transmit CAPTCHA value between client and server

Figure 3: Image below shows user manipulated values being sent to the server

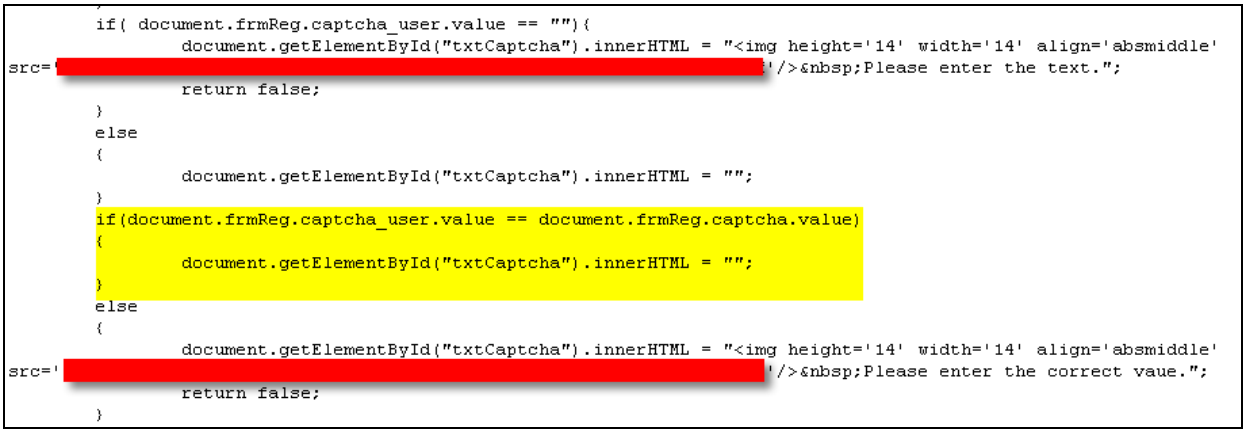


Figure 4: Image shows CAPTCHA JavaScript code to validate CAPTCHA on the web browser

Chosen CAPTCHA Text Attack

During the research it was observed that a few websites delegated CAPTCHA generation routines to the clients while retaining the verification component at the server. This delegation allows the attacker to choose the CAPTCHA value and completely bypass the protection offered. Hence the name "Chosen CAPTCHA Text Attack".

An observed real world implementation is explained below:

1. On the registration page, JavaScript code was used to generate a random number.

2. This random number was sent to server along with SESSIONID to generate a CAPTCHA image.
3. The server generated the CAPTCHA image with random number received from the client. The random number was also stored in HTTP session for verification purpose.
4. The CAPTCHA image was retrieved and displayed on the registration page as a challenge for the user.

To exploit this vulnerability, an attacker has to do the following:

1. Obtain a valid SESSIONID.
2. Set the CAPTCHA value of his choice into the HTTP session by using the SESSIONID obtained in above step.
3. Make a submission with the attacker generated CAPTCHA value to bypass the protection.

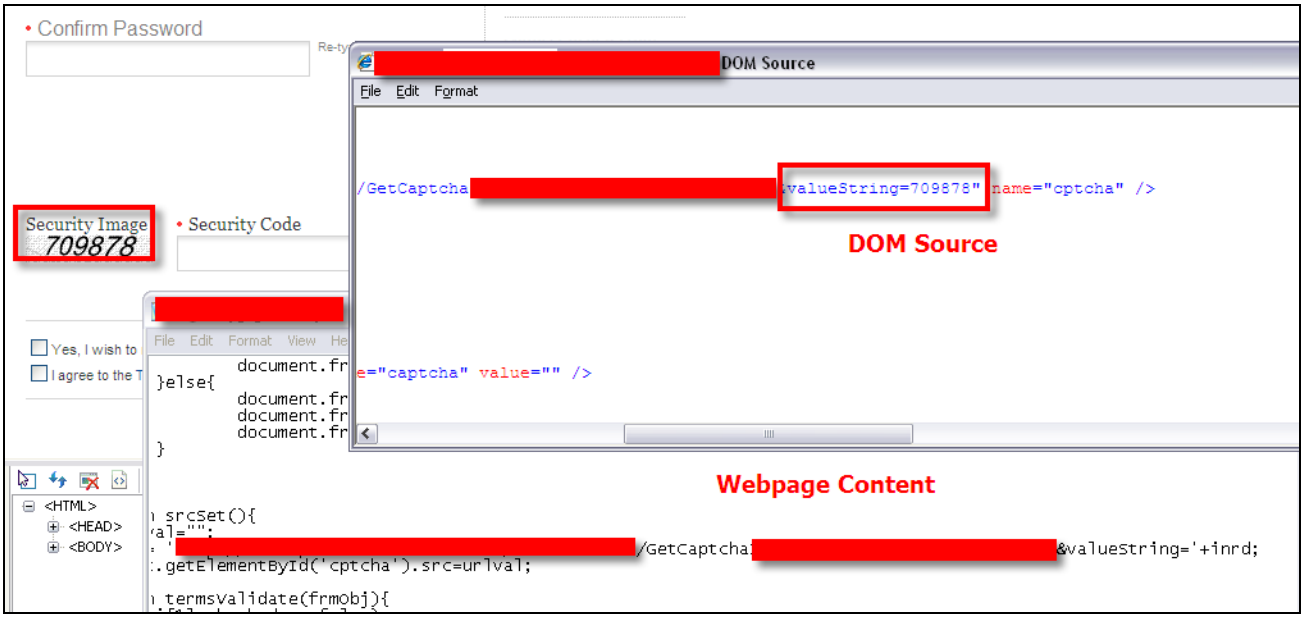


Figure 5: Image shows JavaScript generated CAPTCHA code, corresponding JavaScript code and DOM source

Figure 6: Image shows CAPTCHA value as a Base64 encoded value during image retrieval

Arithmetic CAPTCHAs

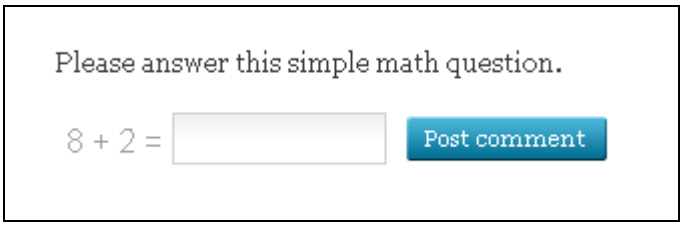


Figure 7: image shows an example Arithmetic CAPTCHA

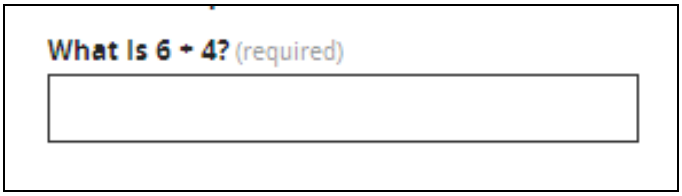


Figure 8: image shows an example Arithmetic CAPTCHA

Arithmetic CAPTCHAs require the user to solve an arithmetic problem. When CAPTCHA data is stored client-side, the effort required to bypass this CAPTCHA implementation is minimal: just parse the HTML content of the returned page, extract the arithmetic question, and solve it at client side. Thus any implementation that stores CAPTCHA data client-side, fails to offer any significant protection.

