White Paper

Attacking CAPTCHAs for Fun and Profit

By Gursev Singh Kalra
Managing Consultant
McAfee® Foundstone® Professional Services

Table of Contents

Introduction

A "Completely Automated Public Turing test to tell Computers and Humans Apart," or "CAPTCHA," is used to prevent automated software from performing actions that degrade the quality of service of a given system. CAPTCHAs aim to ensure that the users of applications are human and ultimately aid in preventing unauthorized access and abuse.

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

- Registration pages
- Forgotten password functionality
- User comment fields for blog posts, news articles, and other content

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 Secure CAPTCHA Implementation

To begin with, let us try to understand a secure 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. The 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 the client request did not provide a valid SESSIONID, the newly generated SESSIONID in step 2 is also returned.
6. The 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).

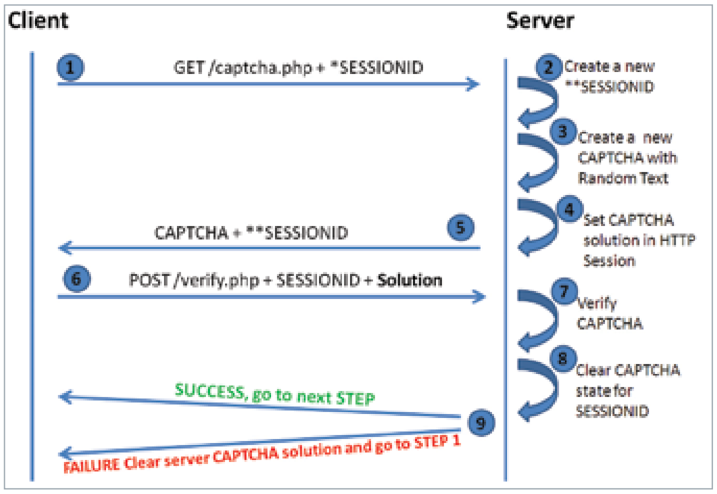

Client
Server
1
GET /captcha.php + *SESSIONID
2
Create a new **SESSIONID
3
Create a new CAPTCHA with Random Text
4
Set CAPTCHA solution in HTTP Session
5
CAPTCHA + **SESSIONID
6
POST /verify.php + SESSIONID + Solution
7
Verify CAPTCHA
8
Clear CAPTCHA state for SESSIONID
SUCCESS, go to next STEP
9
FAILURE Clear server CAPTCHA solution and go to STEP 1

Figure 1. Image shows a secure CAPTCHA implementation.

Here an additional considerations for strong CAPTCHA implementation:

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

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

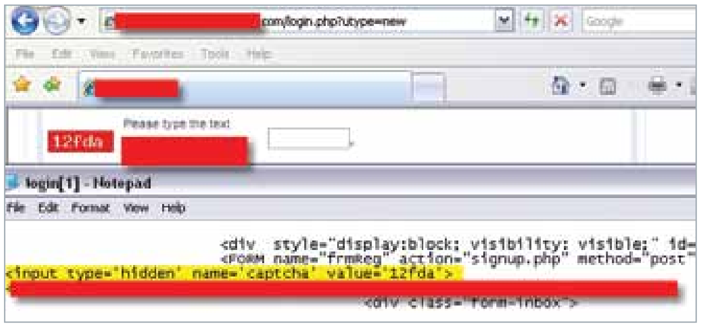
Breaching Client-Side Trust

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

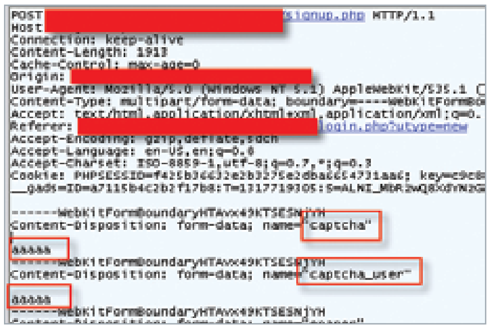
Hidden fields have long been used as an insecure means to pass sensitive information between client and server. Against popular wisdom, CAPTCHA implementations were found to rely on hidden fields to relay CAPTCHA solutions between client and server. These implementations that completely relied on a client 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 the server side.


