A common bypass pattern to exploit modern web apps

sonarsource

whoami

- My name is Simon Scannell
- Vulnerability Researcher for SonarSource
- Discover and disclose vulnerabilities
  - WordPress
  - Magento 2
  - MyBB
  - Zimbra
  - Linux kernel
- Likes to travel

Some of our recent work

- WordPress: **CSRF to Remote Code Execution** (CVE-2019-9787)
- Magento 2: **pre-auth Stored XSS to RCE** (CVE-2019-7877 & CVE-2019-7932)
- MyBB: **unprivileged Stored XSS to RCE** (CVE-2021-27889 & CVE-2021-27890)
- Zimbra: **Webmail compromise via eMail** (CVE-2021-35208 & CVE-2021-35209)

Some of our recent work

… continued

Some of these open source applications have been hardened through...

- Years of bug bounty programs
- Competition in the 0day market
- Static Analyzers
- Security Audits

… Yet, the bugs still occur. Why?

How the way we find web security bugs is changing

Dramatic increase in adaption of:

- Mitigations
- secure-by-default frameworks
- sanitization libraries
- ensuring overall security checks

How the way we find web security bugs is changing

- When a new mitigation or sanitization framework is deployed, we have to look for bugs in places that are not covered by the new mitigation
- This forces us to improve security research and invent new ways to find vulnerabilities

How the way we find web security bugs is changing

- Parser differentials
- Undefined or unclear components of a spec
- Time of check / Time of use

How the way we find web security bugs is changing

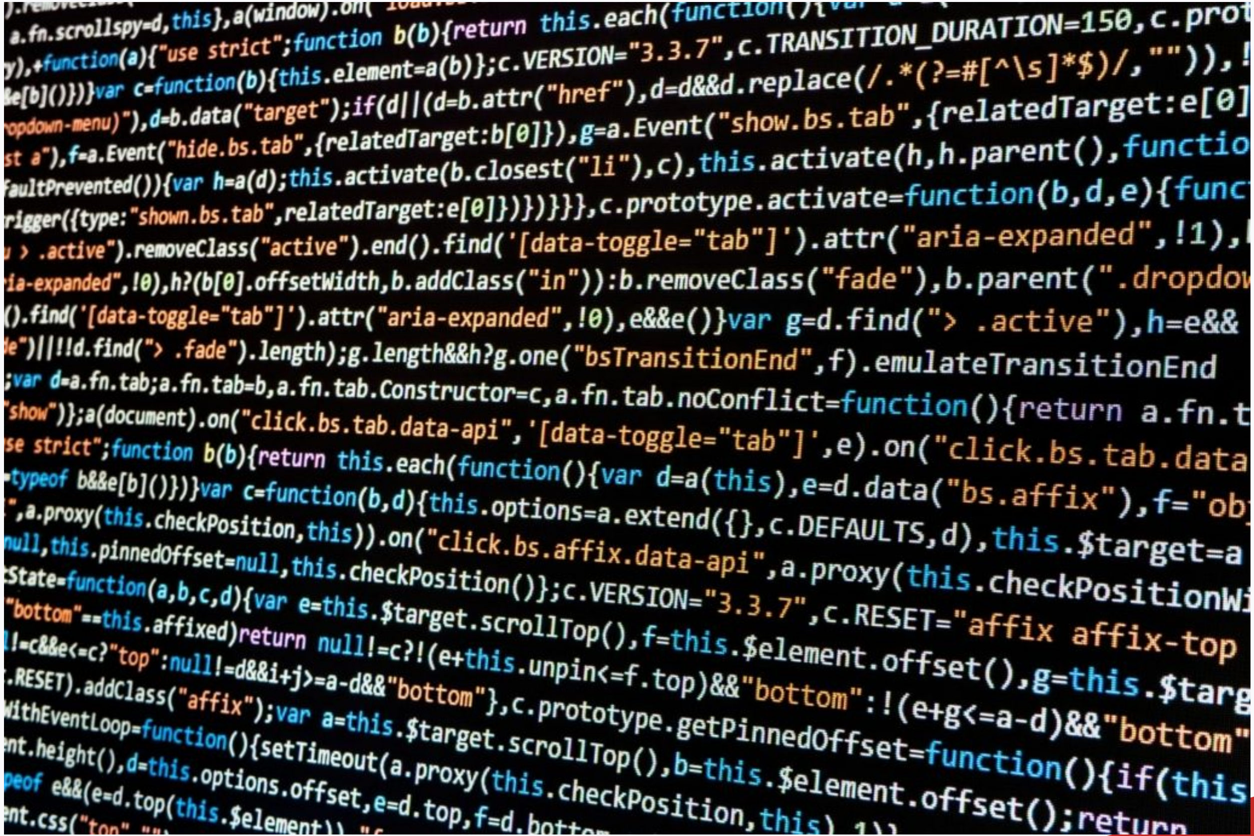
- Parser differentials
- Undefined or unclear components of a spec
- Time of check / Time of use

=> The same old vulnerabilities are still there, just the way we find them changes