Server Side Attacks

So far we have looked at attacks that target the client side trust. Let us now look at various attacks that target the server side implementation flaws.

CAPTCHA Rainbow Tables

As discussed earlier, randomly generating CAPTCHAs during runtime is one of the important aspects of a secure CAPTCHA design. During the research, it was observed that a very large number of websites used finite number of CAPTCHAs and each CAPTCHA was recognized using an identifier. These identifiers were observed to be either numeric or finite length character strings. The identifiers were generally sent to the client as hidden fields or were available as part of URL while retrieving the CAPTCHA.

Further, some websites did not change the CAPTCHA identifiers ever; others chose to randomly change identifiers on periodic basis. Rainbow table based attack vectors targeting websites that use a finite CAPTCHA set are discussed below:

Attacking Static CAPTCHA Identifiers

For websites that use static CAPTCHA identifiers, a large number of CAPTCHAs can be downloaded and solved locally using Optical Character Recognition (OCR) engines, custom solvers, or manually. A rainbow table can then be created with static CAPTCHA identifier and the solution. Whenever the server returns a CAPTCHA identifier for which there is a pre-solved value available, the solution can be quickly looked up and submitted to bypass the CAPTCHA restriction. Multiple CAPTCHA requests can also be made to so that CAPTCHA with known identifier is returned by the server.

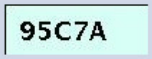
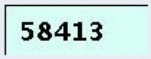
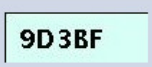
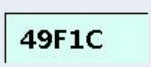
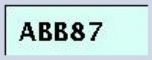
Numeric Identifier	CAPTCHA	Solution
0		95C7A
1		58413
2		9D3BF
3		49F1C
4		ABB87
...		
99999		D498A

Figure 9: Image shows sample CAPTCHA Rainbow table implementation with numeric identifiers

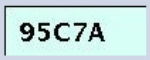
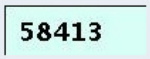
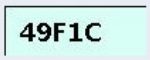
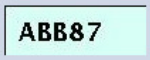
Alphanumeric Identifier	CAPTCHA	Solution
uJSqsPvjxc6		95C7A
9WzrowjPEqI		58413
nm8SfvtEwpP		9D3BF
fespW5LVqNQ		49F1C
dgLSB1CKJRJ		ABB87
...		
QmJF3TQazcH		D498A

Figure 10: Image shows sample CAPTCHA Rainbow table implementation with alphanumeric identifiers

Attacking Dynamic CAPTCHA Identifiers

A computationally slow alternative exists for implementations that periodically or randomly change CAPTCHA identifiers but retain their finite image set. Similar results can be achieved by the following:

1. Download a large number of CAPTCHAs locally.
2. Compute cryptographic hashes (MD5/SHA1/etc) for the downloaded CAPTCHAs.
3. Solve the downloaded CAPTCHAs locally using Optical Character Recognition (OCR) engines, custom solvers or manually.
4. Create a rainbow table with CAPTCHA hash (calculated above) as the key and the corresponding solution.
5. Once the server returns a CAPTCHA with a pre-existing hash, the solution can be looked up submitted to bypass the CAPTCHA restriction.

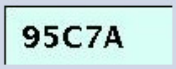
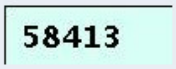
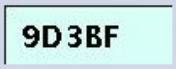
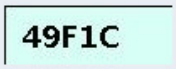
CAPTCHA Content MD5	CAPTCHA	Solution
68ecb8867cd7457421c2eca3227bffbd		95C7A
84a78d24bc9637fcfb152f723b6e8e27		58413
84125db583d64c346d97a74fa9e53848		9D3BF
c6a1ed9477846568cdea62c97e389811		49F1C
e9fa81f69debe45bded7bba4743a8a23		ABB87
. . .		
B9df819f6174d6577661e12859226366		D498A

Figure 11: Image shows sample CAPTCHA Rainbow table created with CAPTCHA MD5 as local identifiers

It was observed that some CAPTCHA implementations change the identifiers as well as insert random noise into images over multiple retrievals. In such scenario, researching and writing custom solvers, or using an existing CAPTCHA solving tool is the suggested attack vector.

The Chosen CAPTCHA Identifier Attack

In certain implementations, servers return the CAPTCHA unique identifiers to the user but did not store the identifier or CAPTCHA solution in the HTTP session. When a form submission arrives, the CAPTCHA identifier was extracted from the request body and then used to perform CAPTCHA solution lookup for verification. Attackers can exploit this behavior by solving a single CAPTCHA, recording its unique identifier and then submitting the recorded identifier and corresponding solution over multiple requests.