[redacted]com/login.php?utype=new
Google
File Edit View Favorites Tools Help
[redacted]
12fda
Please type the text
[redacted]
login[1] - Notepad
File Edit Format View Help

```
                    <div  style="display:block; visibility: visible;" id=
                    <FORM name="frmReg" action="signup.php" method="post"
<input type='hidden' name='captcha' value='12fda'>
<[redacted]
                              <div class="form-inbox">
```

Figure 2. A hidden field being used to transmit CAPTCHA value between client and server.



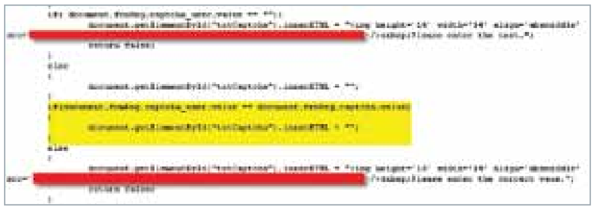
```
POST [redacted]signup.php HTTP/1.1
Host
Connection: keep-alive
Content-Length: 1913
Cache-Control: max-age=0
Origin: [redacted]
User-Agent: Mozilla/5.0 (Windows NT 5.1) AppleWebKit/535.1 (
Content-Type: multipart/form-data; boundary=----WebKitFormBo
Accept: text/html,application/xhtml+xml,application/xml;q=0.
Referer: [redacted]login.php?utype=new
Accept-Encoding: gzip,deflate,sdch
Accept-Language: en-US,en;q=0.8
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.3
Cookie: PHPSESSID=f425b36632e2b3275e2dba6654731aa6; key=c9c8
__gads=ID=a7115b4c2b2f17b8:T=1317719305:S=ALNI_MDR2wQ8XdYN2G

------WebKitFormBoundaryHTAvx49KTSESNjYH
Content-Disposition: form-data; name="captcha"

aaaaa
------WebKitFormBoundaryHTAvx49KTSESNjYH
Content-Disposition: form-data; name="captcha_user"

aaaaa
------WebKitFormBoundaryHTAvx49KTSESNjYH
```

Figure 3. User-manipulated values being sent to the server.



```
    if( document.frmReg.captcha_user.value == ""){
            document.getElementById("txtCaptcha").innerHTML = "<img height='16' width='16' align='absmiddle'
src="[redacted]"/>&nbsp;Please enter the text.";
            return false;
    }
    else
    {
            document.getElementById("txtCaptcha").innerHTML = "";
    }
    if(document.frmReg.captcha_user.value == document.frmReg.captcha.value)
    {
            document.getElementById("txtCaptcha").innerHTML = "";
    }
    else
    {
            document.getElementById("txtCaptcha").innerHTML = "<img height='16' width='16' align='absmiddle'
src="[redacted]"/>&nbsp;Please enter the correct text.";
            return false;
    }
```

Figure 4. CAPTCHA JavaScript code used 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 the server along with a SESSIONID to generate a CAPTCHA image.
3. The server generated the CAPTCHA image with a random number received from the client. The random number was also stored in an HTTP session for verification purposes.
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 the above step.
3. Make a submission with the attacker generated CAPTCHA value to bypass the protection.

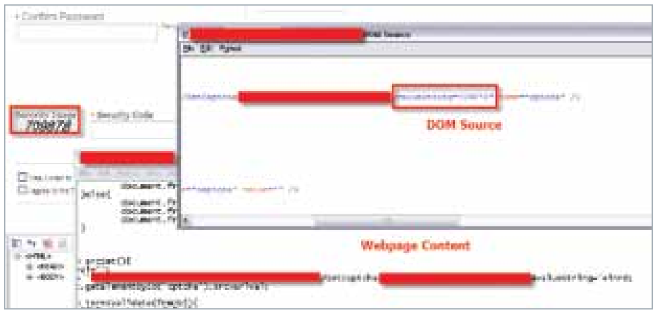


Figure 5. JavaScript-generated CAPTCHA code, corresponding JavaScript code, and DOM source.