How the way we find web security bugs is changing

- Parser differentials
- Undefined aspects of a spec
- Time of check / Time of use
- **Modification of sanitized data**

Let’s build a model for finding bugs



An abstraction of web security bugs

```
data = sanitize(user_input);
use(data);
```

An abstraction of web security bugs

```
// secure example
data = transform(user_input);
data = normalize(data);
data = sanitize(data);
use(data);
```

An abstraction of web security bugs

```
// secure example
data = transform(user_input);
data = normalize(data);
data = sanitize(data);
use(data);
```

Examples of transformations

- Converting shortcodes to HTML
  - [b]Hello Hacktivity![/b] => Hello Hacktivity!
- Modifying or adding HTML attributes to an HTML string
- Censoring of text
- Auto URL highlighting
- Language translations

An abstraction of web security bugs

```
  // secure example
data = transform(user_input);
data = normalize(data);
data = sanitize(data);
use(data);
```

Examples of normalizations

- Unicode normalization
- Path normalization
  - (`/var/www/html/../../../tmp` => `/tmp`)
  - Converting `\\` to `/` (Windows / Unix differences)
- Length truncations
- URL encoding / decoding

An abstraction of web security bugs

```
  // secure example
data = transform(user_input);
data = normalize(data);
data = sanitize(data);
use(data);
```

Examples of sanitization

- Extension checks
- HTML escaping
- Escaping inputs for SQL queries
- Validating input against allow-list

What could possibly go wrong?

Modification of sanitized data

```
// possibly insecure example
data = sanitize(user_input);
data = normalize(data);
data = transform(data);
use(data);
```

Modification of sanitized data

- When modification of sanitized data occurs, the effects of sanitization could be negated

Modification of sanitized data

Sanitization should always be the very last step before using data…

Modification of sanitized data

… However,

- Sanitized data tends to be trusted and used less carefully
- It isn’t always obvious if, how and where data is modified after it has been sanitized

Case studies

Case Study #1 - Zimbra Webmail

- Enterprise ready webmail solution
- Used by over 200.000 businesses, government and financial institutions
- Recent target of a 0day campaign by what is suspected to be a state-actor.
- Email bodies can contain arbitrary HTML code and must be carefully sanitized by a webmail solution

Case Study #1 - Zimbra Webmail

- We discovered a XSS vulnerability in the email body and a SSRF vulnerability that allowed stealing cloud provider credentials (e.g. AWS, Google Cloud)
- => One email is enough to potentially take over an email server of an organization

Demo

Case Study #1 - Zimbra Webmail

Sanitization

- Server-Side sanitization of HTML in email body
- Uses allow-list of HTML tags and attributes
- OWASP Java HTML Sanitizer is used
- We did not discover a bypass in this HTML sanitizer framework

Additionally, very strict encoding...

```
// "&#34;" is shorter than "&quot;"
REPLACEMENTS['"']  = "&#" + ((int) '"')  + ";";  // Attribute delimiter.
REPLACEMENTS['&']  = "&amp;";                    // HTML special.
// We don't use &apos; since it is not in the HTML&XML intersection
REPLACEMENTS['\''] = "&#" + ((int) '\'') + ";";  // Attribute delimiter.
REPLACEMENTS['+']  = "&#" + ((int) '+')  + ";";  // UTF-7 special.
REPLACEMENTS['<']  = "&lt;";                     // HTML special.
REPLACEMENTS['=']  = "&#" + ((int) '=')  + ";";  // Special in attributes.
REPLACEMENTS['>']  = "&gt;";                     // HTML special.
REPLACEMENTS['@']  = "&#" + ((int) '@')  + ";";  // Conditional compilation.
REPLACEMENTS['`']  = "&#" + ((int) '`')  + ";";  // Attribute delimiter.
```

Case Study #1 - Zimbra Webmail

- We realized we had to look for some place where the sanitized HTML output was modified
- We found a code snippet in a JavaScript file located in another repository that does just this

Case Study #1 - Zimbra Webmail

Normalization

- Emails can contain calendar invites
- If such an invite was present, the frontend JavaScript file was used to truncate the HTML description of the invite

Checking for a calendar invite

```
// first let's check for invite notes and use as content if present
if (hasInviteContent && !hasMultipleBodyParts) {
   if (!msg.getMimeHeader(ZmMailMsg.HDR_INREPLYTO)) {
      content = ZmIMsgView.truncateBodyContent(content, isHtml);
 }
}
```

Wrapping the content in DIV tags

```
// ...
var divEle = document.createElement("div");
divEle.innerHTML = content;
var node = Dwt.byId("separatorId",divEle);
// ...
return divEle.innerHTML;
```

Case Study #1 - Zimbra Webmail

Normalization

- Setting the user-controlled (and sanitized) HTML content to .innerHTML of a wrapping div **decodes** HTML entities in user-controlled data
- This does not lead to XSS directly but is important for the next step

Case Study #1 - Zimbra Webmail

Transformations

- The JavaScript front-end looks for *<form>* tags without an *action* attribute
- Emails can contain <form> tags and if no *action* attribute is present, the request is sent to the current location and thus CSRF attacks could be forged
- The Javascript code sets a default *action* attribute