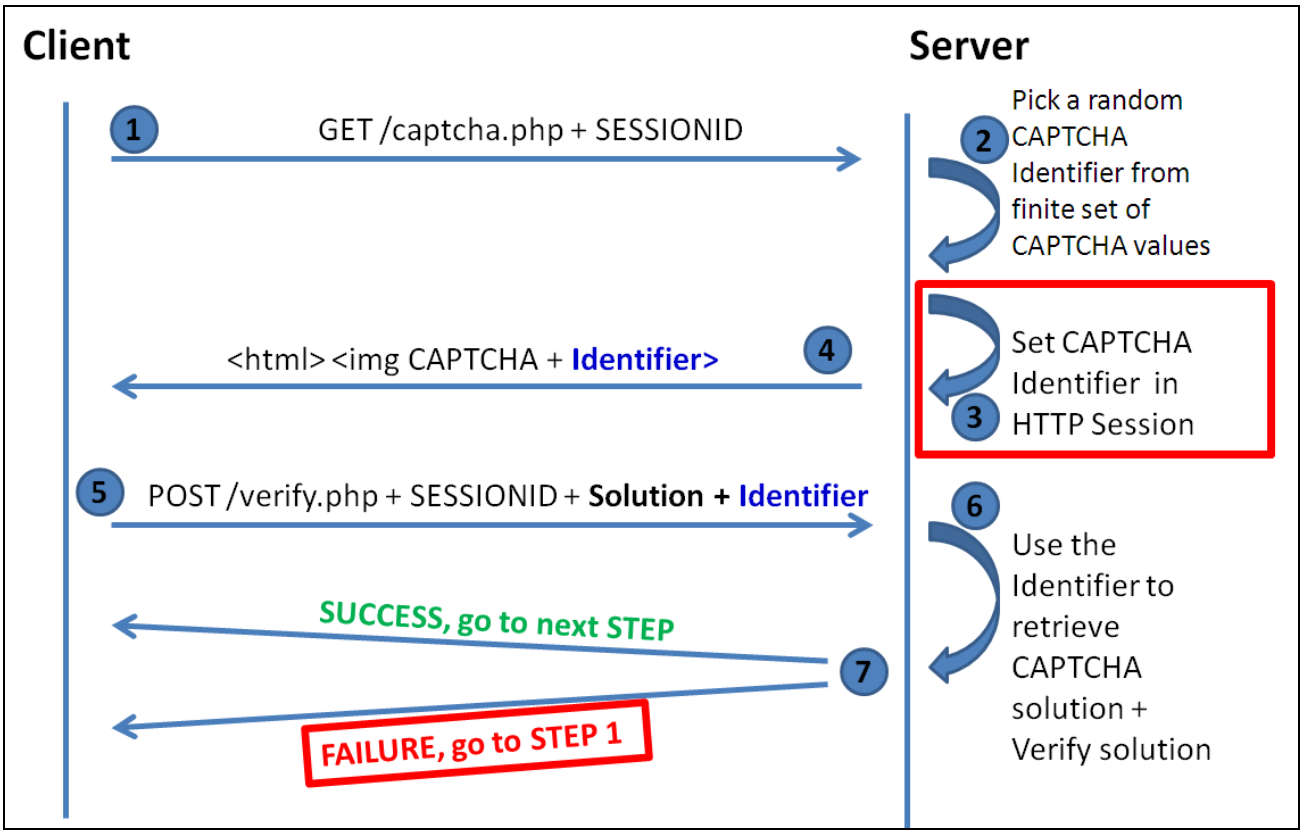


*Figure 12: Image shows a **secure** CAPTCHA implementation scenario where CAPTCHA key is stored in HTTP session*

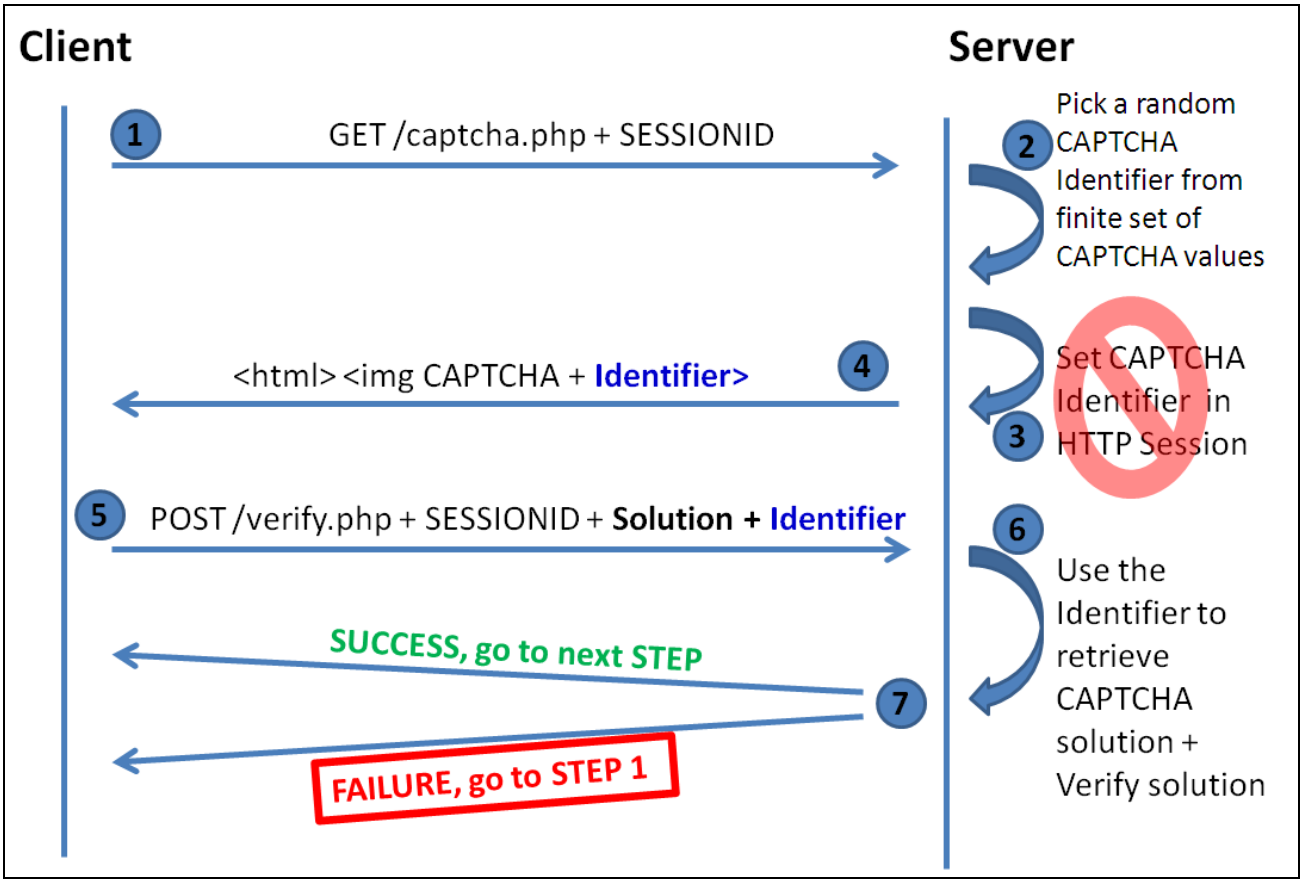


Figure 13: Image shows an insecure CAPTCHA implementation scenario where CAPTCHA identifier is not stored in HTTP session

CAPTCHA Fixation

CAPTCHA Fixation attack exploits a potential race condition in the CAPTCHA implementation relying on unique identifiers for finite CAPTCHA set. This vulnerability allows attackers to insert CAPTCHA identifier of their choice to the HTTP session and then use the pre-solved value to completely bypass CAPTCHA protection. Image and the description below detail a commonly observed implementation scenario and the vulnerability.