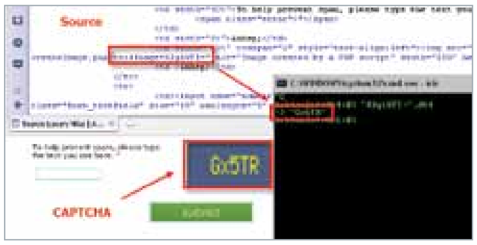
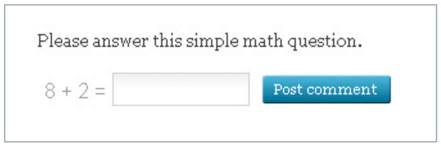


Figure 6. CAPTCHA value as a Base64 encoded value during image retrieval.

Arithmetic CAPTCHAs


Please answer this simple math question.

8 + 2 =

Figure 7. An example of arithmetic CAPTCHA.

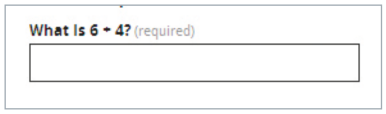

What Is 6 + 4? (required)

Figure 8. An example of 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 client-side trust. Let us now look at various attacks that target the server-side implementation flaws.

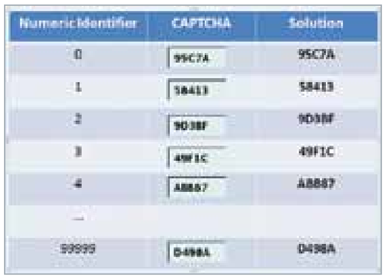
CAPTCHA Rainbow Tables

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 a 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 target websites that use a finite CAPTCHA set are discussed below.

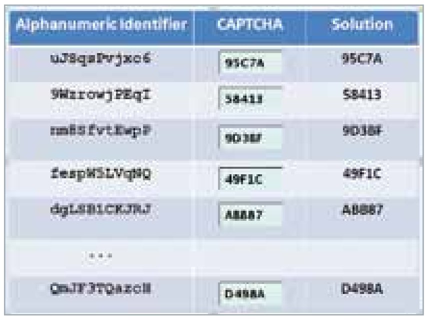
Attacking static CAPTCHA identifiers

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 a 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 a known identifier is returned by the server.



Numeric Identifier	CAPTCHA	Solution
0	95C7A	95C7A
1	58413	58413
2	9D38F	9D38F
3	49F1C	49F1C
4	A8887	A8887
…		
99999	D498A	D498A

Figure 9. Sample CAPTCHA rainbow table implementation with numeric identifiers.



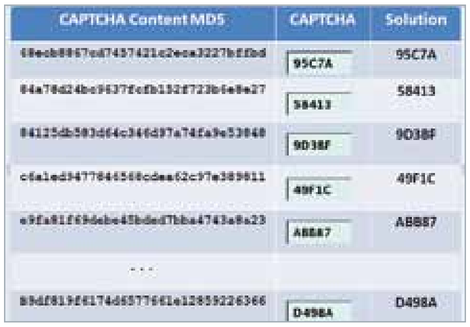
Alphanumeric Identifier	CAPTCHA	Solution
uJ8qsPvjxo6	95C7A	95C7A
9WzrowjPEqI	58413	58413
rm8SfvtEwpP	9D38F	9D38F
fespW5LVqNQ	49F1C	49F1C
dgLSB1CKJRJ	A8887	A8887
. . .		
QmJF3TQazcH	D498A	D498A

Figure 10. 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 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 and submitted to bypass the CAPTCHA restriction.



CAPTCHA Content MD5	CAPTCHA	Solution
68ecb8867cd7457421c2eca3227bffbd	95C7A	95C7A
84a78d24bc9637fcfb132f723b6e8e27	58413	58413
84125db503d64c346d97a74fa9e53848	9D38F	9D38F
c6a1ed3477846568cdea62c97e389811	49F1C	49F1C
e9fa81f69debe45bded7bba4743a8a23	AB887	AB887
. . .		
b9df819f6174d6577661e12859226366	D498A	D498A

Figure 11. 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 scenarios, 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 do not store the identifier or CAPTCHA solution in the HTTP session. When a form submission arrives, the CAPTCHA identifier is 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.

