INF226 – Software Security

Håkon Robbestad Gylterud

2019–09-30

Samy

“but most of all, samy is my hero”

The Samy worm (aka `JS.spacehero`):

- Spread through MySpace profile pages.
- Fastest spreading worm ever:
  - Over one million infected pages within 20 hours!
- Mostly harmless.
- The worm’s author, Samy Kamkar, was raided by US Secret Service.

How did the Samy worm work?

The Samy word was a cross-site scripting worm:

- Samy found a way to put JavaScript on his own profile page.

The script spread the worm whenever someone visited an infected profile page.

How did the Samy worm work?

MySpace had some protections against this:

- Only allow: `<a>`, `<img>` and `<div>`
- Strip out any occurance of the word `javascript`

How did the Samy worm work?

MySpace had some protections against this:

- Only allow: `<a>`, `<img>` and `<div>`
- Strip out any occurance of the word `javascript`

But: JavaScript in CSS `style` attributes meant any tag could be used:

```
<div style="background:url('javascript:alert(1)')">
```

How did the Samy worm work?

MySpace had some protections against this:

- Only allow: `<a>`, `<img>` and `<div>`
- Strip out any occurance of the word `javascript`

But: JavaScript in CSS `style` attributes meant any tag could be used:

```
<div style="background:url('javascript:alert(1)')">
```

And: Browsers will actually also accept `java\nscript`:

```
<div style="background:url('java
script:alert(1)')">
```

More data could be hidden in other attributes:

```
<div id="mycode"
     expr="alert('hah!')"
     style="background:url('javascript:eval(document.all.mycode.expr)')">
```

More data could be hidden in other attributes:

```
<div id="mycode"
     expr="alert('hah!')"
     style="background:url('javascript:eval(document.all.mycode.expr)')">
```

The whole code of `JS.spacehero`, with explaination can be found here:

- `https://samy.pl/myspace/tech.html`

Same-origin policy

Origin

An **origin** is a triple:

- Protocol
- Domain
- Port number

Example: `https://www.uib.no/` gives:

- Protocol: `https`
- Hostname: www.uib.no
- Port number: 443

Same-origin policy and

The **same-origin policy** restricts scripts run in the browser to only *access resources from the same origin*.

Example: A script can only access cookies from the same origin.

Same-origin policy

The following URLs have the same origin:

- `http://www.geocites.com/bob/index.html`
- `http://www.geocites.com/eve/script.html`.

Cross-site scripting

Cross-site scripting

Web browsers insulate resources, such as cookies or JavaScript, from different *origins*.

Cross-site scripting (XSS) occurs when a web-server unintentionally serves JavaScript from an attacker to client browsers.

This allows attacker code to access resources from victim server origin.

Example

```
$username = $_GET['username'];
echo '<div class="header"> Welcome, ' . $username . '</div>';
```

Example

```
$username = $_GET['username'];
echo '<div class="header"> Welcome, ' . $username . '</div>';
```

Now `username` could contain JavaScript which can:

- Steal session cookies
- Trick the user to give their password by showing fake login screen
- Mine bitcoins
- ...

Vectors

How does the attacker inject script?

- User data from one user visible to another (Example: Samy)
- URL variables (There is an example in "Secure and resilient software development")
- User data from post requests
- Evaluating user data in client side script

XML HttpRequest

```
var xhttp = new XMLHttpRequest();
xhttp.onreadystatechange = function() {
    if (this.readyState == 4 && this.status == 200) {
       // Replace the content of "example" element
       // with HTML received from reqest:
       document.getElementById("example").innerHTML = xhttp.responseText;
    }
};
xhttp.open("GET", "newcontent", true);
xhttp.send();
```

XML HttpRequest

Scripts can make HTTP requests to the current origin.

This means that once an attacker has injected a script, he can do anything the user could do:

- GET pages
- POST forms
- ...

Example: The Samy worm used POST requests to update the profile, and add the user `samy` as a friend.

XSS prevention strategies

Filtering input

In general, trying to prevent malicious input is difficult:

- Blacklisting is bad security practice.
- The disallowed charcters (say, &, <, >, " , ' and /) are quite common.
- Client side checking is easy to circumvent.

Can work for simple things like: usernames or e-mail addresses.

Escaping output

How to escape data inserted into HTML depends on the context.

These situations must be handled differently:

- HTML body `<div>DATA</div>`
- Quoted attributes `<div id="DATA"></div>`
- Unquoted attributes `<div id=DATA></div>`
- Quoted strings in JavaScript: `alert('DATA')`
- CSS attribute values `background-color: DATA;`
- JSON data
- ...