1. Client requests a CAPTCHA from the server with a valid SESSIONID.
2. The server picks a random CAPTCHA identifier from the finite set of CAPTCHAs it has.
3. The client is redirected to another URL containing the CAPTCHA identifier from where the CAPTCHA should be retrieved.
4. The client follows the redirect and requests for a CAPTCHA image with the given identifier.
5. The sever stores CAPTCHA identifier in the session.
6. CAPTCHA image is returned.

By not storing the CAPTCHA identifier in HTTP session before sending the identifier to the client, the server exposes itself to CAPTCHA fixation attacks. An attacker can complete steps 1 – 3 (shown in image below) and manipulate the request in step 4 to request for any CAPTCHA identifier for which the correct solution is already known. Once the attacker supplied CAPTCHA identifier is stored inside the HTTP Session at step 4, the corresponding CAPTCHA solution can be provided to bypass the protection.

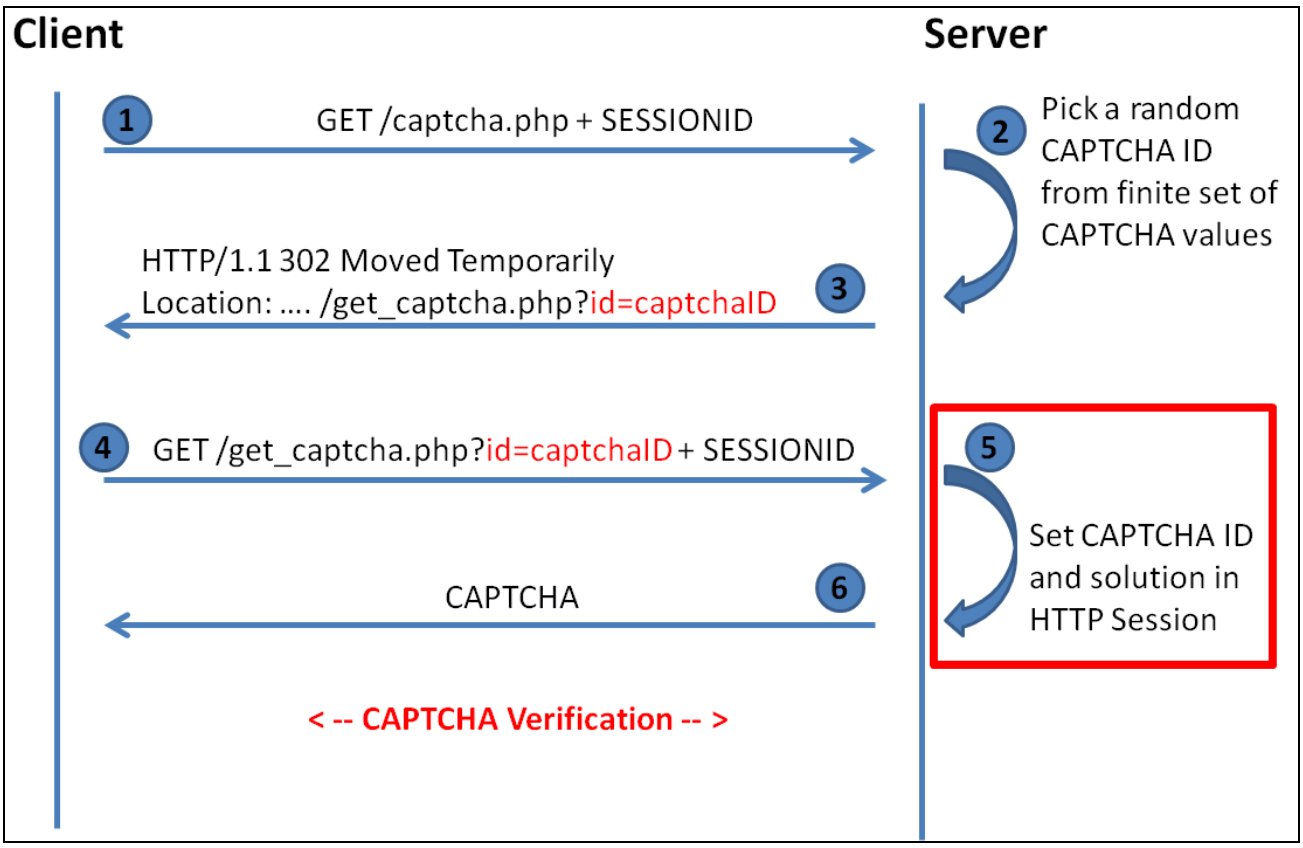


Figure 14: Image shows vulnerable implementation that leads to CAPTCHA Fixation attacks