Looking for <form> tags via regex

```
if (html.search(/(<form)(?![^>]+action)(.*?>)/g)) {
    html = html.replace(/(<form)(?![^>]+action)(.*?>)/ig,
        function(form) {
            if (form.match(/target/g)) {
             form = form.replace(/(<.*)(target=.*)(.*>)/g,
'$1action="SAMEHOSTFORMPOST-BLOCKED" target="_blank"$3');
            }
            else {
                form = form.replace(/(<form)(?![^>]+action)(.*?>)/g, '$1
action="SAMEHOSTFORMPOST-BLOCKED" target="_blank"$2');
            }
            return form;
        });
}
```

Let’s assume the following HTML in an email:

```
<hr
      align="<form > x"
      noshade="<script>alert(document.domain);//"
 />
```

After sanitization:

```
<hr
    align="&lt;form &gt; x"
    noshade="&lt;script&gt;alert(document.domain);//"
/>
```

After normalization:

```
<hr
      align="<form > x"
      noshade="<script>alert(document.domain);//"
   />
```

After regex replacements:

```
<hr
      align="<form action="SAMEHOSTFORMPOST-BLOCKED"
      target="_blank" > x"

noshade="<script>alert(document.domain);alert(document.coo
kie);//"></div>
```

Zimbra Summary

```
// server-side allow list
data = sanitize(user_input);

// .innerHTML normalization
data = normalize(data);

// <form> replacements
data = transform(data);

// display email to users
use(data);
```

Case Study #2 - WordPress

- At the time of writing, over 43% of websites use WordPress
- Has a comment form enabled by default, which can contain raw HTML code
- We discovered a chain of vulnerabilities leading to CSRF to RCE impact in default settings (CVE-2019-9787)
- At the time, SameSite cookies weren't enforced

Case Study #2 - WordPress

Background

- Comment form is not protected by a nonce
- Can contain raw HTML code, becomes sanitized
- Sanitization rules relaxed for admins, but still secure

Case Study #2 - WordPress

Sanitization

- WP sanitizer has been hardened over the years
- Uses an allow-list for HTML tags and attributes. One for admins and one for unauthenticated users
- We did not discover a bypass for the sanitizer

=> We looked for a place where comments are modified **after** the sanitization step

Case Study #2 - WordPress

Transformation

- Comments could contain *<a>* tags
- For SEO optimization purposes, WordPress modified the *rel* attribute, if present
- Only administrators could set *rel* attribute values. The CSRF indirection was thus needed

Case Study #2 - WordPress

Transformation

- WP parsed the <*a*> tags of the **already** sanitized comment and created key value pairs of their attribute values
- The <*a*> tags are then constructed back together...

The *rel* attribute modification:

```
if (!empty($atts['rel'])) {
 // the processing of the 'rel' attribute happens here
 // ...
 $text = '';
 foreach ($atts as $name => $value) {
      $text .= $name . '="' . $value . '" ';
 }
}
return '<a ' . $text . ' rel="' . $rel . '">';
```

Let's assume the following input:

```
<a title='XSS " onmouseover=evilCode() id=" ' rel='nofollow'>
```

After the *<a>* tag has been build back together:

```
<a rel="nofollow" title="XSS " onmouseover=evilCode() id=" ">
```

WordPress summary

```
// sanitize the comment
data = sanitize(user_input);

// process 'rel' attributes
data = transform(data);

// display the comment
use(data);
```

Demo

Case Study #3 - Magento 2

- Magento 2 stores handle hundreds of billions of dollars in annual transactions
- Popular target for hacking groups motivated by financial gain
- e.g. Magecart has been observed to utilize 0days against Magento 2 stores

Case Study #3 - Magento 2

Sanitization

- Low privileged employees could create XML sitemap files
- The filenames had to end with the *.xml* extension
- The filename and content were stored in the database
- When desired, the sitemap file could then be generated and written to disk
- The filename check was secure

Case Study #3 - Magento 2

Normalization

- The database column for the filename was limited to 32 characters
- The database driver class would truncate the filename to 32 characters if it was too long

Case Study #3 - Magento 2

Normalization

- `aaaaaaaaaaaaaaaaaaaaaaaaaaaa.php.xml` `would be truncated to` `aaaaaaaaaaaaaaaaaaaaaaaaaaaa.php`
- `RCE via file write`

Magento 2 summary

```
// enforce.xml extension
data = sanitize(user_input);

// truncate to 32 chars
data = normalize(data);

// write sitemap to disk
use(data);
```

Demo

Summary

- Abstracting vulnerabilities helps find bugs in highly complex and large code bases
- Abstraction helps keeping the big picture in mind when auditing big projects
- Look for places where sanitized data is modified
- Sanitization must always be the last step

Thank you!

- Blog posts and more at: blog.sonarsource.com
- We would love your help at SonarSource to find bugs in projects! Come talk to us :)
- Reach out to me on Twitter: @scannell_simon

Questions?