Implementing the escaping is error prone. DO NOT DO THIS YOURSELF.

Escaping output

For a string placed inside an HTML element (example: `<div>DATA</div>`), we can do the following substitution:

```
&  →  &amp;
<  →  &lt;
>  →  &gt;
"  →  &quot;
'  →  &#x27;
/  →  &#x2F;
```

Use your web-framework's well-tested implementation for this.

The DON'Ts

There are a number of places where one should just avoid inserting untrusted data.

Avoid inserting untrusted data in tag names

```
<NEVER PUT UNTRUSTED DATA HERE... href="/test" />
```

Avoid inserting untrusted data in attribute names

```
<div ...NEVER PUT UNTRUSTED DATA HERE...=test />
```

Avoid inserting untrusted data in scripts

```
<script>...NEVER PUT UNTRUSTED DATA HERE...</script>
```

Avoid inserting untrusted data directly in CSS

```
<style>
...NEVER PUT UNTRUSTED DATA HERE...
</style>
```

More gotchas

- `{ background-url : "javascript:alert(1)"; }`
- `{ text-size: "expression(alert('XSS'))"; }`

Read: OWASP XXS cheat sheet

Text formating

Problem: We want to let the user format their input, but worry about letting them use HTML because of XSS.

Text formating

Problem: We want to let the user format their input, but worry about letting them use HTML because of XSS.

Solutions:

- HTML sanitisers (Example: OWASP AntiSamy project)
- Using another markup language (Markdown, BBCode) with safe conversion to HTML.
  - Markdown allows literal HTML, which must be sanitized.
  - Many BBCode imlementations do nothing to prevent XSS.

Text formating

Problem: We want to let the user format their input, but worry about letting them use HTML because of XSS.

Solutions:

- HTML sanitisers (Example: OWASP AntiSamy project)
- Using another markup language (Markdown, BBCode) with safe conversion to HTML.
  - Markdown allows literal HTML, which must be sanitized.
  - Many BBCode imlementations do nothing to prevent XSS.

Notice: Even graphical formatting tools must represent the formatting in some way, and can be just as vulnerable to XSS as code-based ones.

CWE-352: Cross-Site Request Forgery (CSRF)

CWE-352: Cross-Site Request Forgery (CSRF)

Web form, as sent to browser:

```
<form action="/url/profile.php" method="post">
  <input type="text" name="firstname"/>
  <input type="text" name="lastname"/>
  <br/>
  <input type="text" name="email"/>
  <input type="submit" name="submit" value="Update"/>
</form>
```

Server-side handling:

```
session_start();
// Check session cookie
if (! session_is_registered("username")) {
    echo "invalid session detected!";
    [...]
    exit;
}
update_profile();

function update_profile {
  SendUpdateToDatabase($_SESSION['username']
                       , $_POST['email']);
[...]
  echo "Your profile has been updated.";
}
```

Meanwhile on a different website...

https://attacker.com/attack/:

```
<SCRIPT>
function SendAttack () {
  form.email = "attacker@example.com";
  form.submit();
}
</SCRIPT>

<BODY onload="javascript:SendAttack();">
<form action="http://victim.example.com/profile.php"
      id="form" method="post">
  <input type="hidden"
         name="firstname" value="Funny">
  <input type="hidden"
         name="lastname" value="Joke">
  <br/>
  <input type="hidden" name="email">
</form>
```

What happens if the user visist the attacker's web-site while logged in to `victim.example.com`?

CRSF prevention: Stored tokens

```
<form action="/url/profile.php" method="post">
  <input type="hidden"
         name="csrftoken" value="XolHzuGYZcLw7PQ2qv7WXC1C3dzYyxCg">
  <input type="text" name="firstname"/>
  <input type="text" name="lastname"/>
  <br/>
  <input type="text" name="email"/>
  <input type="submit" name="submit" value="Update"/>
</form>
```

Muddiest point

Answer on `mitt.uib.no`

HTTP Strict Transport Security

HTTP Strict Transport Security (**HSTS**) is a HTTP response header:

```
Strict-Transport-Security: max-age=31536000;
                           includeSubDomains;
                           preload.
```

It tells the client to **always use HTTPS with this domain**.

HSTS can be preloaded into browsers.

HSTS

HSTS protects against:

- User accepting a bad certificate
- Downgrade to plaintext HTTP
- Old HTTP bookmarks

Note: if your domain is on the preload list, you cannot change back to HTTP — clients will no longer accept it.