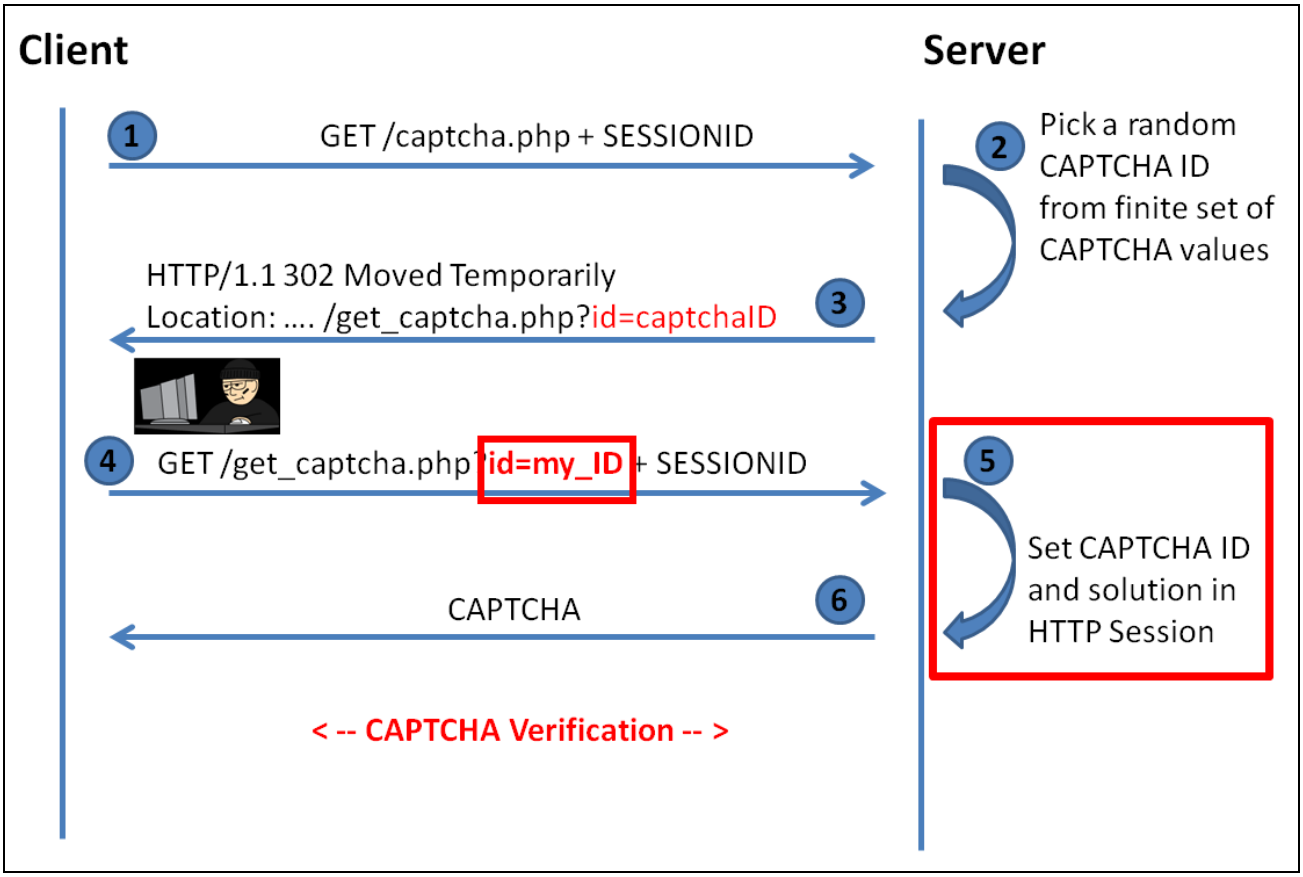


Figure 15: Image shows an example CAPTCHA Fixation attack where attacker provides my_ID as the CAPTCHA identifier

The image below shows an implementation where CAPTCHA identifier is generated and stored in HTTP session before sending the information back to the client.

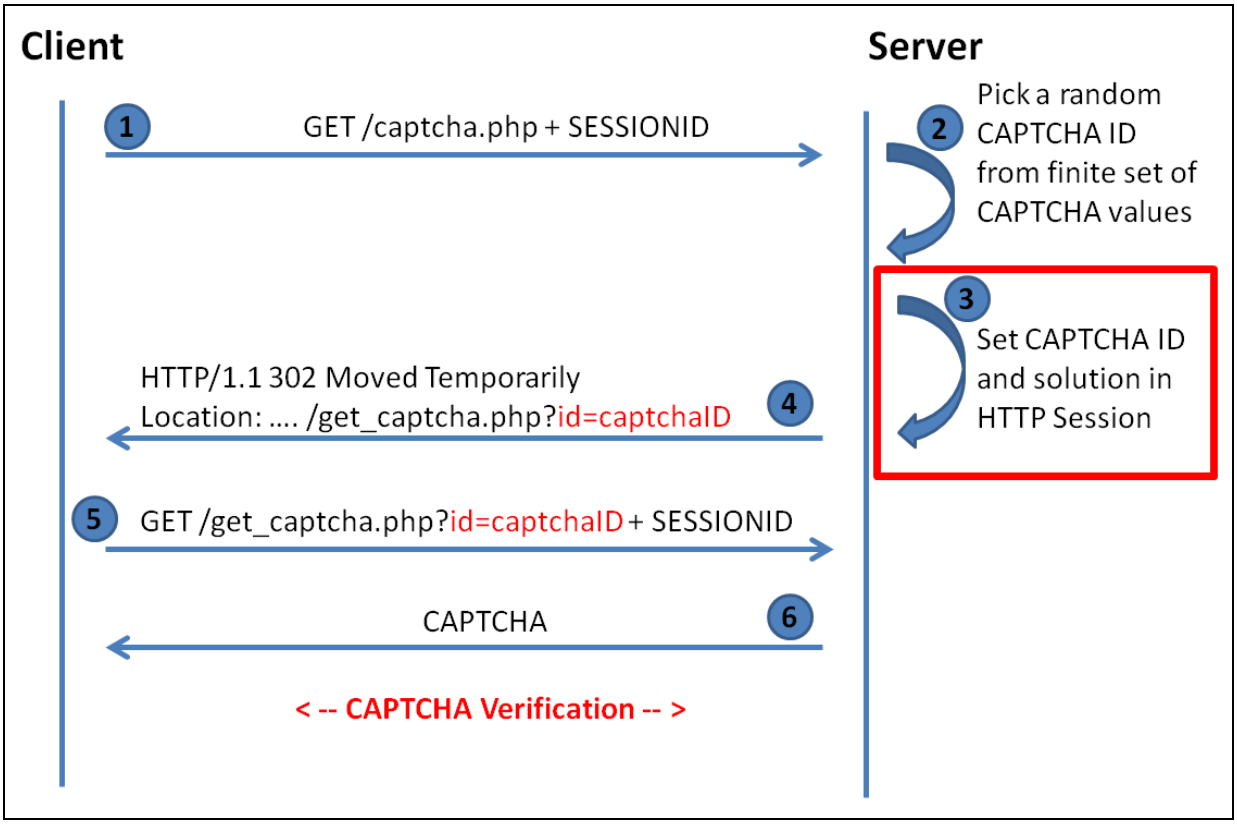


Figure 16: Image shows a secure implementation not vulnerable to CAPTCHA Fixation attacks discussed above

In-Session CAPTCHA Brute-Forcing

The In-Session CAPTCHA Brute-Forcing was one of the most common flaws observed during the research. The widespread existence of this vulnerability is because of the following factors:

1. It is assumed that the client religiously follows the server issued instructions to retrieve a new CAPTCHA if the CAPTCHA verification fails.
2. The code that generates new CAPTCHA and sets the solution in the HTTP session works independently of the code that performs CAPTCHA verification.
3. The code performing CAPTCHA verification does not clear the CAPTCHA solution from HTTP session and hence allows multiple verification attempts on a single CAPCHA solution in that HTTP session.

To exploit this vulnerability, an attacker can direct several submissions directly to the URL that performs CAPTCHA verification and potentially make a successful submission. Steps 5 and 7 in the image below reflect the vulnerability and the exploit scenario.