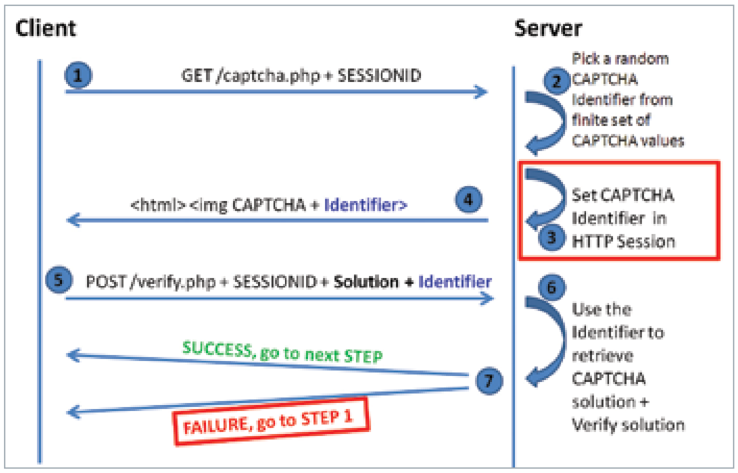

Client
Server
1
GET /captcha.php + SESSIONID
2
Pick a random CAPTCHA Identifier from finite set of CAPTCHA values
Set CAPTCHA Identifier in HTTP Session
3
<html> <img CAPTCHA + Identifier>
4
5
POST /verify.php + SESSIONID + Solution + Identifier
6
Use the Identifier to retrieve CAPTCHA solution + Verify solution
SUCCESS, go to next STEP
7
FAILURE, go to STEP 1

Figure 12. A secure CAPTCHA implementation scenario where the CAPTCHA key is stored in an HTTP session.

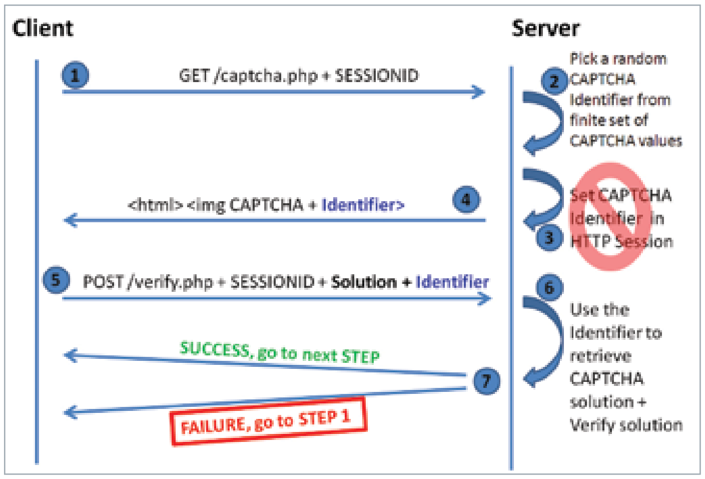

Client
Server
1
GET /captcha.php + SESSIONID
2
Pick a random CAPTCHA Identifier from finite set of CAPTCHA values
<html> <img CAPTCHA + Identifier>
4
Set CAPTCHA Identifier in HTTP Session
3
5
POST /verify.php + SESSIONID + Solution + Identifier
6
Use the Identifier to retrieve CAPTCHA solution + Verify solution
SUCCESS, go to next STEP
7
FAILURE, go to STEP 1

Figure 13. An insecure CAPTCHA implementation scenario where the CAPTCHA identifier is not stored in an HTTP session.

CAPTCHA fixation

A 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 the CAPTCHA identifier of their choice to the HTTP session and then use the pre-solved value to completely bypass CAPTCHA protection. The 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 to 3 (shown in image below) and manipulate the request in step 4 to request 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.