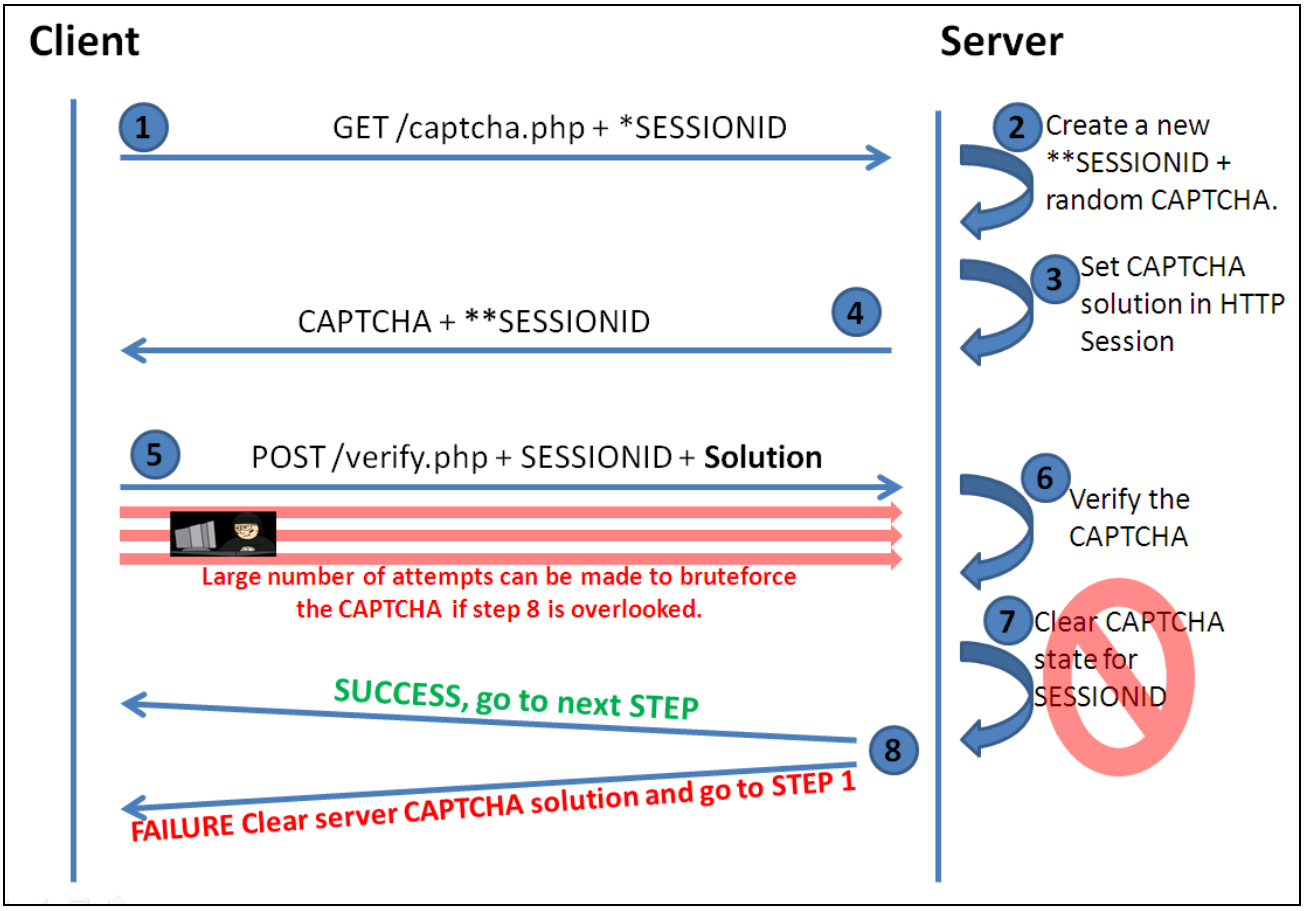


Figure 17: Image shows In Session CAPTCAH bruteforce attack

CAPTCHA Accumulation

Certain CAPTCHA implementations accumulate CAPTCHA solutions or identifiers in their HTTP session. That is, for each request for a new CAPTCHA, the previous value is retained and a new CAPTCHA solution or identifier is also added to the HTTP session. An attacker can exploit this scenario by manually solving one CAPTCHA for an HTTP session and then reusing that solution or identifier and the SESSIONID value to make a large number of successful submissions.

Attacking the Image

A strong CAPTCHA image design is the foundation for an effective anti-automation mechanism. Like encryption, the CAPTCHA image design should be subjected to thorough analysis for its effectiveness against automated text extraction. An alarming number of websites rely on home grown CAPTCHA image designs that offer little protection when subjected to generic image processing techniques and OCR tools.

OCR Assisted CAPTCHA Brute-Forcing

A technique of brute-forcing CAPTCHAs is by leveraging Optical Character Recognition[1] (OCR) software. CAPTCHAs can be copied locally and solved offline using multiple OCR engines. Also, if the CAPTCHA implementation is vulnerable to In-Session CAPTCHA Bruteforce vulnerability discussed above, OCR assisted technique can be used to significantly reduce the number of attempts required to guess the correct solution in a live HTTP session. Following methods can be considered to perform OCR assisted CAPTCHA bruteforce.

1. Each CAPTCHA is subjected to multiple OCR engines and results are combined. The image below shows an example where a CAPTCHA was subjected to two different OCR engines and results were combined. The image assumes that the CAPTCHA implementation is vulnerable to In-Session CAPTCHA bruteforce attack. Here the OCR1 attempt will send `rGsyg` causing a failure. Second OCR will send `r6sy9`, again causing the failure. Since both the solutions differ in two characters, they can be combined to find a correct solution `r6syg`.

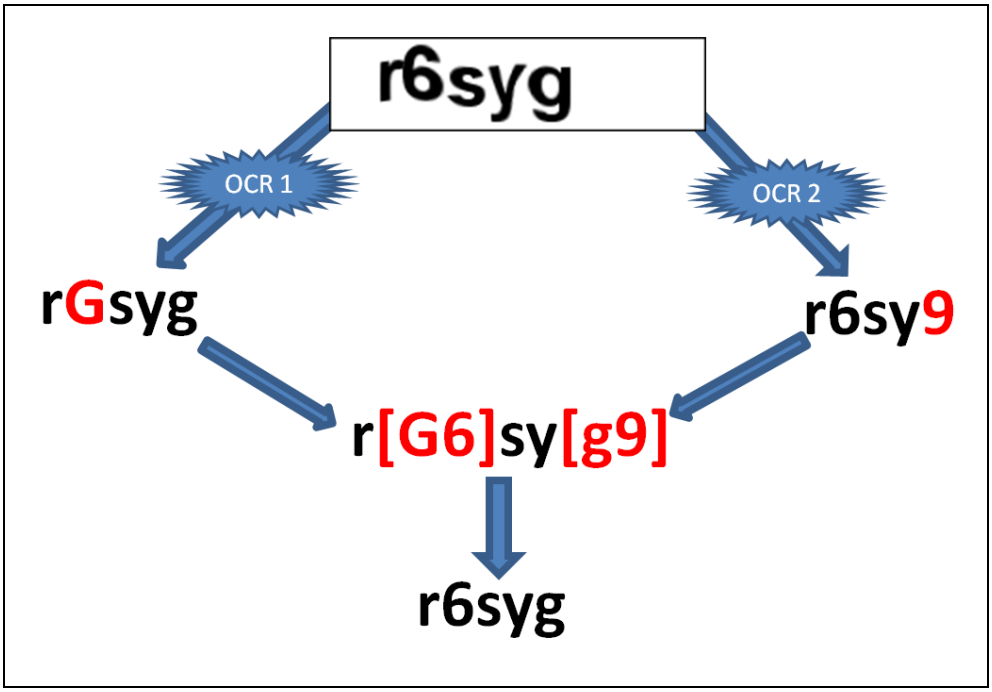


Figure 18: Image shows CAPTCHA solution by combining results of two difference OCR engines

2. After extracting text from CAPTCHA using an OCR engine, selective bruteforce can also be attempted. For example, let us assume that the OCR engine returns the result as `TE5T12`. The bruteforce attempt begins by changing the first character 'T' and retaining the values of the other 5 characters. Then moving on to second character and henceforth. After this, two characters can be bruteforced at

[1] http://en.wikipedia.org/wiki/Optical_character_recognition

tandem, followed by three and till the entire length. This technique, like other bruteforce techniques is high on time and resource requirement.

3. At times, OCR engines may present partially correct solutions. In such scenarios, techniques like simple character substitution techniques can be used to arrive at correct CAPTCHA solution. For example, 'l' can be substituted by 'I', 'G' by 'C', 'S' by '5' etc… Effectiveness of this technique can be enhanced if CAPTCHA character set is known and then relevant substitutions can be performed. For example, if we know that CAPTCHA contains only uppercase characters and the OCR solution contains a number '5', it will be safe to substitute '5' with 'S' to arrive at correct solution.

It is important to note that the OCR engines are better at solving CAPTCHAs with clear text visibility and may not be beneficial for all CAPTCHA types.

Testing CAPTCHAs with TesserCap

TesserCap is a simple CAPTCHA solving tool that can be used to test CAPTCHA images. TesserCap is a GUI based, highly flexible, point and shoot CAPTCHA analysis tool with the following features:

1. A generic image preprocessing engine that can be configured as per the CAPTCHA type being analyzed.
2. Tesseract[2][3] as its OCR engine to retrieve text from preprocessed CAPTCHAs.
3. Web proxy and custom HTTP headers support.
4. CAPTCHA Statistical analysis support.
5. Character set selection for the OCR Engine.

TesserCap and related resources can be downloaded from following locations:

- Tool: http://www.mcafee.com/us/downloads/free-tools/index.aspx.
- Whitepaper: http://www.mcafee.com/apps/view-all/publications.aspx?tf=foundstone&sz=10

Sample TesserCap runs on example CAPTCHAs are shown below.

[2] http://en.wikipedia.org/wiki/Tesseract_(software)
[3] http://code.google.com/p/tesseract-ocr/