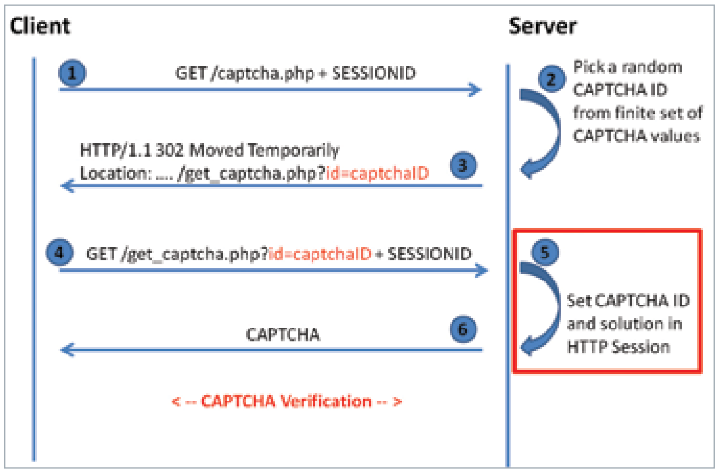

Client
Server
1
GET /captcha.php + SESSIONID
2
Pick a random CAPTCHA ID from finite set of CAPTCHA values
HTTP/1.1 302 Moved Temporarily
Location: .... /get_captcha.php?id=captchaID
3
4
GET /get_captcha.php?id=captchaID + SESSIONID
5
Set CAPTCHA ID and solution in HTTP Session
CAPTCHA
6
< -- CAPTCHA Verification -- >

Figure 14. A vulnerable implementation that leads to CAPTCHA fixation attacks.

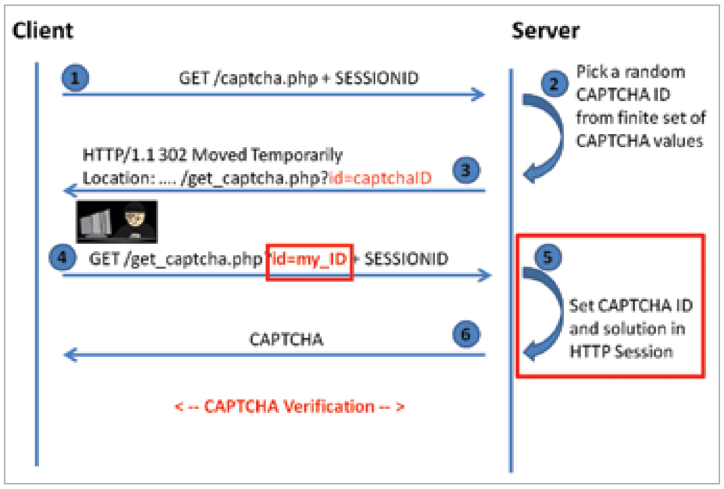

Client
Server
1
GET /captcha.php + SESSIONID
2
Pick a random CAPTCHA ID from finite set of CAPTCHA values
HTTP/1.1 302 Moved Temporarily
Location: .... /get_captcha.php?id=captchaID
3
4
GET /get_captcha.php?id=my_ID + SESSIONID
5
Set CAPTCHA ID and solution in HTTP Session
CAPTCHA
6
< -- CAPTCHA Verification -- >

Figure 15. An example of a CAPTCHA fixation attack where the attacker provides my_ID as the CAPTCHA identifier.