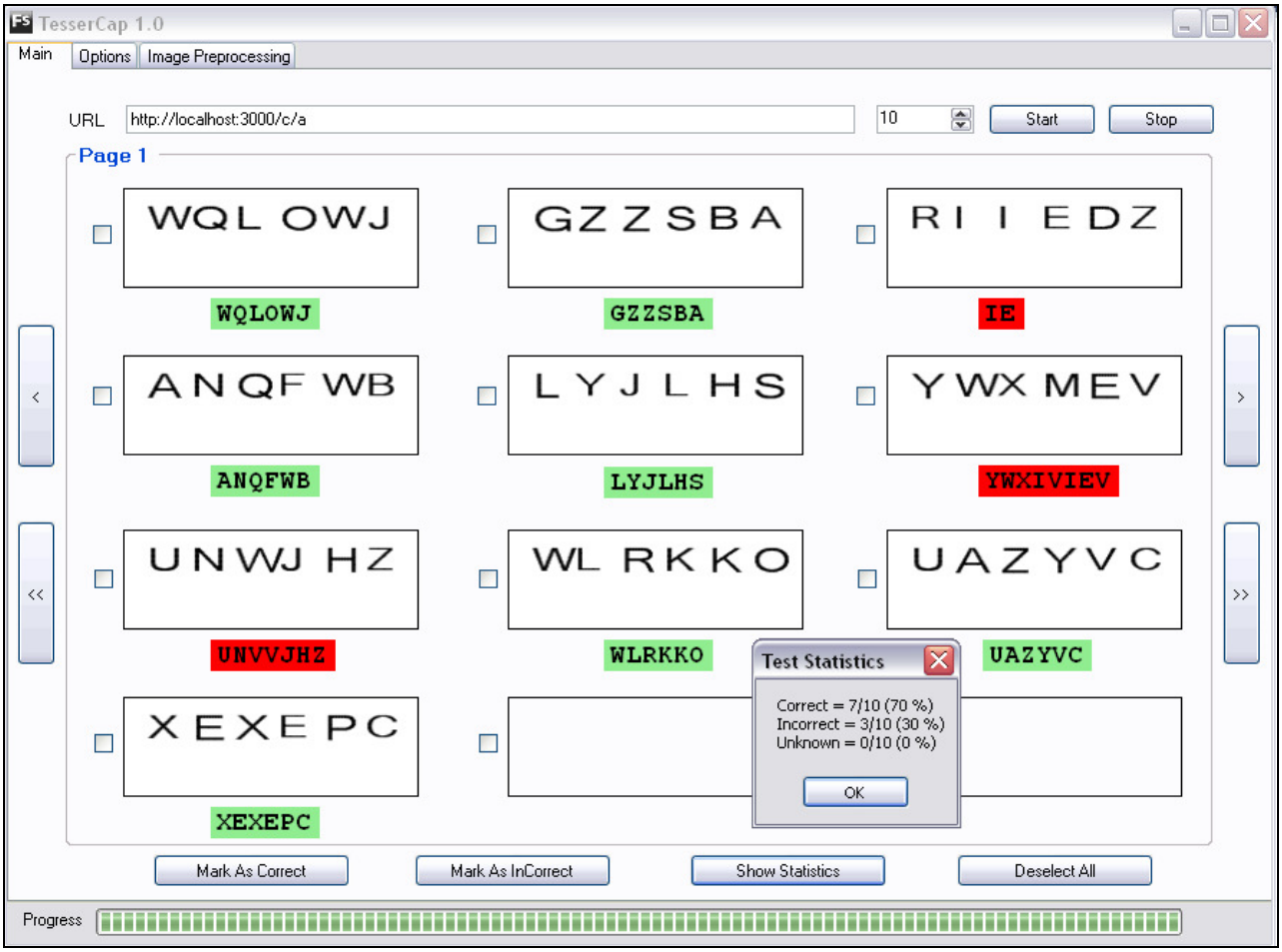


Figure 19: Image shows TesserCap sample Run and test statistics

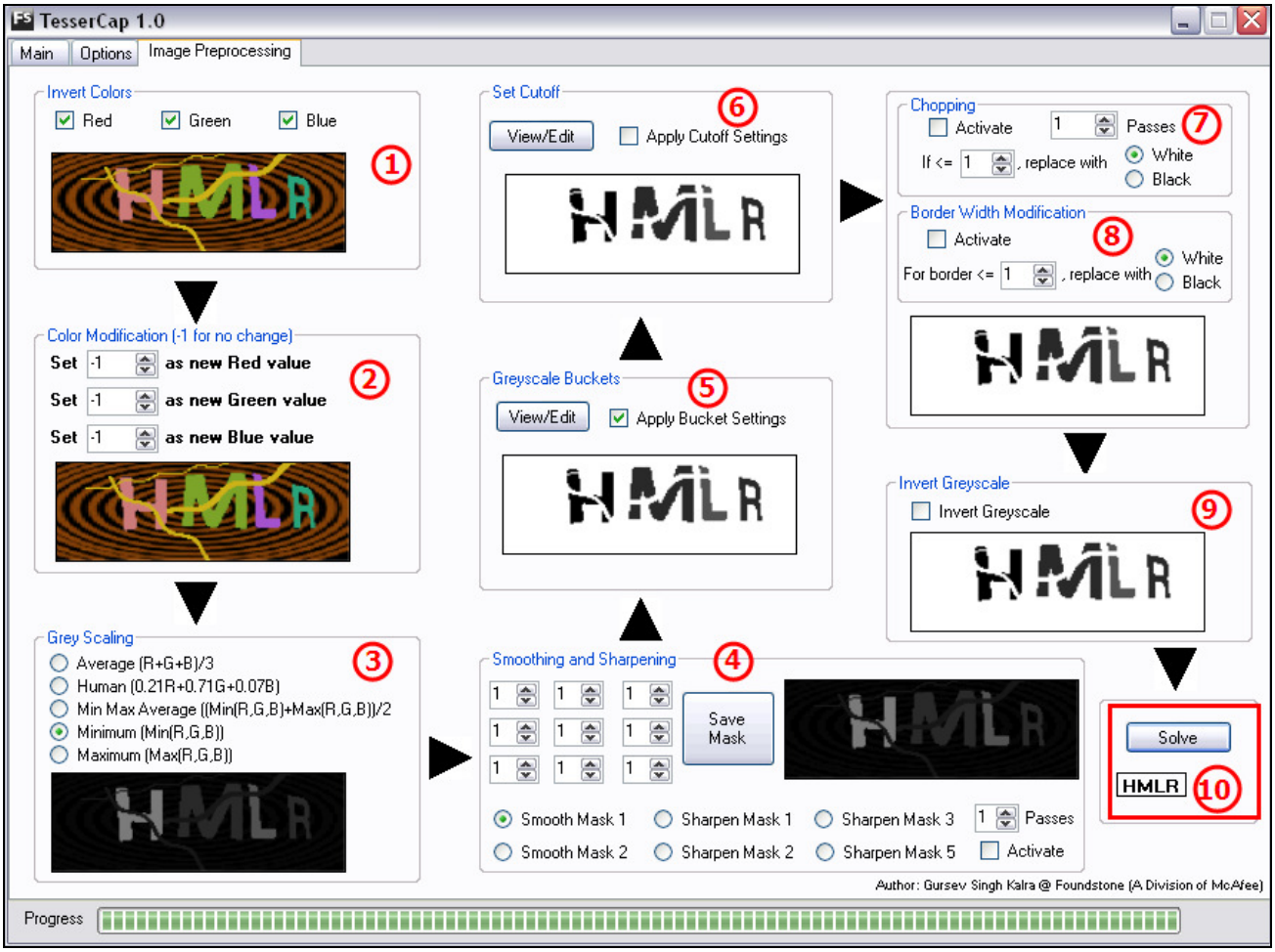


Figure 20: Image shows successful text extraction after applying TesserCap's image preprocessing filters

Writing Custom CAPTCHA Solvers

The most notable approach to solving complex CAPTCHAs is to research the target CAPTCHA scheme and then create custom solvers. This approach requires dedicated effort and is beyond the scope of this whitepaper.

Conclusion

CAPTCHAs have been one of the most potent mechanisms to protect web applications against automated form submissions. As observed in this paper, an assumption or a slight overlook can render a CAPTCHA implementation weak or even ineffective. To have an effective protection against automated forms submissions, it is important to build a strong CAPTCHA ecosystem. A weak CAPTCHA implementation can only provide a false sense of security.

About The Author

Gursev Singh Kalra serves as a Managing Consultant at Foundstone Professional Services, A Division of McAfee. Gursev was a speaker at conferences such as ToorCon, NullCon and ClubHack. Gursev has authored a CAPTCHA testing tool TesserCap and an open source SSL cipher enumeration tool SSLSmart.. Gursev has also developed several internal tools, web applications and loves to code in Ruby, Ruby on Rails and C#.

About Foundstone Professional Services

Foundstone® Professional Services, a division of McAfee. Inc., offers expert services and education to help organizations continuously and measurably protect their most important assets from the most critical threats. Through a strategic approach to security, Foundstone identifies and implements the right balance of technology, people, and process to manage digital risk and leverage security investments more effectively. The company's professional services team consists of recognized security experts and authors with broad security experience with multinational corporations, the public sector, and the US military.