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

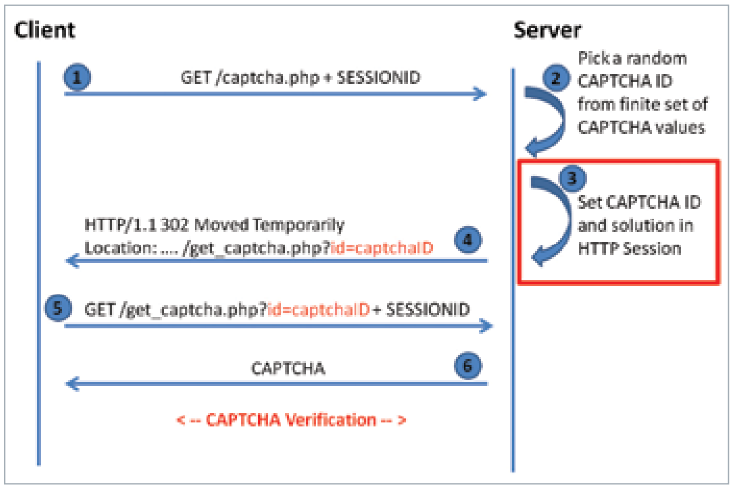

Client
Server
1
GET /captcha.php + SESSIONID
2
Pick a random CAPTCHA ID from finite set of CAPTCHA values
3
Set CAPTCHA ID and solution in HTTP Session
4
HTTP/1.1 302 Moved Temporarily
Location: .... /get_captcha.php?id=captchaID
5
GET /get_captcha.php?id=captchaID+ SESSIONID
6
CAPTCHA
< -- CAPTCHA Verification -- >

Figure 16. A secure implementation not vulnerable to CAPTCHA fixation attacks discussed above.

In-session CAPTCHA brute-forcing

In-session CAPTCHA brute-forcing was one of the most common flaws observed during the research. The widespread existence of this vulnerability is due to 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 a 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 CAPTCHA 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.

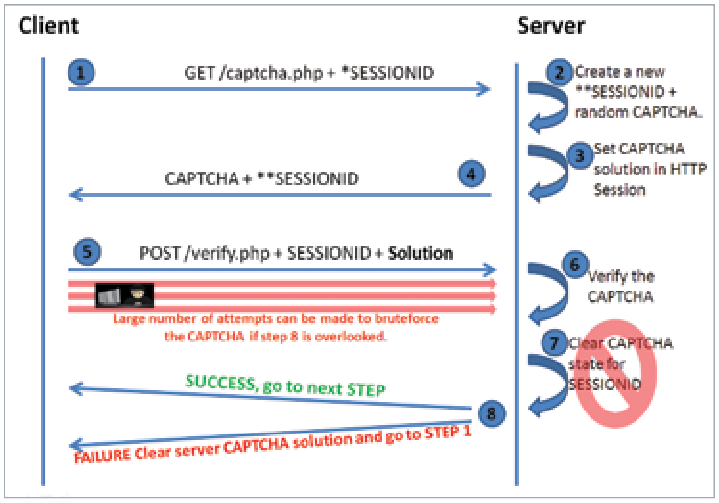

Client
Server
1
GET /captcha.php + *SESSIONID
2
Create a new **SESSIONID + random CAPTCHA.
3
Set CAPTCHA solution in HTTP Session
CAPTCHA + **SESSIONID
4
5
POST /verify.php + SESSIONID + Solution
6
Verify the CAPTCHA
Large number of attempts can be made to bruteforce the CAPTCHA if step 8 is overlooked.
7
Clear CAPTCHA state for SESSIONID
SUCCESS, go to next STEP
8
FAILURE Clear server CAPTCHA solution and go to STEP 1

Figure 17. An in-session CAPTCHA brute-force 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 OCR software. CAPTCHAs can be copied locally and solved offline using multiple OCR engines. Also, if the CAPTCHA implementation is vulnerable to the in-session CAPTCHA brute-force vulnerability discussed above, the OCR-assisted technique can be used to significantly reduce the number of attempts required to guess the correct solution in a live HTTP session. The following methods perform OCR-assisted CAPTCHA brute-force.

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 an in-session CAPTCHA brute-force attack. Here the OCR1 attempt will send rGsyg, causing a failure. The second OCR will send r6sy9, again causing the failure. Since both the solutions differ by two characters, they can be combined to find a correct solution r6syg.

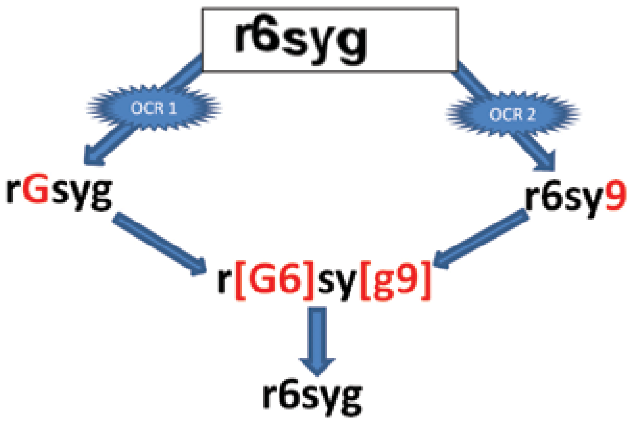


Figure 18. A CAPTCHA solution that combines results of two difference OCR engines.

2. After extracting text from CAPTCHA using an OCR engine, a selective brute-force can also be attempted. For example, let's assume that the OCR engine returns the result as TE5T12. The brute-force attempt begins by changing the first character "T" and retaining the values of the other five characters—and then moves on to the second character and so forth. After this, two characters can be brute-forced in tandem, followed by three, and throughout the entire length. This technique, like other brute-force techniques is high on time and resource requirements.

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," and so on. The effectiveness of this technique can be enhanced if the 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 the 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:

- A generic image preprocessing engine that can be configured as per the CAPTCHA type being analyzed
- Tesseract as its OCR engine to retrieve text from preprocessed CAPTCHAs
- Web proxy and custom HTTP headers support
- CAPTCHA statistical analysis support
- 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.
- White paper: http://www.mcafee.com/apps/view-all/publications.aspx?tf=foundstone&sz=10

Examples of TesserCap running on CAPTCHAs are shown below.

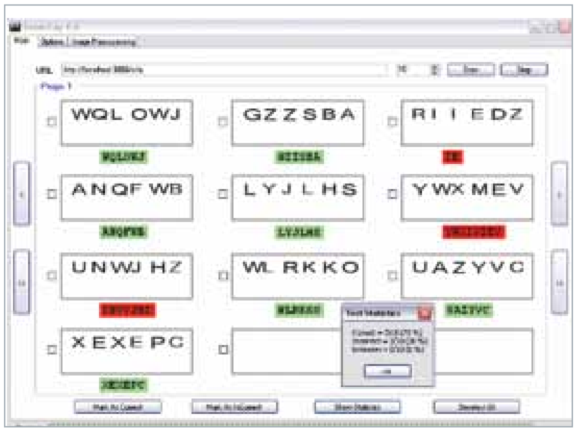


Figure 19. TesserCap sample run and test statistics.

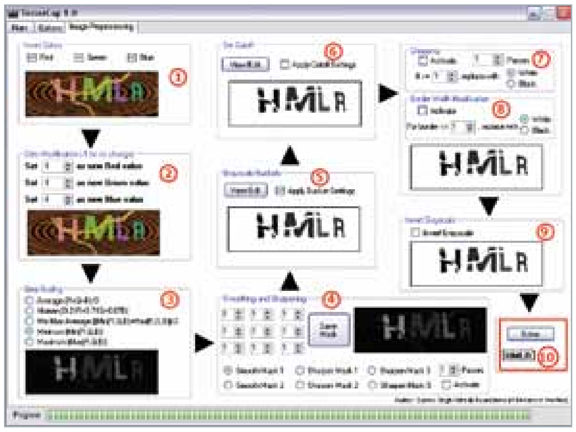


Figure 20. Successful text extraction after applying TesserCap's image preprocessing filters.

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 white paper.

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 oversight 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 McAfee Foundstone Professional Services. Gursev has been a guest speaker at conferences such as ToorCon, NullCon, and ClubHack. Gursev has authored the CAPTCHA testing tool TesserCap and an open source SSL cipher enumeration tool SSLSmart. Gursev has also developed several internal tools and web applications and enjoys coding in Ruby, Ruby on Rails, and C#.

About McAfee Foundstone Professional Services

McAfee Foundstone Professional Services 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, McAfee 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.

McAfee
An Intel Company

2821 Mission College Boulevard
Santa Clara, CA 95054
888 847 8766
www.mcafee.com